Towards Modular Resource-Aware Applications

Somayeh Malakuti, Steven te Brinke, Lodewijk Bergmans, and Christoph Bockisch

University of Twente – Software Engineering group – Enschede, The Netherlands

{malakutis, brinkes, bergmans, c.m.bockisch}@ewi.utwente.nl

Abstract

Resource optimization is an increasingly important requirement in the design and implementation of software systems. It is applied to improve both environmental sustainability and usability of resource-constrained devices. This paper claims that to achieve more modular resource-aware applications, the resource utilization of components must explicitly be modeled. Due to shortcomings of existing modeling languages, we propose a notation for the resource consumption of components and we illustrate the suitability of this notation by means of two real-world examples. We observe that explicitly modeling resource consumption has as result that resource consumption information is scattered across and tangled with the functional services of components.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques—Modules and interfaces

General Terms Design, Performance

Keywords resource-utilization model, resource-consumption optimization, software composition, aspect-oriented programming

1. Introduction

The rise of resource-constrained devices like cell phones, but also the increasing awareness of the need for environmental sustainability, makes optimization of resource consumption an evermore important requirement [4]. Software systems are composed of various different kinds of components organized in different layers. Typical examples are hardware-related software components, middleware components and application components. It has been demonstrated repeatedly that optimization techniques, implemented in software, can lead to substantial reduction of resource usage [3, 7], within both the computer system and the system being controlled. Optimization can take place for components residing at each layer [3, 8, 10] and it can be performed across layers.

Resource optimization can be carried out statically before the actual execution of software; this is usually achieved by modeling the resource consumption at the architectural level and performing analyses on the models. However, due to the complexity of today's software and its execution environment, inevitably optimization must also be carried out during the actual execution of the software. This leads to self-adaptive software that adjusts its behavior based on the changes in resource availability [2].

Our research focuses on industrial embedded software and we are interested in addressing resource-optimization during runtime. As the first step in designing resource-aware self-adaptive applications, we claim that the resource utilization of components must explicitly be modeled to support the following tasks:

- Components utilizing a resource of interest must be identified. As a consequence, they must be target to resource optimization at runtime.
- Relations between the functional services of components and their resource consumption must be specified. This pinpoints the services that must be target to resource optimization at runtime.
- Components that are jointly utilizing a resource and their interaction among each other must be specified.

The resource utilization of a component also affects other components that directly or indirectly use the same resource, e.g., through the component's services. Thus, to enable modular resource optimization, the resource utilization of a component and the effect of a component's service on resource consumption must be explicit in the component interface.

Existing modeling languages such as UML do not offer generic support for modeling the resource utilization of components. However, dedicated UML Profiles [5] describe *specific* sets of common software and hardware resources.

Research on explicitly modeling resource utilization exists [1, 9], but these approaches do not take software modularity into consideration. Also, some middleware is aware of *quality of service* (QoS) [6] and resource consumption can also be considered a QoS. However, the optimization of

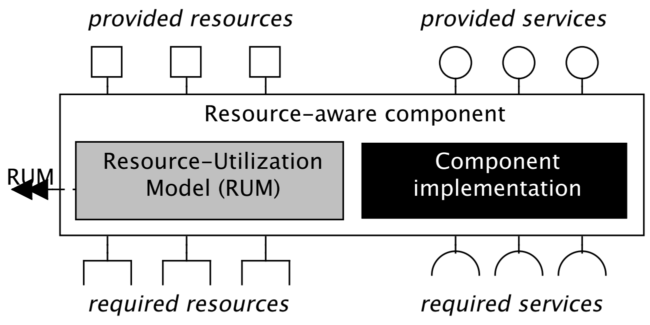

Figure 1. Notation for Resource-Aware Components

QoS in middleware assumes that all optimizations are handled through a single middleware layer.

The contributions of this paper are as follows:

- We propose a notation for resource-aware applications.
- We illustrate the suitability of this notation for a modular design of resource-aware applications and resource optimization by means of two real-world case studies.
- We outline generic realization guidelines for resource-aware applications.

2. A Notation for Modeling Resource-Aware Applications

Traditionally, a component is considered as a unit of development and deployment, with explicit interfaces specifying the services that it provides to and the services that it requires from its environment [11, chapter 5]. We extend components with explicit interfaces specifying the resources that it provides to and requires from its environment. Besides its implementation, each component encapsulates a so-called *resource-utilization model* which expresses the relation between the component services and resources.

Figure 1 represents our notation of components. In the following, we explain each part in more detail (starting at the top-left and continuing clock-wise):

provided resources This is a separate description for all resources a component provides, including which type of resources, and—as appropriate—static constraints on the availability of these resources.

provided services The functional behavior of a component is specified as separately described services. The key issue here is that typically, each service will consume resources, and the component specification describes which resources these are, with—as appropriate—static constraints on the consumption of these resources. The resource consumption of an individual service invocation may be influenced by its parameter values.

required services Similarly, a component may require certain services from other components, to fulfill its duties. The required services may also specify constraints with respect to resource usage that provided services must adhere to (such as a maximum amount of consumed power).

required resources A component may specify that it requires a certain amount of resources; for example, because the component encapsulates (hardware) behavior that requires certain resources, or because the component will provide these resources—often with some restrictions or policy—to other components.

resource-utilization model (RUM) Finally, resource-aware components may declare their resource behavior, i.e., the dynamic relation between resources and services. For example, they may specify that using a certain service of a component increases the availability of a provided resource or puts a component in a state where it consumes more of a required resource. The RUM specifies the impact on resources in detail, e.g.: the degree to which the availability or consumption of the resource changes, the end-condition for staying at this level, the availability of resources and services in different situations, and so on.

Choosing a suitable notation for representing resource-utilization models is a challenge. As shown in section 3, we consider state machines a suitable notation, because (a) they are declarative, (b) they can model the internal behavior of a component and its relation with the services that the component provides to and requires from its environment, and (c) they can conveniently be extended with resource-utilization annotations on states as well as state transitions. Such information can be used by resource optimizers to make more adequate decisions, based on how various components provide and consume resources.

With respect to optimizing the resource consumption of a system, components can have one or more of the responsibilities: providing, consuming, or controlling resources. Pure providers are, for example, drivers that encapsulate hardware resources such as power, memory, or bandwidth. Pure consumers are, for example, application components that consume one or more resources. Controller components can be optimizers that require resources from the resource providers and provide them to consumers after applying certain resource-optimization algorithms. We claim that the notation represented in figure 1 can express these three kinds of components. For example, a resource provider does not have any resource consumption interface, an application component does not have any resource provision interface, and a controller component has both kinds of interfaces.

3. Case Studies

In this section we use the notation presented in section 2 for modeling two real-world resource-aware applications.

3.1 The Hiker's Buddy Application

The Hiker's Buddy application [3] receives information sent by a Global Positioning System (GPS) receiver and plots the hiker's current location on a topographical map that is retrieved by searching a database of map segments. This application is necessarily mobile and battery-powered.

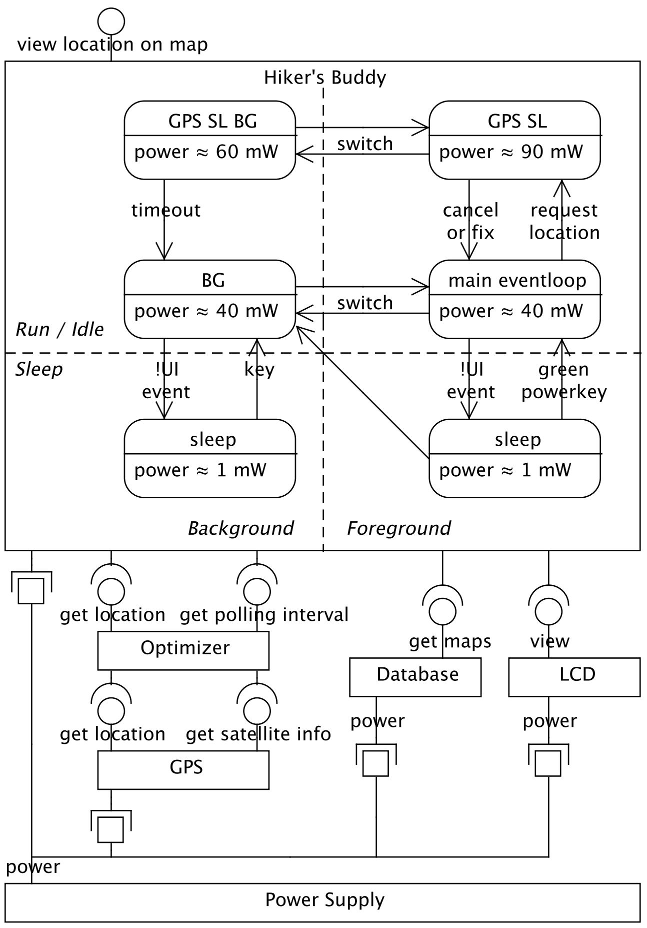

Figure 2. A component diagram for Hiker's Buddy

Architecture The main components of Hiker's Buddy are depicted in figure 2. The application *Hiker's Buddy* uses an *LCD* to show maps to the user. These maps are loaded from a *Database*. In order to know the current position of the user, a *GPS* is used. The *GPS* component encapsulates both acquiring satellite information and providing this information to the system—which is done over a serial line. The *Optimizer* can be used to optimize the *GPS* utilization, which will be explained in the next paragraphs. The *Power Supply* delivers battery power to the different components.

Resource optimization Polling the GPS for location information is a costly power state. Its total energy expenditure depends on both power consumption of the state and the time spent in this state. These are influenced by the polling frequency of GPS data: The faster polling occurs, the faster Hiker's Buddy can discover the availability of a good GPS fix, thus reducing the time spent in a costly power state; but polling less frequently allows the processor to spend more time in idle mode, thus lowering the overall power cost of this loop. If that does not significantly affect the total time for the loop, energy use is reduced. An adaptive polling frequency is usually a better policy for energy use than any fixed polling interval, e.g., by increasing the polling frequency when a GPS fix is getting close.

Resource-utilization models A high-level model of the power states for Hiker's Buddy—which is adopted from Ellis [3]—is shown inside the *Hiker's Buddy* component in figure 2. The states are divided in two parts by the vertical dashed line: (a) The states to the left of the line capture the device requirements remaining when another application is in the foreground; (b) the states to the right of the line represent Hiker's Buddy as the foreground application. Every state is annotated with the approximate power consumed by the device while it is in this state. The state *GPS SL* is actively reading data from the GPS while Hiker's Buddy is in the foreground; it is the most power consuming state. The *main eventloop* polls the GPS for the current location by switching to the *GPS SL* state at regular intervals.

The *GPS* component does not only offer the current location, but also information about the acquired satellites. Using this information, *Optimizer* can estimate how close a GPS fix is and—together with the information provided by the RUM of *Hiker's Buddy*—optimize the polling interval.

The power utilization shown in this example was acquired by Ellis, measuring the power consumption of the device as a whole. Thus, we can only present a RUM for the Hiker's Buddy component. While this is sufficient to perform the desired optimization, it hinders separate maintainability of the different components. If, for example, we want to replace the *Database* or *GPS* component, we cannot see how this would influence the resource-utilization model. In order to do so, the whole RUM must be updated. Therefore, we claim that every component should have its own RUM.

3.2 Smart Phone Network Traffic Reduction

Mobile devices consume power while transmitting data over the 3G radio network. In this section, we use our notation to model the solution proposed by Qian et al. [7] for optimizing the power usage of these devices.

Architecture The high-level architecture of the smart phone device shown in figure 3 contains four components. The component *Application* represents mobile applications that require services from the 3G network. In our example, we focus on one application only, i.e., a media-player application, but in practice multiple applications could be executing simultaneously. The media-player mobile application provides three services: *play*, *pause*, and *stop*. The component requires the services *connect*, *disconnect*, *download*, and the resource *connection*. The component *Optimizer* improves the usage of connections by the applications, which consequently leads to the optimization of radio power usage.

The component *Network Manager*—offered by the 3G network—provides the actual data transfer over the network, for which it requires the resource *radio power* provided by the component *Power Supply*.

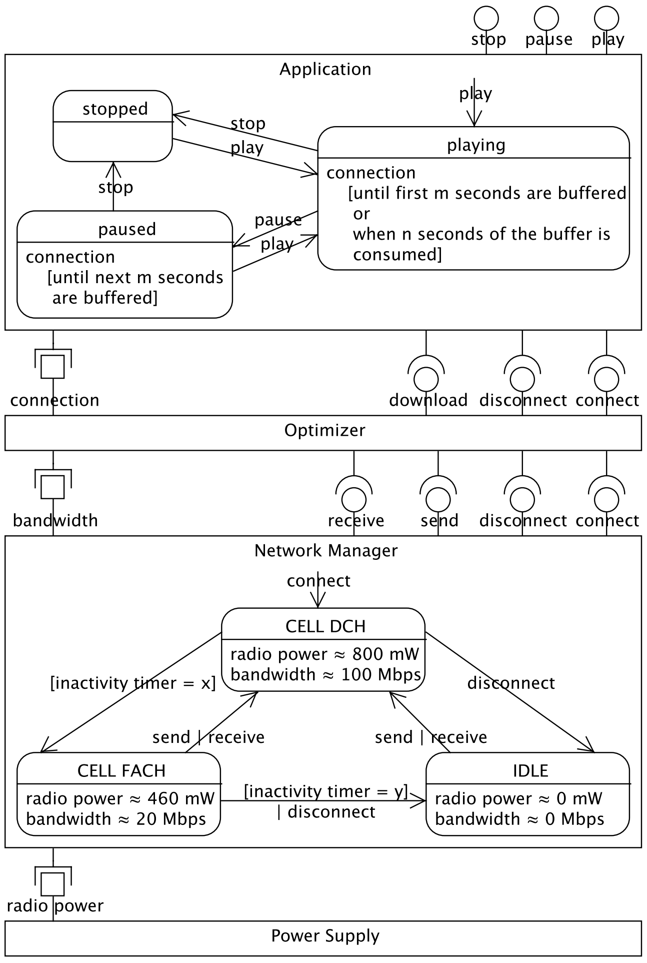

Figure 3. A component diagram for smart phone network performance

Resource optimization In cellular networks, the connection of phones to the 3G network consumes radio power. To efficiently utilize this limited power, the component *Network Manager* introduces a so-called inactivity timer for each phone and degrades or releases connections if a phone is inactive for the specified amount of time. Although this reduces the radio-power consumption and bandwidth usage, during the inactivity time, radio power is still being consumed to some extent. Therefore, power consumption is further improved by introducing the component *Optimizer* as mediator between *Application* and *Network Manager*. It explicitly commands *Network Manager* to release connections when they are no longer needed by *Application*.

Resource-utilization models The internal resource consumption of the component *Application* is depicted in figure 3 as a state machine. Here, the state *playing* is the start state. To buffer media content, in this state the resource *connection* is consumed by invoking the service *connect* followed by *download*. When the first *m* seconds of media content are buffered, the connection is released by invoking the service *disconnect*. The connection is again established after *n* seconds of the buffered content are played.

The invocation of the service *stop* causes a transition to the state *stopped* that does not need the resource *connection* anymore. Therefore, it invokes the service *disconnect*. The invocation of the service *pause* causes a transition to the state *paused* if the state chart is in the state *playing*. In the state *paused* the resource *connection* is required until *m* seconds of media content are buffered.

The component *Optimizer* uses the algorithm proposed by Qian et al. to manage the connections for the applications. In short, when an application invokes the service *disconnect*, it also identifies the next time when it requires a connection. If all of the applications require the connection no sooner than *x* seconds (*x* is defined by the RUM of *Network Manager*), *Optimizer* invokes the service *disconnect* on the component *Network Manager*; otherwise, it keeps consuming resources. In this way, it keeps a balance between download latency and resource consumption.

The component *Network Manager* keeps a state machine for each phone to manage its bandwidth consumption. The available bandwidth and the power consumption are different per state. Here, the state *IDLE* is the default state, indicating that the phone has not established a connection, thus no bandwidth is allocated and no radio power is consumed. The state *CELL DCH* indicates that the phone is allocated dedicated transport channels in both downlink and uplink. The state *CELL FACH* indicates that a connection is established but there is no dedicated transport channel allocated to a phone. Instead, the phone can only transmit user data through shared low-speed channels. Consequently, this state consumes less radio power than the state *CELL DCH*.

To manage the connectivity and, thus, the resource consumption, *Network Manager* defines inactivity times and switches between states. For example, as shown in figure 3, after *x* seconds of idle time a state transition occurs from *CELL FACH* to *CELL DCH*, and after *y* seconds of idle time a transition occurs to *IDLE*.

4. Modularity Requirements for Resource-Aware Software

Our goal is to facilitate the structured development and maintenance of software for resource-aware systems. A key technique for achieving this is to separate concerns and implement them in separate modules. Our proposed notation has been designed to support this. We can observe several modeling requirements when designing resource-aware systems according to this approach; namely the need to:

1. *Perform optimizations in various locations within the system*: Depending on the particular system and optimization techniques, optimizations may involve and affect multiple software modules. There is no single optimization technique that is generally applicable; for example, in the *Hiker's Buddy*, there is a more centralized con-

trol, whereas in the *Smart Phone* case, the optimization involves network-layer support, a centralized controller, and application-level involvement.

2. *Separate optimization and functionality*: To manage complexity and evolution, optimization and functionality should be separated, even though these are—to varying degrees—inter-dependent. This is also exemplified by both case studies, where optimization takes place *outside* the various modules in the system—but it does require communication with those modules.
3. *Separate functional and resource-usage interfaces*: We must provide explicit interfaces for both functionality *and* resource usage—both required and provided resources. This is not very strongly emphasized by the case studies, but derived from the need to manage and represent resources explicitly (so they can be optimized), as well as the need for independent (functional) modules.
4. *Know the behavior and/or plans of other modules w.r.t. resource usage*: Advanced optimizations do not only rely on measuring the state of the system (such as resource usage), but need information about the ongoing and planned activities from the involved modules. For example, in the *Smart Phone* case, the optimizing *controller* must know when the applications expect to reconnect.

The above list emphasizes the engineering requirements for separation of concerns during modeling. Separate concerns can then be implemented in independent modules (we deliberately disregard technology specifics here). However, these modules can in some cases be tightly related, and always need to be composed into a single integrated system. We can now state the following requirements upon the techniques for the composition of modules in a resource-aware system. We must be able to:

1. express modular optimization components that cross-cut the system, such as:
   - the ability to observe the resource usage and availability of other modules, so that optimization techniques can take the correct decisions, or
   - the ability to affect or control certain behavior or state of other modules for optimization purposes.
2. extend the functional interfaces with relevant resource usage information; e.g. how much resources a certain service consumes when invoked.
3. compose the RUM with the functional implementation; these two are tightly intertwined, but need to be implemented modularly.
4. provide information about the resource usage *behavior* (such as the RUM) of modules to other modules, for optimization purposes.

5. Conclusions and Future Work

To achieve resource-aware applications, we need to explicitly model: the components utilizing a resource of interest, the relation between functional services and resource utilization of components, and the interaction among components jointly utilizing a resource. Due to the lack of a suitable notation, this paper proposes one and shows its usefulness by means of two real-world examples. Further, to achieve modularity in resource-aware applications, we must be able to modularize: the optimization components, the resource-utilization model of components, and the functionality of components. Also, we should be able to compose application-specific and optimization components by augmenting the component interfaces with necessary resource-utilization information.

As future work, we will investigate the implementation of resource-aware, self-adaptive applications and evaluate the suitability of current languages—such as aspect-oriented languages—for achieving modularity in these implementations. As another future direction, we will investigate a standard notation for representing resource-utilization models such that various kinds of discrete and possibly also continuous models can be expressed.

References

[1] H. Ammar, V. Cortellessa, and A. Ibrahim. Modeling resources in a UML-based simulative environment. In *Proceedings of AICCSA*, pages 405–410. IEEE, 2001.

[2] B. Cheng, R. Lemos, P. Inverardi, and J. Magee, editors. *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*. Springer, 2009.

[3] C. S. Ellis. The case for higher-level power management. In *Proceedings of HOTOS*, pages 162–167. IEEE, 1999.

[4] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J. M. Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *Proceedings of SIGMETRICS*, number 1, pages 252–263. ACM, 2000.

[5] OMG. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Technical Report formal/2009-11-02, OMG, 2011.

[6] T. Patikirikorala, A. Colman, and J. Han. A multi-model framework to implement self-managing control systems for QoS management. In *Proceedings of SEAMS*, pages 218–227. ACM, 2011.

[7] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. TOP: Tail optimization protocol for cellular radio resource allocation. In *Proceedings of ICNP*, pages 285–294. IEEE, 2010.

[8] C. Schurgers, V. Raghunathan, and M. B. Srivastava. Modulation scaling for real-time energy aware packet scheduling. In *Proceedings of GLOBECOM*, pages 3653–3657. IEEE, 2001.

[9] C. Seceleanu, A. Vulgarakis, and P. Pettersson. REMES: A resource model for embedded systems. In *Proceedings of ICECCS*, pages 84–94. IEEE, 2009.

[10] B. Steigerwald and A. Agrawal. Developing Green Software. Technical report, Intel Corporation, 2011.

[11] C. Szyperski. *Component software: beyond object-oriented programming*. ACM, 1998.