A Design Method For Modular Energy-Aware Software

Steven te Brinke, Somayeh Malakuti, Christoph Bockisch, Lodewijk Bergmans, and Mehmet Akşit
University of Twente – Software Engineering group – Enschede, The Netherlands
{brinkes, malakutis, c.m.bockisch, bergmans, aksit}@cs.utwente.nl

ABSTRACT

Nowadays reducing the overall energy consumption of software is important. A well-known solution is extending the functionality of software with energy optimizers, which monitor the energy consumption of software and adapt it accordingly. To make such extensions manageable and to cope with the complexity of the software, modular design of energy-aware software is necessary. Therefore, this paper proposes a dedicated design method for energy-aware software.

Categories and Subject Descriptors

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

General Terms

Design, Performance

Keywords

energy-aware software, modularity, design method

1. INTRODUCTION

Green computing emphasizes the need for reducing the environmental impacts of IT solutions by reducing their energy consumption. Green computing can be achieved by making software energy-aware by augmenting it with so-called *energy optimizers*, which monitor the energy consumption of the software during its execution and optimize it accordingly.

Today's software is already facing the problem of complexity [7], and extending its functionality with energy optimizers increases this problem. Modularization is commonly considered as means to cope with the complexity of the software, because the scope of focus can be reduced to individual modules [6], which communicate with each other through well-defined interfaces.

To cope with the complexity of energy-aware software, we claim that energy optimizers must be modularized. This can be understood as the separation of the functional and optimization concerns. Energy optimizers need to gather necessary information from functional components to optimize their energy consumption. For exchanging this information, functional and optimizer components must provide suitable interfaces to each other. Along this line, we have proposed a dedicated notation for modeling such components [5].

However, a modeling notation is not sufficient to achieve modularity in the design of energy-aware software. In this paper, we propose a design method to guide designers through the activities performed to identify and modularly design (1) necessary components and the energy-specific interfaces of these components, (2) the models that must be prepared during each activity, and (3) the analysis that must be performed on the models. More details of the design method and a concrete realization to modularly design a real-life media player are described in a technical report [2].

SAC'13, March 18–22, 2013, Coimbra, Portugal.

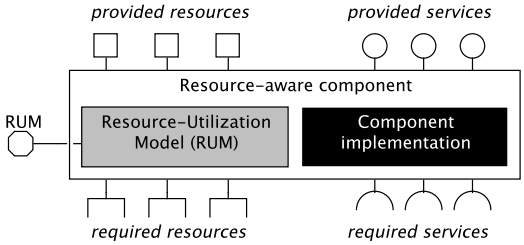

Figure 1: Resource-Aware Component Notation

2. NOTATION FOR COMPONENTS

Figure 1 depicts a notation—which we proposed earlier [5]—for modeling resource-aware components. We consider energy as a special kind of resource; a component may consume various resources, which eventually may lead to the consumption of energy. Therefore, such resources must also be taken into account. The provided services, required services, provided or required resources are specified in terms of a name and signature of a single attribute.

In contrast to the services and resources, the resource utilization model (RUM) of the component is more complex, because it represents the relations between all the services and resources provided and required by the component. The RUM is represented as the light gray box inside the component and exposed through the octagonal port; each component has only one RUM. It has already been proposed to express the RUM as a state chart in which states are annotated with resource behavior, and invocations on the services of the component are modeled as transitions.

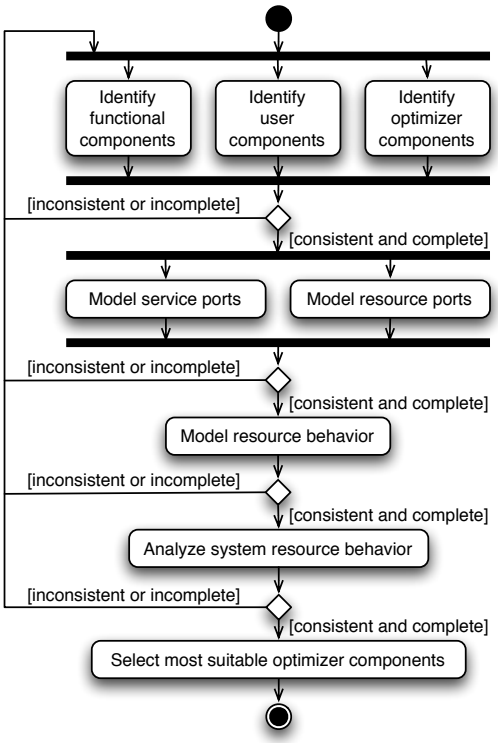

Figure 2: Design method for energy-aware software

3. DESIGN METHOD

This paper proposes a method to design energy-aware software systems such that modularity is achieved in the design of such systems. Figure 2 is a UML activity diagram that depicts the activities that are performed in our design method, along with the order and dependency of activities. The activities are depicted as boxes and each activity results in a model. The arrows represent the order of activities; the activities between bars can be performed in any order. The diamonds represent points in our method where the models are evaluated and possibly are redesigned iteratively, e.g., by changing the decomposition, or adding details. The activities result in a set of modeled components, which are represented in the notation depicted in Figure 1.

While our design method can always be applied with pen and paper, if automatic analyses are desired, a tool must be used. In a technical report [2], we present more details about the design method and our concrete experience with the model checker Uppaal.

3.1 Identify components

At the top of the diagram are the activities for identifying the software components. The design method distinguishes three kinds of components: *functional*, *user* and *optimizer*.

The *functional* components refer to both software and hardware components which form the target system; each component implements part of the functionality of the system and interacts with other components to accomplish the overall functionality. Since there are already various guidelines for modularizing functional aspects of software [6, 3], our method suggests to adopt an existing guideline to identify functional components.

The way in which *users* interact with software can have a large influence on the overall energy consumption [4]. For example, if a media-player application adopts a caching mechanism, caching would not be effective if a user disturbs the normal stream of video by seeking forward and backward. Therefore, to be able to analyze the effectiveness of an optimization strategy, it is needed to model the usage scenarios.

Since it is not possible to foresee all possible usage scenarios, it is desirable to at least model most common scenarios. Our design method represents these scenarios as user components. To facilitate analysis of resource consumption, we propose to model deterministic user behavior.

The *optimizer* components monitor and adapt the resource consumption of functional components during execution. For this matter, they interact with functional components. Our design method distinguishes optimizer components from other functional components to emphasize that they must be modularized so that various optimization strategies can be used interchangeably during the concrete design.

3.2 Model ports

The *service* ports of a component represent the functionality provided and required by the component. These ports are part of the interface of the component and must be modeled after the components are identified.

Both software and hardware functional components can provide and consume *resources*; the kind of resources may be different for hardware and software components. For example, a hardware device consumes electric energy as resource; whereas a software component may require a network connection or a buffer as a resource.

To be able to analyze resource consumption of components, the provided and required resources must be specified as the interface of the components. In addition, the inter-connection among the components must be specified so that a provided resource can be consumed by other components. Therefore, our design method proposes the following approach for modeling the resource ports.

First, identify a resource with respect to which the software should be optimized and the components that directly require this resource. Second, add the corresponding required resource ports to these components. Components that consume the resource of interest in general also provide resources which relate to the resource of interest (e.g., because they consume the initial resource in order to provide another resource at a higher abstraction level). Such related resources must be added as provided resource ports to the components. This activity must be applied recursively until all relevant resources are identified.

3.3 Model resource behavior

After modeling the components and their interfaces, we model their *resource behavior*. The resource behavior is the dynamic resource consumption and provision of components during their execution, and we represent them via RUMs (see Figure 1). This dynamic relation can, for example, include the following information: which resources are consumed or produced by each service port, and how switching among the services influences the resource consumption.

RUMs are energy state charts [1] that represent the resource behavior of components. The RUM of a component can be modeled, for example, through the following activities.

Start by identifying states with distinct functional behav-

ior of a component, for example, by considering the functionality defined as service ports, or by referring to the requirements or hardware specification. Define transitions between these states that can happen at events related to service invocation and possibly other events such as time-outs. Next, add the resource consumption to the model by annotating each state with the amount of resources it consumes or produces; thereby you must refer to the required and provided resource ports of the component.

States with multiple characteristics of resource usage must be split into multiple states. Now, identify the possible transitions between these new states and the already existing states, and events triggering these transitions. This step must be applied recursively until all states have a single characteristic of resource consumption.

3.4 Analysis

Our design method considers checking activities after each modeling step. Examples of checks are: checking whether any component is missing; checking whether the required interfaces of components are bound to compatible provided interfaces of other components, etc. If a model fails such checks, the initial activities are performed again in a new iteration to improve the design until it passes the checks.

When at least an initial version of all models exists, the effectiveness of various optimizer components can be analyzed. Specific to our method is the analysis of the system's resource behavior using RUMs. This can, for example, be used to analyze which optimization results in the least resource consumption. Based on this analysis, designers can select a composition for the final system design using the most suitable one among alternative optimizer components.

4. RELATED WORK

This section discusses two categories of related work. For each category, we only compare our work with one representative paper. A more extensive review of related work can be found in a technical report [2].

A wide range of techniques and mechanisms are proposed for making software green. However, there is a lack of methods and techniques that take software modularity into account. For example, adopting the solution proposed by Gotz et al. [4] requires to design and implement the software such that it complies with their component model. However, this might not be an effective solution; first because large-scale commercial software might not be implemented in this component model; and second there are already a large number of legacy software systems that must be extended with energy optimization functionality, and re-implementing them according to a new component model might not be considered suitable. In our design model we reused the existing notation and activities to design modular software and extended them with energy-specific notations and activities.

This paper focuses on optimization of energy usage, but our work can more generally be used to specify the software's usage of arbitrary resources. This relates to modeling the non-functional properties of services. Zschaler [8] proposes a semantic framework for the specification of non-functional properties. This framework allows checking whether a certain composition meets the quality goals. However, the framework only reasons about a single composition; it does not know how the composition can be improved. Therefore, Zschaler's approach does not support optimization. In contrast, our approach specifies the amount of resource usage and how this can be influenced, to facilitate optimization.

5. CONCLUSIONS AND FUTURE WORK

This paper discussed the need for designing energy-aware software in a modular way, and proposed a method for this matter. We also applied the method to a media-player as a case study. This case study, including more details about our approach, is described in a technical report [2].

One of the challenges when applying the approach is how much detail should be added to resource utilization models. RUMs are part of the interface of a component, so it is desired that RUMs do not contain any implementation details. However, some implementation details might be needed to explain the resource behavior. This increases the coupling: Controllers that depend on the precise resource behavior given in the RUM, might not be applicable to implementations that have different resource behavior. Therefore, it might be desired to add less detail to the RUM, even though it could hinder some optimization strategies. We will address the desired level of detail in RUMs in future work.

As future work we will apply this method to large-scale systems such as industrial printers. In addition, we will extend our notation to be able to model diverse kinds of adaptation actions that energy optimizers can perform on functional components. We will also investigate suitable programming mechanisms to implement and modularize the software that is designed according to our method; aspect-oriented languages are among our first candidates.

6. REFERENCES

[1] L. Benini, R. Hodgson, and P. Siegel. System-level power estimation and optimization. In *Proc. Int. Sympos. Low Power Electron. Des.*, pages 173–178, Aug. 1998.

[2] S. te Brinke, S. Malakuti, C. M. Bockisch, L. M. J. Bergmans, and M. Akşit. A design method for modular energy-aware software. Technical Report TR-CTIT-12-28, Centre for Telematics and Information Technology, University of Twente, Enschede, Nov. 2012.

[3] J. Garland and R. Anthony. *Large-Scale Software Architecture: A Practical Guide using UML*. Willey, 1st edition, 2003.

[4] S. Gotz, C. Wilke, S. Cech, and U. Assmann. Architecture and mechanisms for energy auto tuning. In *Proc. Sustainable ICTs and Management Systems for Green Computing*, 2012.

[5] S. Malakuti Khah Olun Abadi, S. te Brinke, L. M. J. Bergmans, and C. M. Bockisch. Towards modular resource-aware applications. In *Proc. 3rd Int. Workshop on Variability & Composition (VariComp 2012)*, pages 13–17, New York, March 2012. ACM.

[6] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[7] W. Royce. Improving software economics-top 10 principles of achieving agility at scale. White paper, IBM Rational, May 2009.

[8] S. Zschaler. Formal specification of non-functional properties of component-based software systems. *Software and Systems Modelling*, 9:161–201, 2009.