A Comprehensive Approach to Power Management in Embedded Systems

Antônio Augusto Frölich Laboratory for Software and Hardware Integration (LISHA)
Federal University of Santa Catarina (UFSC)
88040-900 Florianópolis - SC - Brazil
guto@lisha.ufsc.br

***Abstract*—In this article, power management is addressed in the context of embedded systems from energy-aware design to energy-efficient implementation. A set of mechanisms specifically conceived for this scenario is proposed, including: a power management API defined at the level of user-visible system components, the infrastructure necessary to implement that API (namely, battery monitoring, accounting, auto-suspend, and auto-resume), an energy-event propagation mechanism based on *Petry Nets* and implemented with *Aspect-Oriented Programming* techniques, and an autonomous power manager build upon the proposed API and infrastructure. These mechanisms are illustrated and evaluated by a realistic embedded system that is also used to sustain comparisons with other proposals at each of the considered levels. As a result, this article has its main contribution on the introduction of a comprehensive and systematic way to deal with power management issues in embedded systems.**

I. Introduction

Power management is a subject of great relevance for two large groups of embedded systems: those that operate disconnected from the power grid, taking their power supply from batteries, photovoltaic cells, or from a combination of technologies that yet impute limitations on energy consumption; and those that face heat dissipation limitations, either because they depend on high-performance computations or because they are embedded in restrictive environments such as the human body. Both classes of embedded systems can benefit from power management techniques at different levels, from energy-efficient peripherals (e.g. sensors and actuators), to adaptive digital systems, to power-aware software algorithms.

Historically, power management techniques rely on the ability of certain components to be turned on and off dynamically, thus enabling the system as a whole to save energy when those components are not being used [1]. Only more recently, techniques have been introduced to enable some components to operate at different energy levels along the time [2]. Multiple operational modes and *Dynamic Voltage Scaling* (DVS) are examples of such techniques that are becoming commonplace for microprocessors. Unfortunately, microprocessors are seldom the main energy drain in embedded systems—peripherals are—, so traditional on/off mechanisms are sill of great interest.

Even concerning microprocessors, which are the cores of the digital systems behind any embedded system, current power management standards, such as APM and ACPI, only define a software/hardware interface for power management, mostly disregarding management strategies and fully ignoring the designer knowledge about how energy is to be used—and therefore how it can be saved—in the system. Moreover, these standards evolved in the context of portable personal computers and usually do not fit in the limited-resource scenario typical of embedded systems. Other initiatives in the scope of embedded operating systems, some of which will be discussed later in this article, introduce power management mechanisms at the level of hardware abstraction (viz. HAL), demanding programmers to go down to that level in order to manage energy. This compromises several aspects of software quality, portability and time-to-marketin particular. Yet, others assume that the operating system is capable to doing power management by itself, defining policies and implementing automatic mechanisms to enforce them.

We believe that power management in embedded systems could be made far more effective if designers were provided with adequate means to express their knowledge about the power characteristics of the system directly to the power manager. In contrast to general-purpose systems, embedded systems result from a design process that is usually driven by the requirements of a single application. Assuming that the traditional autonomous power management mechanisms found in portable computers will ever be able to match the designers expertise about such tailor-made systems is unrealistic. Furthermore, power management for portable computers is mostly conceived around the idea of maximizing operating time for a given energy budget. We believe that many embedded systems would prefer to have it modeled to ensure a minimum *system lifetime*.

In this article, we introduce a set of mechanisms that enable designers to directly influence, or even control, the power management strategy for the system. These mechanisms have been modeled around typical embedded system requirements, including small footprint, little overhead and non-interference with real-time constraints. They are:

- A power management API defined at the level of user-visible system components (e.g. files, sockets, and processes) that supports semantic energy modes (i.e. *off*, *stand-by*, *light*, and *full*), arbitrary energy modes (i.e. device-specific), and dynamic voltage scaling.
- A power management infrastructure for system components, with accounting, auto-suspend and auto-resume mechanisms, implemented around *Aspect-Oriented Programming* (AOP) concepts and formalized through *Petry*

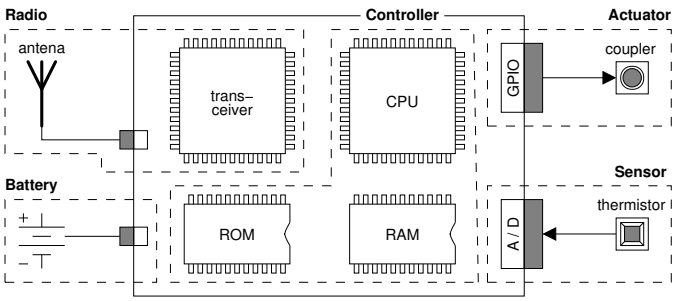

Figure 1. Example monitoring system block diagram.

Nets.

- An autonomous power manager, whose policies can be configured, statically or dynamically, and whose decisions take in consideration the interactions between applications and system done through the management API, thus enabling applications to override specific policies.

The reminder of this text discusses the design of these three mechanisms, their implementation in the EPOS Project, experiments carried out to corroborate the proposal, a discussion about related work, and is closed with a reasoning about the proposed mechanisms.

II. POWER MANAGEMENT API

In order to introduce a discussion about power management *Application Programming Interfaces* (API), let us first recall how energy consumption requirements arise during the design of an energy-aware embedded system and how they are usually captured. In such systems, designers look for available energy-efficient components and, eventually, specify new components to be implemented. During this process, they inherently acquire knowledge about the most adequate operating strategy for each component and for the system as a whole. Whenever the identified strategies are associated to modifications in the energy level of a given component, this can be captured in traditional design diagrams, such as sequence, activity and timing, or by specific tools [3].

Now let us consider the design of a simple application, conceived specifically to illustrate the translation of energy constraints from design to implementation. This application realizes a kind of remote monitoring system, capable of sensing a given property (e.g. temperature), reporting it to a control center, and reacting by activating an actuator (e.g. external cooler) whenever it exceeds a certain limit. Interaction with the control center is done via a communicator (e.g. radio). The system operates on batteries and must run uninterruptedly for one year. A block diagram of the system is shown in figure 1.

The application is modeled around four tasks whose behavior is depicted in the sequence diagrams of figures 2 through 5: `Main` allocates common resources and creates threads to execute the other three tasks; `Monitor` is responsible for periodic temperature monitoring (every second), for reporting the current temperature to the control center (every 10 seconds), and, in case the temperature threshold is exceeded, for triggering the emergency handling thread; `Trigger` is responsible

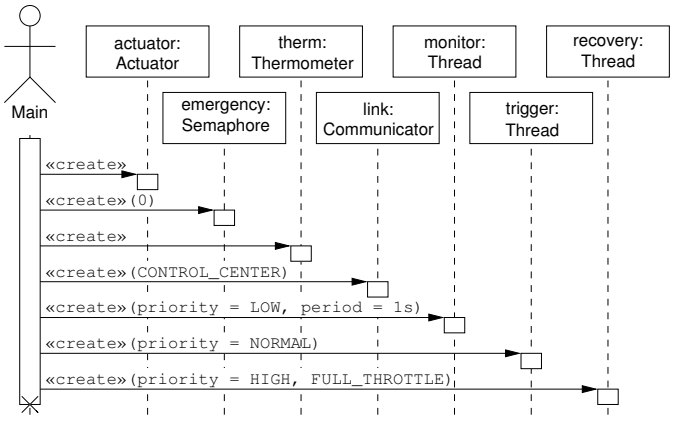

Figure 2. Main thread sequence diagram with power management actions.

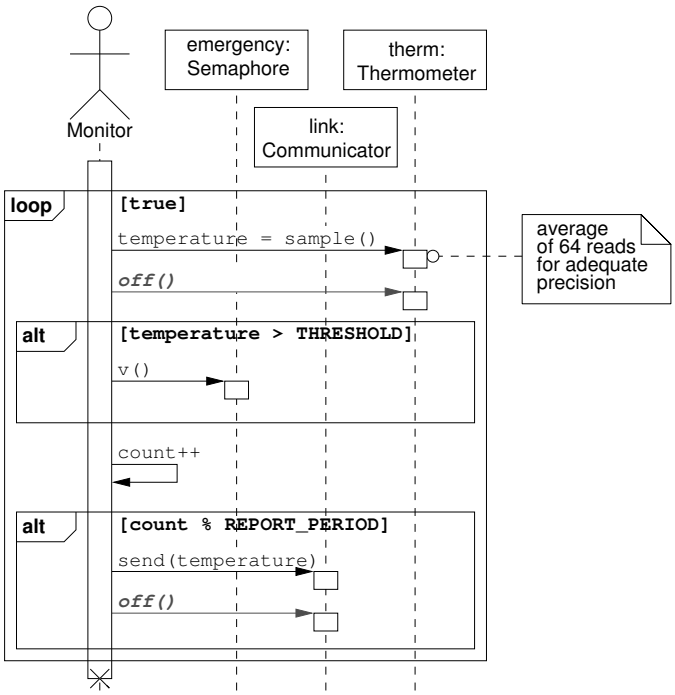

Figure 3. Monitor thread sequence diagram with power management actions.

for triggering the emergency handling thread on command of the control center; and `Recovery`, the emergency handling thread, is in duty of firing an one-shot actuator intended at restoring the temperature to its normal level. Coordination is ensured by properly assigning priorities to threads and by the `emergency` semaphore.

In the sequence diagrams of figures 2 through 5, energy-related actions captured during design are expressed by messages and remarks. For instance, the knowledge that the `Thermometer` component uses a low-cost thermistor and therefore must perform 64 measurements before being able to return a temperature value within the desired precision is expressed as a note associated with the method (figure 3). In order to be energetically efficient, the circuitry behind `Thermometer` (i.e. ADC and thermistor) should be kept active during all the measurement cycle, thus avoiding repetitions

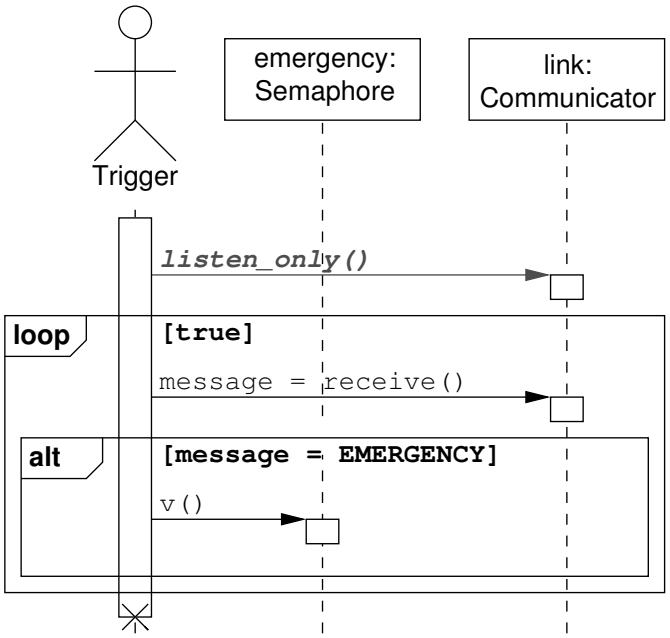

Figure 4. Trigger thread sequence diagram with power management actions.

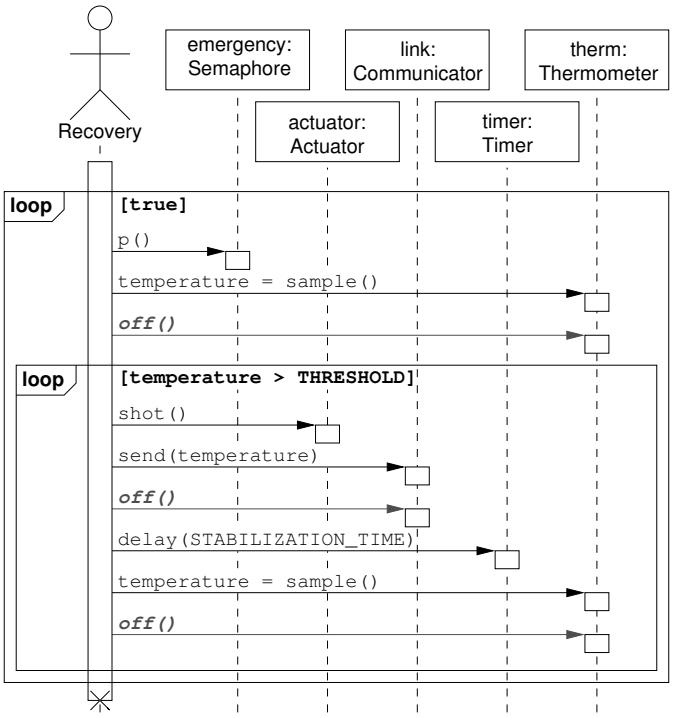

Figure 5. Recovery thread sequence diagram with power management actions.

Block	State	Cur. (μA)	Duty Cycle	Cons. (μAh)	Cons. (mAh)	Energy (J)
Controller	stand-by	8	98.69%	7.90	0.0344	0.3720
	idle	470	0%	0.00		
	min. freq.	1,950	1.30%	25.35		
	max. freq.	12,000	0.01%	1.20		
Radio	off	0.2	0.20%	0.00	0.3722	4.0199
	listen	96	97.78%	93.87		
	receive	9,600	< 0.01%	0.96		
	transmit	13,800	2.01%	277.38		
Sensor	off	1	99.97%	1.00	0.0014	0.0147
	at 25 C	900	0.02%	0.18		
	at 40 C	1,775	< 0.01%	0.18		
Actuator	on	2,200	< 0.01%	0.22	0.0002	0.0024
	off	0	99.99%	0.00		
Total consumption per hour					**0.4082**	**4.4089**

Table I
EXAMPLE MONITORING SYSTEM ENERGY CONSUMPTION ESTIMATES.

of the electrical stabilization phase[1].

The diagrams also show power management hints for the `Communicator` component (figures 3, 4, and 5), which is mostly used to listen for a message from the control center and thus can be configured on a listen-only state for most of the time. And, indirectly, also for the `CPU` component, which must operate in the maximum frequency while running the `Recovery` thread, but can operate in lower frequencies for the other threads (figure 2). With this information in hand, the system can be implemented to be more efficient in terms of energy, either by the programmer himself or by means of an automatic power manager.

Considering the functional properties described so far and the execution period of each thread, it is possible to estimate duty cycles for each of the major components in the example system. This information can then be combined with energy consumption estimates of individual components to calculate the power supply required by the system. This procedure is summarized in table I, which shows hypothetical energy consumption estimates, duty cycles and energy consumption for the four major components in the system[2].

In order to match the requirement of operating uninterruptedly for one year, the system would demand a battery capable of delivering approximately 3576 mAh (at 3 V). For comparison, the same system operating with all components constantly active, that is, without any power saving strategy, would require about 142 Ah, almost 40 times more.

A. Current APIs

Few systems targeting embedded computing can claim to deliver a real Power Management API. Nevertheless, most systems do deliver mechanisms that enable programmers to directly access the interface of some hardware components. These mechanism, though not specifically designed for power management, can be used for that purpose at the price of binding the application to hardware details.

[1]In a typical ADC/thermistor configuration, the electrical stabilization phase accounts up to 98% of the measurement cycle, both in terms of time and in terms of energy [4].

[2]Estimates were based on the Mica2 sensor node [5]

```
int thermometer_sample() {
    // Open ADC power attribute on sysfs
  struct sysfs_attribute * adc_power
    = sysfs_open_attribute(
        "/sys/devices/i2c/AD8493/power/state"
      );

  int accumulator = 0;
  int value;

  // Switching to power state 0 (ON)
  sysfs_write_attribute(adc_power, "0", 1);
  for(int i = 0; i < 64; i++) {
      // Read ADC result
    ad8493_get(adc_device, &value);
    accumulator += value;
  }
  // Switching to power state 3 (OFF)
  sysfs_write_attribute(adc_power, "3", 2);

  // Convert reading into celcius degrees
  return raw_to_celcius(accumulator / 64);
}
```

Figure 6. `Thermometer::sample()` method implementation for μCLINUX.

μCLINUX, like many other UNIX-like systems, does not feature a real power management API. Some device drivers provide power management functions inspired on ACPI. Usually these mechanisms are intended to be used by the kernel itself, though a few device drivers export them via the `/sys` or `/proc` file systems, thus enabling applications to directly control the operating modes of associated devices.

The source code in figure 6 is a user-level implementation of the `Thermometer::sample()` method of our example monitoring system. In this implementation, programmers must explicitly identify the driver responsible for the ADC to which the thermistor is connected. Besides the overhead of interacting with the device driver through the `/sys` file system, μCLINUX PM API creates undesirable dependencies and would fail to preserve the application in case the thermistor gets connected to another ADC channel or in case the ADC in the system gets replaced by another model.

TINYOS, a popular operating system in the wireless sensor network scene, allows programmers to control the operation of hardware components through a low-level, architecture-dependent API. Though not specifically designed for power management purposes, this API ensures direct access to the hardware and thus can be used in this sense. When compared to μCLINUX, TINYOS delivers a lighter mechanism, more adequate for most embedded system, yet suffers from the same limitations in respect to usability and portability. The use of TINYOS hardware control API for power management is illustrated in figure 7, which depicts the implementation of the `Trigger` thread of our example.

MANTIS, features a POSIX-inspired API that abstracts hardware devices as UNIX special files. Differently of μCLINUX and TINYOS, however, MANTIS does not propose that API to be used for power management purposes: internal mechanisms automatically deactivate components that have not been used

```
// ...
  // When initializing system
  event void Boot.booted() {
    // ...
    // Put radio in listening mode
    call RadioControl.start();
    // ...
  }

  // When data is received
  event message_t* Receive.receive(
                     message_t* bufPtr,
                     void* payload, uint8_t len
                   ) {
    if (len != sizeof(radio_sense_msg_t))
      return bufPtr;
    else {
      radio_sense_msg_t* rsm =
              (radio_sense_msg_t*) payload;
      if (rsm->data == EMERGENCY) {
        // Turn radio off
        // Someone has to turn it on again later
        RadioControl.stop();
        emergency_semaphore++;
      }

      return bufPtr;
    }
  }
// ...
```

Figure 7. `Trigger` thread implementation for TINYOS.

```
for (int i = 0; i < 64; i++) {
  // Read from device
  dev_read (DEV_MICA2_TEMP, &data, 1);
  accumulator += data;
}
// MantisOS device driver turns sensor
// ON and OFF for every reading
```

Figure 8. Thermometer::sample() method implementation for MANTIS.

for a given time, or perform an "on-act-off" scheme, thus implementing a sort of OS-driven power manager. This strategy can be very efficient, but makes it difficult for programmers to express the knowledge about energy consumption acquired during the design process. This is made evident in the implementation of the `Thermometer::sample()` depicted in figure 8. Unaware of the precision required for the temperature variable, MANTIS cannot predict that the ADC is being used in a loop and misses the opportunity to avoid the repetition of the expensive electrical stabilization phase of the thermometer operation.

Some systems assume that architectural dependencies are intrinsic to the limitations of typical embedded systems, however, this is exactly the share of the computing systems market that could benefit from a large diversity of suppliers [6] and therefore would profit from quick changes from one architecture to another. This, in addition to the fact that current APIs do not efficiently support the expression of design knowledge during system implementation, led us to propose a new PM API.

B. Proposed API

The Power Management API proposed here arose from the observation that currently available APIs require application programmers to go down to the hardware whenever they want to manage power, inducing unnecessary and undesirable architectural dependencies between application and the hardware platform. In order to overcome these limitations, we believe a PM API for embedded systems should present the following characteristics:

- Enable direct application control over energy-related issues, yet not excluding delegation or cooperation with an autonomous power manager;
- Act also at the level of user-visible components, instead of being restricted to the level of hardware component interfaces, thus promoting portability and usability;
- Be suitable for both application and system programming, thus unifying power management mechanisms and promoting reuse;
- Include, but not be restricted to, semantic modes, thus enabling programmers to easily express power management operations while avoiding the limitations of a small, fixed number of operating modes (as is the case of ACPI).

With these guidelines in mind, we developed a very simple API, which comprises only two methods, and an extension to the methods responsible for process creation. They are:

```
Power_Mode power(void)
void power(Power_Mode)
```

The first method returns the current power mode of the associated object (i.e., component), while the second allows for mode changes. Aiming at enhancing usability, four power modes have been defined with semantics that must be respected for all components in the system: *off*, *stand-by*, *light* and *full*. Each component is still free to define additional power modes with other semantics, as long as the four basic modes are preserved. Enforcing universal semantics for these power modes enables application programmers to control energy consumption without having to understand the implementation details of underlying components (e.g., hardware devices). Allowing for additional modes, on the other hand, enables programmers to precisely control the operation of special components, whose operation transcend the predefined modes.

The introduction of these methods in user-visible components such as files and sockets certainly requires some sort of propagation mechanism and could itself introduce undesirable dependencies. We describe a strategy to implement them using a combination of *Aspect-Oriented Programming* techniques and *Hierarchical Petry Nets* later in section III. For now, lets concentrate on the characterization of the API, not the mechanisms behind it.

Table II summarizes the semantics defined for the four universal operating modes. A component operating in mode *full* provides all its services with maximum performance, possibly consuming more energy than in any other mode. Contrarily, a component in mode *off* does not provide any service and also does not consume any energy. Switching a component from *off* to any other power mode is usually an expensive operation, specially for components with high initialization and/or stabilization times. The mode *stand-by* is an alternative to *off*: a component in *stand-by* is also not able to perform any task, yet, bringing it back to *full* or *light* is expected to be quicker than from mode *off*. This is usually accomplished by maintaining the state of the component "alive" and thus implies in some energy consumption. A component that does not support this mode natively must opt between remaining active or saving its state, perhaps with aid from the operating system, and going off.

Mode	FULL	LIGHT	STANDBY	OFF
Energy	high	low	very low	none
Functionality	all	all	none	none
Performance	high	low	NA	NA
Wake up Delay	NA	very low	high	very high

Table II
SEMANTIC POWER MODES OF THE PROPOSED PM API.

Defining the semantics for mode *light* is not so straightforward. A component in this mode must deliver all its services, but consuming the minimum amount of energy. This definition brings about two important considerations. First, if there is a power mode in which the component is able to deliver all its services with the same quality as if it was in mode *full*, then this should be mode *full* instead of *light*, since it would make no sense to operate in a higher consumption mode without arguable benefits. Hence, mode *light* is often attained at the cost of performance (e.g., through DVS). This, in turn, brings about a second consideration: for a real-time embedded system, it would be wrong to state that a component is able to deliver "all its services" if the added latency is let to interfere with the time requirements of applications. Therefore, mode *light* shall not be implicitly propagated to the CPU component. Programmers must explicitly state that they agree to slow down the processor to save energy, or a energy-aware, real-time scheduler must be deployed [7].

Besides the four operating modes with predefined, global semantics, a component can export additional modes through the API. These modes are privately defined by the component based on its own peculiarities, thus requiring the client components to be aware of their semantics in order to be deployed. The room for extensions is fundamental for hardware components with many operating modes, allowing for more refined energy management policies. For instance, the *listen-only* radio mode in our example (see figure 4) relies on such an extension.

The proposed API also features the concept of a `System` pseudo-component, which can be seen as a kind of aggregator for the actual components selected for a given system instance. The goal of the `System` component is to aid programmers to express global power management actions, such as putting the whole system in a given operating mode, perhaps after having defined specific modes for particular components.

Figure 9 presents all these interaction modes in a UML communication diagram of a hypothetical system instance. The application may access a global component (`System`) that has

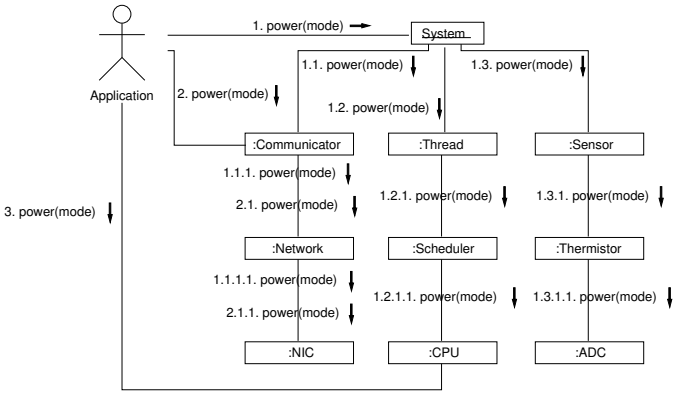

Figure 9. Power Management API utilization example.

knowledge of every other component in the system, triggering a system-wide power mode change (execution flow 1). The API can also be accessed to change the operating mode of a group of components responsible for the implementation of a specific system functionality (in this example, communication functionality through execution flow 2). The application may also access the hardware directly, using the API available in the device drivers, such as *Network Interface Card* (`NIC`), `CPU`, `ADC` (in the figure, application is accessing the CPU through the execution flow 3). The API is also used between the system's components, as can be seen in the figure.

In a system that realizes the proposed API, the monitoring system introduced earlier could be implemented as show in figure 10, a rather direct transcript of the sequence diagrams of figures 2 through 5.

III. Power Management Infrastructure

From the discussion about traditional power management APIs for embedded systems in the previous section, we can infer that the infrastructure behind those APIs are mostly based on features directly exported by hardware components and do not escape from the software/hardware interface. As a matter of fact, the power management infrastructure available in modern hardware components is far more evolved than the software counterpart, which not rarely is restricted to mimic the underlying hardware. For example, XScale microprocessors support a wide range of operating frequencies that allow for fine grain DVS. They also feature a Power Management Unit that manages idle, sleep, and a set of quick wake-up modes, and a Performance Monitoring Unit that furnishes a set of event counters that may be configured to monitor occurrence and duration of events.

Power management mechanisms can benefit from such hardware features to implement context-aware power management strategies. Device drivers for the operating systems discussed in the previous section, however, do not make use of most of these features. In order to take advantage of them, application programmers must often implement platform-specific primitives by themselves. This, besides being out-of-scope for many embedded application developers, will certainly hinder portability and reuse. The same can be observed with peripherals such as wireless network cards, which often provide a large

```
Thermometer thermometer;
Actuator actuator;
Semaphore emergency(0);
Communicator link(CONTROL_CENTER);

int main() {
  new Thread(&recovery, HIGH_PRIO, NO_DVS);
  new Thread(&trigger, NORMAL_PRIO);
  new Periodic_Thread(&monitor, LOW_PRIO, 1000000);
  return 0;
}
void monitor() {
  int count = 0;
  while(true) {
    int temperature = thermometer.get();
    thermometer.power(OFF);
    if (temperature > THRESHOLD)
      emergency.v();
    if (!(++count % 10)) {
      link.write(temperature);
      link.power(OFF);
    }
    wait_for_next_cycle;
  }
}
void trigger () {
  while(true) {
    link.power(LISTEN_ONLY);
    if ( link.read() == EMERGENCY) // blocks calling thread
      emergency.v();
  }
}
void recovery() {
  while(true) {
    emergency.p();
    int temperature = thermometer.get();
    thermometer.power(OFF);
    while(temperature > THRESHOLD) {
      actuator.shoot();
      link.write(temperature);
      link.power(OFF);
      delay(STABILIZATION_TIME);
      temperature = thermometer.get();
      thermometer.power(OFF);
    }
  }
}
```

Figure 10. Example monitoring system implementation using the proposed PM API.

set of configurable characteristics that are not well explored.

In order to support both application-directed and autonomous power management strategies, the infrastructure necessary to implement the proposed API must feature the following services:

a) Battery monitoring: monitoring battery charge at runtime is important to support power management decisions, including generalized operating mode transitions when certain thresholds are reached; some systems are equipped with intelligent batteries that inherently provide this service, others must tackle on techniques such as voltage variation along discharge measured via an ADC to estimate the energy still available to the system [8].

b) Accounting: tracking the usage of components is fundamental to any dynamic power management strategy; this can be accomplished by *event counters* implemented either in software or in hardware; some hardware platforms feature event counters that are usually accessible from software, thus allowing for more precise tracking; in some systems, for which energy consumption measurements have been carried out on a per-component basis, it might even be possible to perform energy accounting based on these counters [9].

c) Auto-resume: a component that has been set to an energy-saving mode must be brought back to activity before it can deliver any service; in order to relieve programmers from this task, most infrastructures usually implement some sort of "auto-resume" mechanism, either by inserting mode verification primitives in the method invocation mechanism of components or by a trap mechanism that automatically calls for operating system intervention whenever a inactive component is accessed.

d) Auto-suspend: with accounting capabilities in hand, a power management infrastructure can deliver "auto-suspend" mechanisms that automatically change the status of components that are not currently being used to energy-saving modes such as *stand-by* or *off*; however, suspending a component and resuming it shortly after will probably spend more energy than letting it to continue in the original mode, therefore, the heuristics used to decide which and when components should be suspended is one of the most important issues in the field and is now subject to intense research [10], [11], [12], [13], [14].

Our proposed power management API allows interaction between the application and the system, between system components and hardware devices, and directly between application and hardware. Thus, in order to realize this API, each software and hardware component in our system must be adapted to provide the above listed services.

A. Implementation through Aspect Programs

Aspect-Oriented Programming (AOP) [15] allows non-functional properties (e.g. identification, synchronization, sharing control) to be modeled separately from the components they affect. Associated implementation techniques enable the subsequent implementation of such properties as aspect programs that are kept isolated from components, thus preventing a generalized proliferation of manual, error-prone modifications across the system. As a non-functional property, power management fits well into this paradigm.

EPOS [16], our testbed system, supports AOP through a C++ construct called *Scenario Adapter*. Scenario adapters enable aspects to be implemented as ordinary C++ programs that are subsequently applied to component code during system compilation, thus eliminating the need for external tools such as aspect weavers. Figure 11 shows the general structure of a scenario adapter. The aspect programs `Aspect` implement their duties as the `Scenario_Adapter` intercepts every invocation of a component operation by its `Clients` and embraces it within a `enter`/`leave` pair. The `Scenario` construct collects these aspect programs, each with its own

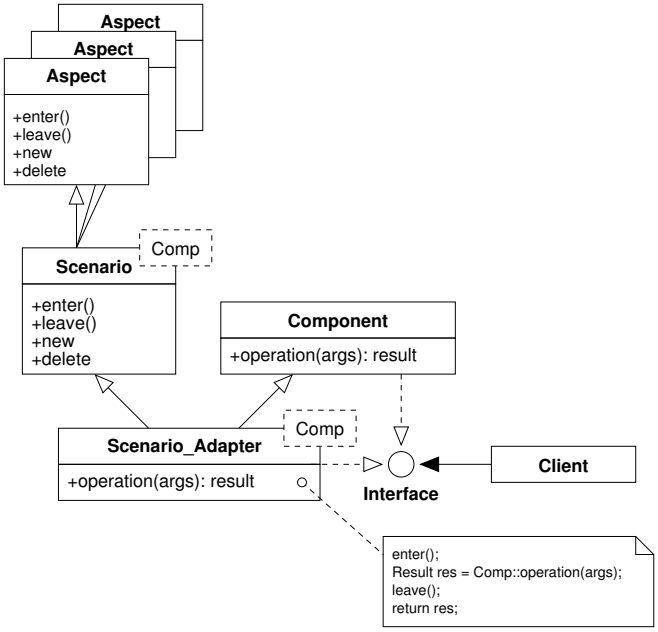

Figure 11. EPOS Scenario Adapter.

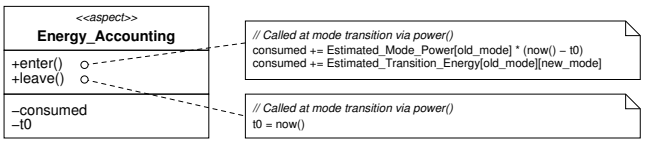

Figure 12. Energy accounting aspect.

definition for `enter` and `leave`, and adjusts their activation for each individual target component[3]. C++ operators `new` and `delete` can also be redefined to induce the invocation of static versions of `enter` and `leave` respectively for the instantiation and destruction of components.

Following AOP principles, **energy accounting** can be implemented as an aspect program that adds event counters to components and adapts the corresponding methods to manipulate them as illustrated in figure 12. When `power()` is invoked on a component, the aspect program checks for mode changes while *entering* the corresponding *scenario*, issuing the accounting directives accordingly. **Auto-resuming** a component that has been put in an energy-saving mode can be accomplished by testing and conditionally restoring the component's power mode on each method invocation as illustrated by figure 13.

Auto-suspend mechanisms can also take advantage of AOP

[3]Each component in the system is characterized by a `Trait` construct that is used by `Scenario` to decide which aspect programs must be applied to the component and in which order.

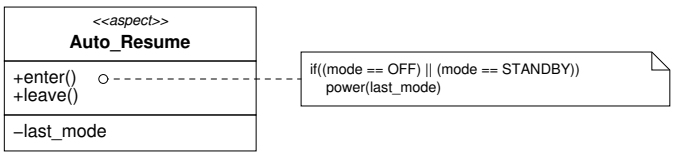

Figure 13. Auto-resume aspect.

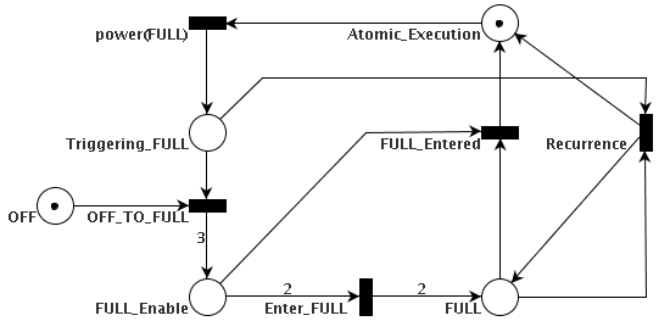

Figure 14. Generalized Operating Mode Transition Network.

techniques. Turning off components that are no longer being used could be easily accomplished by an aspect program that maintains usage counters associated to components. An automated suspend policy could then be implemented in the corresponding `leave` method (that would probably rely on heuristics to decide whether suspension should really take place). Nonetheless, automating power management decisions without taking scheduling concerns into consideration might compromise the correctness of real-time embedded applications. A more consistent strategy would consist in deploying aspect programs of this kind to collect run-time information, while delegating actual power management to an agent integrated with the scheduler.

One pitfall in using AOP techniques to implement a power management infrastructure arises from the fact that individual software components manipulate distinct hardware components in quite specific ways. Implementing the proposed API, so that power mode transitions can be issued at high-level abstractions such as files and processes, would require the envisioned aspect program to consider a complex set of rules. In this proposal, we tackle this problem by formalizing the interaction between components through a set of *Hierarchical Petri Nets* that are automatically transformed in the component-specific rules that are used by our generic aspect programs.

B. Operation Mode Transition Networks

Petri Nets are a convenient tool to model operating mode transitions of components, not only because of its hierarchical representation capability, but also due to the availability of verification tools that can automatically check the whole system for deadlocks and unreachable states [17]. Figure 14 shows a simplified view of the operating mode transition networks used in this proposal (only the transition from OFF to FULL is shown). The complete network encompasses all valid transitions in a similar way, with *places* being associated to operating modes (FULL and OFF in the figure), and *resources* designating the component's current operating mode.

The `Atomic_Execution` place is responsible for ensuring that multiple mode change operations do not take place simultaneously. For that, this place is always initialized with one resource. When a power management API method is invoked, the corresponding transition is triggered

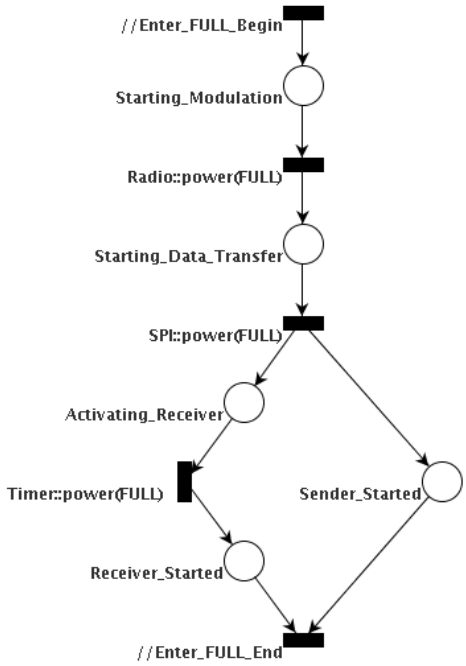

Figure 15. Communicator transition network to enter mode FULL.

(in the figure, `power(FULL)`) and the resource in the `Atomic_Execution` place is consumed. Additionally, a new resource is inserted into the `Triggering_FULL` place to enable the transactions that remove the resources that marks the component's current operating mode (`OFF`). Since the component in the example is in the OFF state, only the `OFF_TO_FULL` transition is enabled. When this transition is triggered, the resource that marked the `OFF` place is consumed, and three resources are inserted into the `FULL_Enable` place. This enables the `Enter_FULL` transition, that is responsible for executing the operations that actually change the component's power mode. After this transition is triggered, two resources are inserted into the `FULL` place, enabling the `FULL_Entered` transition, which finalizes the process, consuming the final resource in the `FULL_Enable` place, and inserting one resource back into the `Atomic_Execution` place. The entire process results in a resource being removed from the `OFF` place and inserted into the `FULL` place. In order to avoid deadlock when a component is requested to switch to its current operating mode (i.e., a component in `FULL` mode is requested to go into `FULL` mode), another transition was added to the model: `Recurrence`. This transition returns the resource removed from the `Atomic_Execution` place in case of recurrence.

The generalized network represents operating mode transitions from a high-level perspective, without modeling the specific actions that must be taken to put a component into a given power mode. Those actions are subsequently modeled by specializations of mode transitions (such as `Enter_FULL` in figure 14) for each individual component. At this refinement level, Petry Net tools can be used to simulate the network, validating it while generating traces that can be directly mapped to rules used in the aspect programs described earlier.

For instance, the communicator in our example propagates a `power(FULL)` directive down to associated hardware components as specified by the transition network shown in figure 15. The simulation of this transition network produces a trace that is automatically converted to the following code:

```
void Communicator::power_full()
{
    _radio.power(Radio::FULL);
    _spi.power(SPI::FULL);
    _timer.power(Timer::FULL);
}
```

Note that each distinct communicator has its own transition network, thus ensuring that an application issuing the power directive does not need to be patched if the radio on the hardware platform changes or even if it is replaced by a wired transceiver. Similar transition networks are used for all modes, including the apparently more complex *stand-by* and *light* modes. The role of transition networks is solely to propagate power management invocations from high-level abstraction down to hardware components in a consistent manner. The implementation of method `power()` for hardware mediators (i.e. device drivers) does not use the traces of Petry Net simulations. They are entirely written by hand, taking in consideration the operating peculiarities of each hardware device.

Furthermore, invocations of `power()` at the level of hardware mediators cannot be simply propagated, since transitions initiated by invocations on different high-level components (possibly by distinct threads) might conflict as they reach the hardware. For instance, a thread could issue a `power(OFF)` on a file that is stored in a flash memory that also stores other files currently in use by other threads. Therefore, each hardware mediator defines its own `power()` method considering the operating modes available in hardware but also considering its peculiarities in respect to higher level access. Common duties, such as serialization and share control are available as generic aspect programs, but the deployment of such programs is carefully decided by the development team.

IV. Autonomous Power Manager

A considerable fraction of the research effort around power management at software-level has been dedicated to design and implement *autonomous power managers* for general-purpose operating systems, such as Windows and Unix. Today battery-operated portable computers, including notebooks, PDAs, and high-end cellphones, can rely on sophisticated management strategies to dynamically control how the available energy budget is spent by distinct application processes. Although not directly applicable to the embedded system realm, those power managers bear concepts that can be promptly reused in this domain.

As a matter of fact, autonomous power managers grab to a periodically activated operating system component (e.g. timer, scheduler, or an specific thread) in order to trigger operation mode changes across components and thus save energy. For instance, a primitive power manager could be implemented by simply modifying the operating system scheduler to put the CPU in standby whenever there are no more tasks to be executed. DVS capabilities of underlying hardware can also be easily exploited by the operating system in order to extend the battery lifetime at the expense of performance, while battery discharge alarms can trigger mode changes for peripheral devices [18]. Nevertheless, these basic guidelines of power management for personal computers must be brought to context before they can be deployed in embedded systems:

- Embedded systems are often engineered around hardware platforms with very limited resources, so the power manager must be designed to be as slim as possible, sometimes taking software engineering to its limits.
- Many embedded systems run real-time tasks, therefore a power manager for this scenario must be designed in such a way that its own execution does not compromise the deadlines of such tasks. Furthermore, the decisions taken by an autonomous power manager must be in accordance with the requirements of such tasks, since the latency of operating mode changes (e.g. waking up a component) may impact their deadlines. For a real-time embedded system, having a power manager that runs unpredictably might be of consequences similar to the infamous garbage collection issues in Java systems [19].
- Embedded systems often pay a higher energy bill for peripheral devices than for the CPU. Therefore, CPU-centric strategies, such as DVS-aware scheduling, must be reviewed to include external devices. Thus an active power manager must keep track of peripheral device usage and apply some heuristics to change their operating mode along the system lifetime. The decision of which devices will have their operating modes changed and when this will occur is mostly based on event counters maintained by the power management infrastructure, either in hardware or in software.
- As a matter of fact, critical real-time systems are almost always designed considering energy sources that are compatible with system demands. Power saving decisions, such as voltage scaling and device hibernation, are also made at design-time and thus are also taken in consideration while defining the energy budget necessary to sustain the system. At first sight, autonomous power management might even seem out of scope for critical systems. Nonetheless, complex, battery-operated, real-time embedded system, such as satellites, autonomous vehicles, and even sensor networks, are often modeled around a set of tasks that include both, critical and non-critical tasks. A power manager for one such embedded system must respect design-time decisions for critical parts while trying to optimize energy consumption by non-critical parts.

With these premises in mind, the next section briefly surveys the current scenario for power management in embedded systems.

A. Current Power Managers

Just like APIs and infrastructures, most of the currently available embedded system power managers focus on features exported by the underlying hardware. μCLinux captures

APM, ACPI or equivalent events to conduct mode transitions for the CPU and also for devices whose drivers explicitly registered to the power manager [20].

In TINYOS, OS-driven power management is implemented by the task scheduler, which makes use of the `StdControl` interface to start and stop components [21]. When the scheduler queue is empty, the main processor is put in *sleep* mode. In this way, new tasks will only be enqueued during the execution of an interrupt handler. This method yields good results for the main microcontroller, but leaves more aggressive methods, including starting and stopping peripheral components up to the application. When compared to μCLINUX, TINYOS delivers a lighter mechanism, more adequate to embedded systems, yet suffers from the same limitations with regard to usability and portability.

MANTIS uses an *idle* thread as entry point for the system's power management policies, which put the processor in *sleep* mode whenever there are no threads waiting to be executed [22].

GRACE-OS is an energy-efficient operating system for mobile multimedia applications implemented on top of LINUX [23]. The system combines real-time scheduling and DVS techniques to dynamically control energy consumption. The scheduler configures the CPU speed for each task based on a probabilistic estimation of how many cycles they will need to complete their computations. Since the systems is targeted at soft real-time multimedia applications, loosing deadlines due to estimation errors is tolerated. GRUB-PA follows the same guidelines, but addresses hard real-time requirements more consistently by imposing DVS configuration restrictions for this kind of task [24].

Niu also proposes an strategy to minimize energy consumption in soft real-time systems through adjusts in the system QoS level [25]. In this proposal, tasks specify CPU QoS requirements through `(m,k)` pairs. These pairs are interpreted by the scheduler as execution constraints, so that a task must meet at least `m` deadlines for any `k` consecutive releases. The possibility to lose some deadlines enables the scheduler to explore DVS more efficiently at the cost of preventing its adoption in many (hard real-time) embedded systems.

Yet in the line of energy savings through adaptive scheduling and QoS, ODYSSEY takes the concept of soft real-time to the limit. The system periodically monitors energy consumption by applications in order to adjust the level of QoS. Whenever energy consumption is too high, the system decreases QoS by selecting lower performance and power consumption modes. In this way, system designers are able to specify a minimum lifetime for the system, which might be achieved by severely degrading performance [26].

ECOS defines a currency, called *currentcy*, that applications use to *to pay for* system resources [27]. The system distributes *currentcies* to tasks periodically accordingly to an equation that tracks the battery discharge rate as to ensure a minimum lifetime for the system. Applications are thus forced to adapt their execution pace according to their *currentcy* balances. This strategy has one major advantage over others discussed so far in this paper: the *currentcy* concept encompasses not only the energy spent by the CPU (to adjust DVS configuration), but the energy spent by the system as a whole, including all peripheral devices.

Harada explores the trade-off between QoS maximization and energy consumption minimization by allocating processor cycles and defining operating frequencies with QoS guarantees for two classes of tasks: real-time (mandatory) and best-effort (optional) [28]. The division of tasks in two parts, one *mandatory*, that must always be executed, and another *optional* that is only executed after ensuring that there are enough resources to execute the mandatory parts of all tasks is the basic premise behind *Imprecise Computation* [29], which is also one of the foundations of the power manager proposed in this work.

B. Proposed Power Manager

From the above discussion about currently available power managers for embedded system, one can conclude that no single manager consistently addresses all the points identified earlier in this section: leanness, real-time conformance, peripheral device control, and design-time decision awareness. We follow these premises and build on the API proposed in section II and on the infrastructure presented in section III to propose an effective autonomous power manager for real-time embedded systems.

For the envisioned scenarios of battery-operated, real-time, embedded systems, energy budgets would be defined at design-time based on critical tasks, while non-critical tasks would be executed on a best-effort policy, considering not only the availability of time, but also of energy. Along with the assumption that an autonomous power manager cannot interfere with the execution of hard real-time tasks (i.e., cannot compromise their deadlines), the separation of critical and non-critical tasks at design-time lead us to the following scheduling strategy:

- Hard real-time tasks are handled by the system as mandatory tasks, executed independently of the energy available at the moment. These tasks are scheduled according to traditional algorithms such as Earliest Deadline First (EDF) and Rate Monotonic (RM) [30], either in their original shape or extended to support DVS.
- Best-effort tasks, periodic or not, are assigned lower priorities than hard real-time ones and thus are only executed if no hard real-time tasks are ready to run. Furthermore, the decision to dispatch a best-effort task must also take in consideration whether the remaining energy will be enough to schedule all hard-real time tasks.
- Whenever a best-effort task is prevented from executing due to energy limitations, a speculative power manager is activated in order to try to change components, including peripheral devices, to less energy-demanding operating modes, thus promoting energy savings.

With this strategy, the autonomous power manager will only be executed if energy consumption is detected excessive (i.e. a best-effort task has been denied execution) and time is available (i.e. a best-effort task would be executed). Non-interference between power manager and hard real-time tasks is ensured, in terms of scheduling, by having the power

manager to run in preemptive mode, so that a hard real-time task would interrupt its execution as soon as it gets ready to run (e.g. after waiting for the next cycle).

This scheduling strategy has only small implications in terms of process management at the operating system level, but require a comprehensive power management infrastructure, like the one presented in section III, in order to be implemented. In particular, battery monitoring services are needed to support the scheduling decisions around best-effort tasks and component dependency maps are needed to avoid power management decisions that could impact the execution of hard real-time tasks.

The battery monitoring service provided by the PM infrastructure can be combined with the energy accounting service to reduce the costs of gauging the amount of energy still available to the system. With updated statistics from the energy accounting infrastructure in hand, the scheduler can predict battery discharge without having to physically interact with it, thus sparing the corresponding energy. In this way, battery monitoring is programmed to take place sporadically based on the lifetime specified for the system. An additional trigger is bound to the prediction counter kept by the scheduler, so monitoring also takes place when power consumption reaches specified thresholds.

The operating mode transition networks introduced in section III as means to control the propagation of power management actions from high-level components down to the hardware can be used by the autonomous power manager to keep track of dependencies among components. Along with a list of currently active components maintained by the operating system, these transition networks build the basis on which peripheral control can be done by the power manager. For instance, if a task has an open file that is no longer being used, the power manager could track that component down to a flash memory and change its operating mode to standby or off.

Nevertheless, the compromise with real-time systems requires our power manager to take API calls made by hard real-time tasks as "orders" instead of "hints". We assume that, if a hard real-time task calls the `power()` API method on a component to set its operating mode to *full*, then that component must be kept in that mode even if the collected statistics indicate that it is no longer being used and thus would be a good candidate to be shutdown. Otherwise, the corresponding task could miss its deadline due to the delay in reactivating that component.

V. Implementation in EPOS: a Case Study

In order to validate the power management strategy for embedded systems proposed in this paper, which includes an API specification, guidelines for power management infrastructure implementation through aspect-programs, and design constraints for the development of autonomous power management agents, these mechanisms have been implemented in EPOS along with the hypothetical remote monitoring application described in section II.

A. EPOS *Overview*

EPOS, the Embedded Parallel Operating System, aims at building tailor-made execution platforms for specific applications [31]. It follows the principles of *Application-driven Embedded System Design* [16] to engineer families of software and hardware components that can be automatically selected, configured, adapted, and arranged in a component framework according with the requirements of particular applications.

An application written based on EPOS published interfaces can be submitted to a tool that performs source code analysis to identify which components are needed to support the application and how these components are being deployed, thus building an execution scenario for the application. Alternatively, users can specify execution scenarios by hand or also review an automatically generated one. A build-up database, with component descriptions, dependencies, and composition rules, is subsequently accessed by the tool to proceed component selection and configuration, as well as software/hardware partitioning based on the availability of chosen components in each domain. If multiple components match the selection criteria, then a cost model is used, along with user specifications for non-functional properties, such as performance and energy consumption, to choose one of them[4].

After being chosen and configured, software components can still undergo application-specific adaptations while being plugged into a statically metaprogrammed framework that is subsequently compiled to yield a run-time support system. This application-specific system can assume several shapes, from simple libraries to operating system micro-kernels. On the hardware side, component selection and configuration yields an architecture description that can be either realized by discrete components (e.g. microcontrollers) or submitted to external tools for IP synthesis. An overview of the whole process can be seen in figure 16.

B. *Example Application*

The remote sensing application described in section II was implemented in EPOS as excerpted in figure 10. When submitted to EPOS tools, the remote sensing program yielded a run-time library that realizes the required interfaces and a hardware description that could be matched by virtually any hardware platform in the system build-up database. We forced the selection of a well-known platform, the Mica2 sensor node [5] by manually binding `Communicator` to the CC1000 radio on the Mica2 platform and `Actuator` to a led.

In the experiment, energy for the system was delivered by two high-performance alkaline AA batteries with a total capacity of 58320 J (5400 mAh at 3 V), in excess of table I estimates of what would be necessary to match the intended life-time of one year (3576 mAh at 3 V). The system was configured with a scheduling quantum of 15 ms and a battery monitoring period of one day. Energy accounting was enabled and produced statistics that were used by the scheduler on every thread dispatching.

[4]Design-space exploration is currently being pursued in EPOS by making the cost model used by the building tool dynamic.

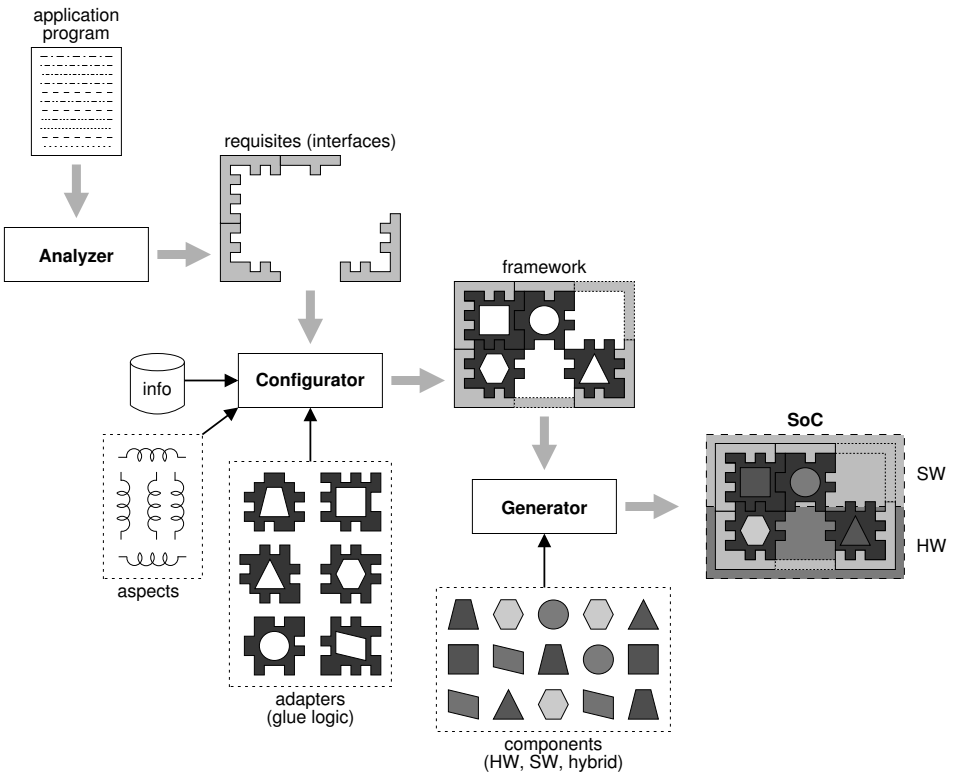

Figure 16. Overview of tools involved in EPOS automatic generation.

The experiment was profiled during approximately one week using a digital oscilloscope. From the collected data, we determined the average energy consumption per hour to be of 5.07 J. We then extrapolated the total energy consumption for one year to be of 4112 mAh. This extrapolation projects a system lifetime of 479 days, confirming that the system will match the expectations in this respect. This experiment also shows that the energy overhead caused by the implemented power management mechanisms is largely compensated by the power it saves (as calculated in section II, running the example application without any power management would demand 40 times more energy than the predicted 3576 mAh). It is also important to notice that the additional 536 mAh cannot be entirely accounted to power management. A fraction of it arises from the additional circuitry needed to couple the key components considered at design-time, another fraction from the non-linear discharge nature of the chosen batteries, and yet another can be accounted to misleading estimates published by manufacturers.

This experiment also allowed us to assess the strategy overhead in terms of memory and CPU utilization. Table III shows the memory increase caused by the proposed mechanisms. The reference system was stripped of any PM capabilities and than enriched with the PM API, power accounting, and finally the autonomous manager integrated into the scheduler. The considerable increase in size for every step is justified by the fact that they affect all components in the system. The PM API required versions of mediators that are able to control the power mode of associated devices plus a global `System` object and access control to handle event propagation

EPOS Configuration	Code	Data	Total
No PM	19086	588	19674
With PM API	29146	721	29867
+ Accounting	31914	791	32705
+ Autonomous PM	45262	816	46078

Table III
POWER MANAGEMENT MEMORY OVERHEAD (SIZES IN BYTES).

conflicts as described in section III-B. Accounting enriched components with counters and the associated maintenance code. The autonomous manager required battery monitoring plus statistics handling and decision making support.

In respect to performance, the proposed mechanisms only substantially affect hardware mediators and the scheduler. Other system components, although adorned with `power()` methods and event counters, do not have their original behavior altered by the aspect programs responsible for power management and therefore show no performance loss[5]. In order to precisely assess the overhead caused by the proposed power management mechanisms, we inserted simple primitives to switch a led on and off around target methods and obtained the average active period with an oscilloscope. Table IV shows the increase in execution time for context switch and I/O operations. The calculations performed by the scheduler to keep statistics and decide whether a best-effort task can be dispatched extended context switch time by 9μs (the target platform features an 8-bit AVR running at 8MHz). The incre-

[5]Some event counters are initialized at object construction time through modified versions of operator `new`, but this small overhead is usually restricted to start up phases.

EPOS Configuration	Context Switch	I/O Path
No PM	33	380
With Autonomouns PM	42	440

Table IV
POWER MANAGEMENT PERFORMANCE OVERHEAD (TIMES IN μS).

ment in the path to I/O devices caused by auto-resume and accounting was measured to be 60μs and reveals weakness of the target architecture to handle the associated arithmetics. Platforms capable of keeping event counters associated to I/O ports in hardware could eliminate a reasonable fraction of this overhead.

C. Autonomous Power Manager

The example application discussed along this article has been conceived to support the explanation of the proposed power management strategy. Its implementation described in the previous section also allowed us to confirm most of the claimed benefits. Nonetheless, it does not feature a best-effort task that could corroborate the proposed autonomous power manager design. Therefore, we extended it with two additional best-effort threads on a second experiment: thread `Calibrator` periodically calibrates the temperature sensor, and thread `Display` shows the current temperature on a led display. Both threads make use of hardware components available in the original platform, but with a significant difference: `Calibrator` uses the thermometer, which it shares with `Monitor` and `Recovery`, while `Display` uses dedicated leds. This distinction is important to drive the power manager through a situation in which the decision about suspending resources (ADC in the thermometer) used by a frozen best-effort task (`Calibrator`) must consider hard real-time tasks demands (`Monitor` and `Recovery`). The threads were created with periods of 100 seconds and 100 ms, respectively.

Just like the example application, this second experiment was profiled during approximately one week using a digital oscilloscope and a new battery set. The results of the experiment are summarized in table V, which presents the system average energy consumption for five different setups: (a) executing without the additional threads; (b) executing the `Calibrator` thread with hard real-time priority and `Display` as best-effort; (c) the reverse situation, `Display` as hard real-time and `Calibrator` as best-effort; (d) with both threads running with hard-real time priority; and (e) with both threads running in best-effort priority.

Setup (a) is equivalent to the example application evaluated in the previous section and produced equivalent results. Setup (b) is impacted by the periodic ADC operations performed by `Calibrator` (100 samplings every 100 seconds). The best-effort thread `Display` has virtually no chance to execute in this setup, since the accounting system quickly feeds the scheduler with information indicating that the desired lifetime of 365 days cannot be matched under the observed energy consumption rate. Setup (c) features a similar situation, but with switched roles. The led display used has a high tool on energy, so the system reached a lifetime of only 84 days. In this setup, `Calibrator` is the thread that is prevented from executing, however, the ADC it uses (via the thermometer abstraction) is not implicitly put to sleep, since it is also used by the `Monitor` and `Recovery` hard real-time threads.

For setup (d), both threads have been configured as hard real-time, so they are always executed. This reduces the system's lifetime to about 76 days. Setup (e) is the one that best characterizes the proposed autonomous power manager. Running the additional threads in best-effort priority enables the scheduler to suppress their execution whenever the energy budget needed to achieve the specified lifetime is threatened. This smoothly drives the system toward the desired lifetime, enabling both threads to run just sporadically when the battery monitor indicates that there is enough energy.

VI. CONCLUSION

In this article, power management in embedded systems was addressed from energy-aware design to energy-efficient implementation, aiming at introducing a set of mechanisms specifically conceived for this scenario. A power management API defined at the level of user-visible system components was proposed and compared with traditional APIs. Its implementation was discussed in the context of the necessary infrastructure, including battery monitoring, accounting, auto-suspend, and auto-resume. An energy-event propagation mechanism based on *Petry Nets* was proposed and its implementation using *Aspect-Oriented Programming* techniques was depicted. The use of the proposed infrastructure by an autonomous power manager integrated into a real-time scheduler was also discussed, thus covering the main components of a modern power management system for embedded systems.

The proposed strategy was illustrated and evaluated through a didactic, yet realistic, example embedded system targeted at environment temperature monitoring. The example was described from early design stages down to a real implementation for the EPOS system on the Mica2 Mote, a well-known platform that helps to put the proposal into perspective. Experiments with this implementation showed that integrating the proposed power management mechanisms into a hard-real time run-time support system comes at a high cost in terms of program memory, specially in platforms with limited hardware support. Nonetheless, they also showed a relatively small impact on performance, slightly adding to the latency of scheduling and I/O operations. This can be explained mainly by the sporadic nature of power management operations.

The experiments carried out also made evident the benefits of the proposed mechanisms in terms of energy efficiency and system utility as they confirmed the strategy's ability to sustain a given lifetime for the system without affecting the deadlines of hard real-time tasks at the same time it enables the safe usage of the remaining energy by best-effort tasks. These benefits arise from proposed strategy itself and are not dependent from EPOS or Mica2. Therefore, the intended contribution for this article is not "yet another power manager for embedded systems", but the introduction of a broader and systematic way to deal with power management issues in embedded systems.

	Setup		1-Hour	365-Day Con-	Lifetime
	Calibrator	Display	Energy (J)	sumption (mAh)	(days)
(a)	-	-	5.07	4112	479
(b)	RT	BE	8.13	6597	299
(c)	BE	RT	28.83	23384	84
(d)	RT	RT	31.89	25869	76
(e)	BE	BE	6.66	5402	365

Table V
ENERGY CONSUMPTION UNDER DIFFERENT AUTONOMOUS POWER MANAGER SETUPS (RT = HARD REAL-TIME / BE = BEST-EFFORT).

ACKNOWLEDGMENT

I would like to thank and acknowledge former LISHA members Arliones S. Hoeller Jr, Geovani R. Wiedenhoft, Giovani Gracioli and Lucas Wanner for implementing many of the concepts and ideas presented in this article.

REFERENCES

[1] J. Monteiro, S. Devadas, P. Ashar, and A. Mauskar, "Scheduling techniques to enable power management," in *Proceedings of the 33rd Annual Design Automation Conference*, Las Vegas, USA, Jun 1996, pp. 349–352.

[2] L. Benini, A. Bogliolo, and G. D. Micheli, "Dynamic power management of electronic systems," in *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. New York, USA: ACM Press, 1998, pp. 696–702.

[3] P. H. Chou, J. Liu, D. Li, and N. Bagherzadeh, "Impacct: Methodology and tools for power-aware embedded systems," *DESIGN AUTOMATION FOR EMBEDDED SYSTEMS, Special Issue on Design Methodologies and Tools for Real-Time Embedded Systems*, vol. 7, no. 3, pp. 205–232, Oct 2002.

[4] Panasonic, *ERTJ Multilayer Chip NTC Thermistors Datasheet*, Panasonic, 2004.

[5] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy, "The platforms enabling wireless sensor networks," *Communications of the ACM*, vol. 47, no. 6, pp. 41–46, 2004.

[6] D. Tennenhouse, "Proactive Computing," *Communications of the ACM*, vol. 43, no. 5, pp. 43–50, May 2000.

[7] G. R. Wiedenhoft, A. S. H. Junior, and A. A. Fröhlich, "A Power Manager for Deeply Embedded Systems," in *12th IEEE International Conference on Emerging Technologies and Factory Automation*, Patras, Greece, Sep. 2007, pp. 748–751. [Online]. Available: http://www.lisha.ufsc.br/~guto/publications/etfa2007-wiedenhoft.pdf

[8] S. Mandal, R. Mahapatra, P. Bhojwani, and S. Mohanty, "Intellbatt: Toward a smarter battery," *Computer*, vol. 43, no. 3, pp. 67–71, mar 2010. [Online]. Available: http://dx.doi.org/10.1109/MC.2010.72

[9] S. Kellner and F. Bellosa, "Energy accounting support in tinyos," in *GI/ITG KuVS Fachgespräch Systemsoftware und Energiebewusste Systeme*, no. 2007-20. Karlsruhe, Germany: Fakultät für Informatik, Universität Karlsruhe (TH), Oct. 2007, pp. 17–20, interner Bericht. [Online]. Available: http://i30www.ira.uka.de/research/publications/pm/

[10] Z. Ren, B. Krogh, and R. Marculescu, "Hierarchical adaptive dynamic power management," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 409–420, 2005.

[11] N. Aboughazaleh, D. Mossé, B. R. Childers, and R. Melhem, "Collaborative operating system and compiler power management for real-time applications," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 1, pp. 82–115, feb 2006.

[12] S.-Y. Bang, K. Bang, S. Yoon, and E.-Y. Chung, "Run-time adaptive workload estimation for dynamic voltage scaling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 9, pp. 1334–1347, sep 2009. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2009.2024706

[13] G. Dhiman and T. Rosing, "System-level power management using online learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 676–689, may 2009. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2009.2015740

[14] N. Pettis and Y.-H. Lu, "A homogeneous architecture for power policy integration in operating systems," *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 945–955, jul 2009. [Online]. Available: http://dx.doi.org/10.1109/TC.2008.180

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the European Conference on Object-oriented Programming'97*, ser. Lecture Notes in Computer Science, vol. 1241. Jyväskylä, Finland: Springer, Jun. 1997, pp. 220–242. [Online]. Available: http://www.parc.xerox.com/csl/groups/sda/publications/papers/Kiczales-ECOOP97/for-web.pdf

[16] A. A. Fröhlich, *Application-Oriented Operating Systems*. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001. [Online]. Available: http://www.lisha.ufsc.br/~guto/publications/aoos.pdf

[17] J. L. Peterson, "Petri nets," *ACM Comput. Surv.*, vol. 9, no. 3, pp. 223–252, 1977.

[18] V. Devadas and H. Aydin, "On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications," in *Proceedings of EMSOFT 08*, Atlanta, USA, oct 2008, pp. 99–108.

[19] D. F. Bacon, P. Cheng, and V. T. Rajan, "A real-time garbage collector with low overhead and consistent utilization," *ACM SIGPLAN Notices*, vol. 38, no. 1, pp. 285–298, 2003.

[20] S. Vaddagiri, A. K. Santhanam, V. Sukthankar, and M. Iyer, "Power management in linux-based systems," *Linux Journal*, 2004.

[21] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, USA, 2000, pp. 93–104.

[22] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms," *Mob. Netw. Appl.*, vol. 10, no. 4, pp. 563–579, 2005.

[23] W. Yuan, "Grace-os: An energy-efficient mobile multimedia operating system," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2004.

[24] C. Scordino and G. Lipari, "Using resource reservation techniques for power-aware scheduling," in *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*. New York, USA: ACM Press, 2004, pp. 16–25.

[25] L. Niu and G. Quan, "A hybrid static/dynamic dvs scheduling for real-time systems with (m, k)-guarantee," *rtss*, vol. 0, pp. 356–365, 2005.

[26] J. Flinn and M. Satyanarayanan, "Managing battery lifetime with energy-aware adaptation," *ACM Trans. Comput. Syst.*, vol. 22, no. 2, pp. 137–179, 2004.

[27] H. Zeng, C. S. Ellis, and A. R. Lebeck, "Experiences in managing energy with ecosystem," *IEEE Pervasive Computing*, vol. 4, no. 1, pp. 62–68, 2005.

[28] F. Harada, T. Ushio, and Y. Nakamoto, "Power-aware resource allocation with fair qos guarantee," in *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Washington, USA: IEEE Computer Society, 2006, pp. 287–293.

[29] J. W. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, "Imprecise computations," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 83–94, Jan 1994.

[30] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[31] A. A. Fröhlich and L. Wanner, "Operating system support for wireless sensor networks," *Journal of Computer Science*, vol. 4, no. 4, pp. 272–281, 2008.