A Design Method For Modular Energy-Aware Software

Technical Report

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

November 2012

Nowadays achieving green software by reducing the overall energy consumption of the software is becoming more and more important. A well-known solution is to make the software energy-aware by extending its functionality with energy optimizers, which monitor the energy consumption of software and adapt it accordingly. Modular design of energy-aware software is necessary to make the extensions manageable and to cope with the complexity of the software. To this aim, we require suitable methods that guide designers through the necessary design activities and the models that must be prepared during each activity. Despite its importance, such a method is not investigated in the literature. This paper proposes a dedicated design method for energy-aware software, discusses a concrete realization of this method, and—by means of a concrete example—illustrates the suitability of this method in achieving modularity.

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.

To reduce the energy consumption of software, it is necessary to identify the functional components that directly or indirectly use energy, to identify the usage patterns of these components, and finally to influence these usage patterns to reduce the energy consumption. For this matter, energy optimizers need to interact with multiple components of software, gather necessary information from them and optimize their energy consumption. Note that the functionality of energy optimization can be embedded in the functional components of software.

Today's software is already facing the problem of complexity [10], and extending its functionality with energy optimizers increases this problem further. For example, if a functional component evolves such that functionalities are added or removed, its energy consumption may change, and the energy optimizers must be changed accordingly to consider the new amount of energy consumption of the component.

In the software engineering literature, 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 [9]. In the literature, a module is defined as a reusable software unit with well-defined interfaces and an implementation. Modules are communicating with each other through their interfaces, which implies that depending on the kind of communication, the interfaces must convey sufficient information to facilitate the communication.

To cope with the complexity of energy-aware software, we claim that energy optimizers must be modularized from the rest of software. This can easily be understood as the separation of the functional concerns and optimization concerns. To this aim, functional and optimizer components must provide suitable interfaces to each other, so that necessary information for performing the optimization can be exchanged among them. Along this line, a dedicated notation for modeling such components has been proposed [8].

However, the presence of a modeling notation is not sufficient to achieve modularity in the design of energy-aware software. We need to guide designers through the activities that must be performed to identify and modularly design (1) necessary components, (2) the models that must be prepared during each activity, and (3) the necessary analysis that must be performed on the models.

This paper proposes a dedicated design method for energy-aware software. The design method helps designers to identify the kinds of components that are typically needed to be taken into account, the energy-specific interfaces of these components, their interaction with each other and the analyses that must be performed on them. This paper explains a concrete realization of this method by means of the Uppaal tool [2], and illustrates the suitability of Uppaal to analyze the design models. An application of this design method to modularly design a real-life media-player software is illustrated.

This paper is organized as follows. As the background, Section 2 explains the notation that is adopted in this paper for modeling the components of energy-aware software. Section 3 explains our design method; Section 4 explains the realization of the method in Uppaal and Section 5 shows the application of the method to the media-player application. Section 6 outlines the lessons that we learnt from applying this design method; Section 7 explains the related work and Section 8 outlines the future work and conclusion.

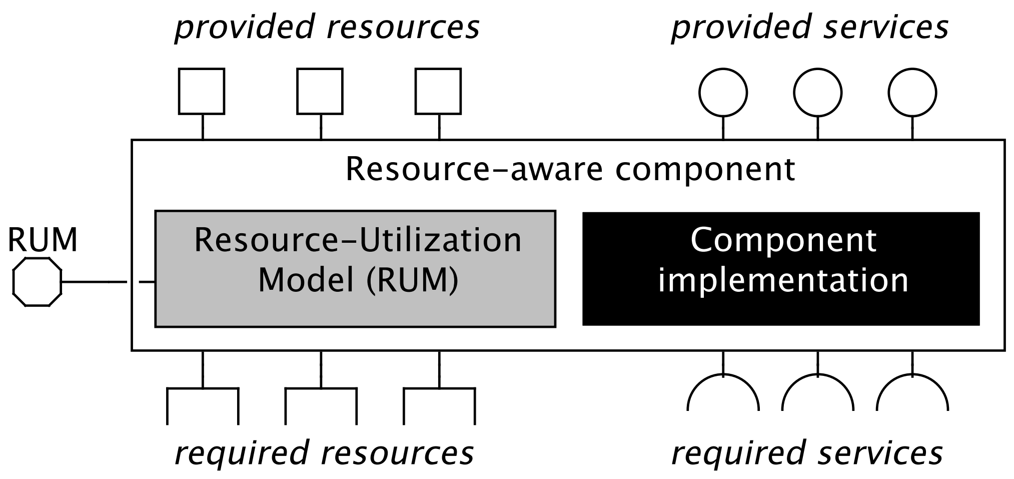

Figure 1: Notation for Resource-Aware Components

2 Notation for Energy-Aware Components

Figure 1 depicts a notation [8] for modeling resource-aware components[1]. 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. Since the RUM is too complex to be represented as an annotation to a port, it is represented as the light gray box inside the component and exposed through the octagonal port. This is also motivated by the fact that there is only one, *the* RUM of the component. 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.

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 represented by boxes in the figure and each activity results in a model. The arrows

[1]In this paper, 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, these resources must also be taken into account.

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.

In the next subsection, we will give an overview of the activities in our method. Those specific to our method will be elaborated in the subsection thereafter.

3.1 Overview

At the top of the diagram are the activities for identifying the components of the software. The design method distinguishes among 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 software and hardware components to accomplish the overall functionality. Since there are already various guidelines for modularizing functional aspects of software [9, 6], our method suggests to adopt an existing guideline to identify functional components.

The way in which users interact with software plays a role in its energy-consumption [7]. To be able to analyze the effectiveness of an optimization strategy, our design method considers *user* as a component of the system, which represents usage scenarios.

The *optimizer* components modularize the optimization strategies; for this matter they interact with both functional and user components.

After identifying and modeling the components, their interface must be modeled. As explained in the previous section, our notation considers both *service* and *resource* interfaces for the components. The service ports represent the functionality provided and required by a component. The resource ports represent the component's state of resource consumption, information which is required by optimizer components.

After modeling the components and their service/resource interfaces, we require to model the *resource behavior* of the components. The resource behavior represents the dynamic resource consumption and provision of components during their execution, and we represent them via RUMs (see Figure 1).

Our design method considers dedicated checking activities after each modeling step. Typical 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; checking the safety and liveness of the models, etc. If a model is considered to have problems, the initial activities are performed again in a new iteration to resolve the problems.

When at least an initial version of all models exists, the effectiveness of various optimizer components—in terms of the reduced energy consumption—can be analyzed. When the models are sufficiently precise for this analysis to be performed, designers can select an optimizer component based on the analysis' results.

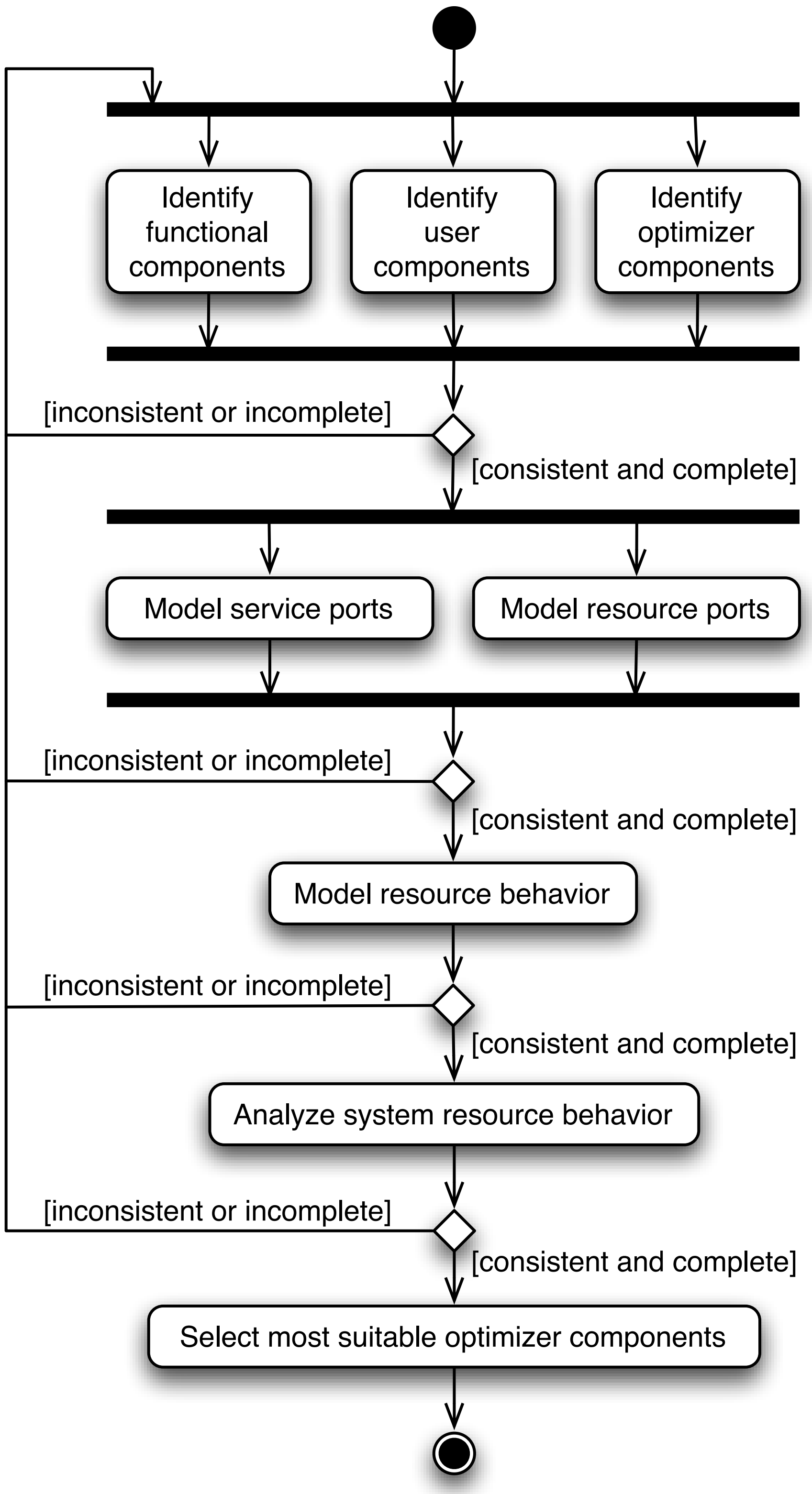
Identify functional components
Identify user components
Identify optimizer components
[inconsistent or incomplete]
[consistent and complete]
Model service ports
Model resource ports
[inconsistent or incomplete]
[consistent and complete]
Model resource behavior
[inconsistent or incomplete]
[consistent and complete]
Analyze system resource behavior
[inconsistent or incomplete]
[consistent and complete]
Select most suitable optimizer components

Figure 2: Design method for energy-aware software

3.2 Design Activities in Detail

This subsection will explain those activities of the methodology outlined above in detail, which are specific to our approach. We skip details about the design of functional components and services, since this is an activity commonly carried out in all component-based software development approaches. We start with the analysis performed throughout the design process and motivate from there the other activities and related models.

3.2.1 Analysis

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.

Since a system is composed of multiple components, which are specifying a RUM, the overall behavior of system is represented by a composition of multiple RUMs. Such a composition of RUMs easily becomes complex, which makes the manual analysis of RUMs difficult and error-prone. Therefore, we claim that the analysis must be automated by means of suitable tools. While the choice of tools is not dictated by our method, automated analysis implies that RUMs must be specified precise enough to perform automated reasoning about some properties.

3.2.2 Model Resource Ports of Functional Components

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 resources that are provided and required by the components 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 that are in need. For this matter, in our design method we propose 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. For example, if the energy consumption should be optimized, components that directly use hardware components receive energy as a required resource port. 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.

3.2.3 Model Resource Behavior of Functional Components

After modeling the service/resource ports of the components, we need to specify the resource behavior of components, which represents the dynamic relation between the service and resource ports. This dynamic relation can, for example, include the following information: which resources are required or produced by each service port; the amount of the resources; how the amount changes during the execution of a service; how switching among the services influences the resource consumption and provision; etc.

RUM is an energy state chart and facilitates representing resource behavior of components. RUMs can be added to components, for example, through the following activities.

Start by identifying states with distinct functional behavior 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 timeouts. 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.

3.2.4 Model User Components

The overall energy consumption of a system and the effectiveness of an optimization strategy can be largely influenced by the way how the system is used. In general, different kinds of users exist, ranging from human users to other systems; in this paper we uniformly use the term user to refer to any kind of client actively using the system under design. For example, if a media-player application adopts a caching mechanism to reduce the amount of network connections, 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. In our design method, we represent these scenarios as user components. All functionality that is invoked during the scenario is modeled as required service ports. Based on these services, a RUM is designed, describing how these services are used, as discussed in Section 3.2.3.

The simplest specification is a user that might do anything non-deterministically. Then the RUM is a non-deterministic state machine that has one state with self-loop transitions for every service that can be invoked. Such a specification can be useful for analyzing the correctness of the system, but—in general—is not useful for analyzing the resource consumption.

On the contrary, to facilitate analysis of resource consumption, we propose to model deterministic user behavior. It is possible to specify multiple different user behaviors as alternative components. Then distinct analyses can be performed using each of the available user components to identify the usage scenario under which a given optimization performs best.

3.2.5 Model Optimizer Components

The optimizer components implement the functionality to monitor and adapt the resource consumption of functional components during execution. For this matter, they interact with functional components. The optimizer components can be regarded as special kinds of functional components, too. Nevertheless, our design method distinguishes them from other functional components to emphasize that they must be modularized from the other components so that various optimization strategies can be used interchangeably.

Before adding an optimizer component, the functional components interact directly. There are two main kinds of components: high-level components are those directly controlled by the user behavior and low-level components are those that directly consume the resource to be optimized. An optimizer component must be inserted as an indirection to the interaction between low-level and high-level components.

There can also be intermediary components through which the high and low-level components interact. When an optimizer component is inserted between such intermediary components, the components closest to the high-level components play the role of the high-level components; this is analogous for the low-level components.

The optimizer component must have as provided interface the services and resources that are required by the higher-level component; the required interface accordingly mirrors the provided services and resources of the lower-level component. The optimization logic must be modeled as the component's RUM following the guidelines in Section 3.2.3.

As for the user behavior, it is also possible to design multiple alternative optimizer components. In the analysis phase, different combinations of user behavior and optimizer components can be analyzed. That way, the designer can identify which of the optimizer components is optimal for which expected user behavior.

4 Design Method using Uppaal

While the design method presented in the previous section can always be applied with pen and paper, if automatic analyses are desired, a tool must be used. We could either develop a dedicated tool to analyze resource consumption or use an existing tool. The advantage of a dedicated tool is that the tool will support exactly our notation, the disadvantage is that the development of a tool is error prone and time consuming. The advantage of using an existing tool is that it can be used immediately, the disadvantage is that we may need to adapt our models to the tool, which may have negative influence on the modularity of the models.

Uppaal [2] is a robust tool to model, verify, and validate timed automata, which can represent power state charts. Also, scenarios can be modeled as timed automata. In this paper, we present our concrete experience with using Uppaal, however, other model checking tools can be used as well.

The following subsection introduces the features of Uppaal we used. The subsection thereafter explains in detail how Uppaal can be used as part of our approach.

4.1 Overview of Uppaal

Here we give a brief and simplified explanation of the concepts in Uppaal and map them to the terminology used in this paper. In Uppaal, a system is modeled as several timed automata in parallel. A timed automaton is a finite-state machine with numeric and clock variables.

Transitions can have a so-called synchronization label, which can have one of two meanings: First, it can be specified that the transition is triggered when the event corresponding to the label occurs; second, it can be specified that the corresponding event is emitted when the transition is taken. Transitions can be annotated with guards, which are side-effect free expressions; a transition is only enabled when the guard evaluates to true. And they can contain assignments that update variables.

States can be annotated with side-effect free expressions, called invariants. They restrict the time during which the system can be in a certain state. So-called committed states can be used to represent actions that must occur immediately after each other. A committed state cannot delay and the next transition must involve an outgoing transition of at least one of the committed states in one of the parallel automata.

4.2 Design Activities using Uppaal

In this section we describe the design activities in which Uppaal is used.

4.2.1 Analysis

After designing a model, the designer can simulate behavior in Uppaal by stepping through the diagram semi-automatically, which is the simplest form of analysis that Uppaal facilitates and is mainly useful in the early stages of development. When the state charts become larger, it becomes more useful to let Uppaal fully do the model checking. Uppaal provides a subset of timed computation tree logic (TCTL), in which many properties can be expressed, such as safety and liveness properties. For example, we can check whether our model is free of deadlocks. Whenever a given property does not hold, Uppaal provides a counterexample, that can be used by the designer to analyze whether the model or the property specification is incorrect.

When we have specified a scenario of expected user behavior, Uppaal can analyze the behavior of the system and output a trace of the expected behavior. Such a trace contains the resource consumption over time. Thus, by analyzing different optimization components, the optimization that consumes least resources can be identified.

4.2.2 Model Resource Ports

The structure of the application is modeled in Uppaal by creating an automaton for every component. In Uppaal variables and labels can be local to an automaton or global to all parallel automata. Everything that is on the interface of a component is globally accessible. Therefore, we define global labels for all services and global variables for all resources present in the interface. Variables that are written to correspond to provided resource ports, read variables correspond to required resource ports.

4.2.3 Model Resource Behavior

The contents of the automaton—which was created to model the component—represent the resource behavior. Such automaton can be designed using the steps presented in the general approach; we only explain the Uppaal-specific details here.

Every transition which changes the consumption of a resource by the component is annotated with an assignment that updates the value of the corresponding variable to the current consumption. The resource behavior is linked to the functional interface by adding synchronization labels to transitions. For example, when *play* is defined as a service on the interface, the label *play!* represents requiring (invoking) the service and *play?* represents providing the service.

4.2.4 Model User Components

The behavior of a user can also be modeled as an automaton. A non-deterministic user can be modeled in Uppaal as elaborated in the general approach (cf. Section 3.2.4). Scenarios where the user performs certain actions after a specified amount of time can be modeled by adding timing constraints to transitions as guards and to states as invariants.

4.2.5 Model Optimizer Components

The definition of an automaton in Uppaal is actually a template: The automaton must be instantiated before it can be used. This can be used to provide alternatives. When alternative components are desired, multiple components with the same interface can be defined. Then, replacing one component with another is as simple as replacing one statement: the instantiation of the controller. Since templates can take parameters, parameter-based adaptation is just as simple: Define a component that takes a parameter and pass a value to it during instantiation.

5 Media Player Case Study

As a case study we use a media player that runs on a smart phone; thus reducing its energy consumption is relevant. We started with the high-level structure model of the relevant software components of the smart phone, including the media player. Figure 3 shows the part of the model representing the media player component. Gradually, we formalized the resource behavior—shown informally in figure 3—until we ended at the

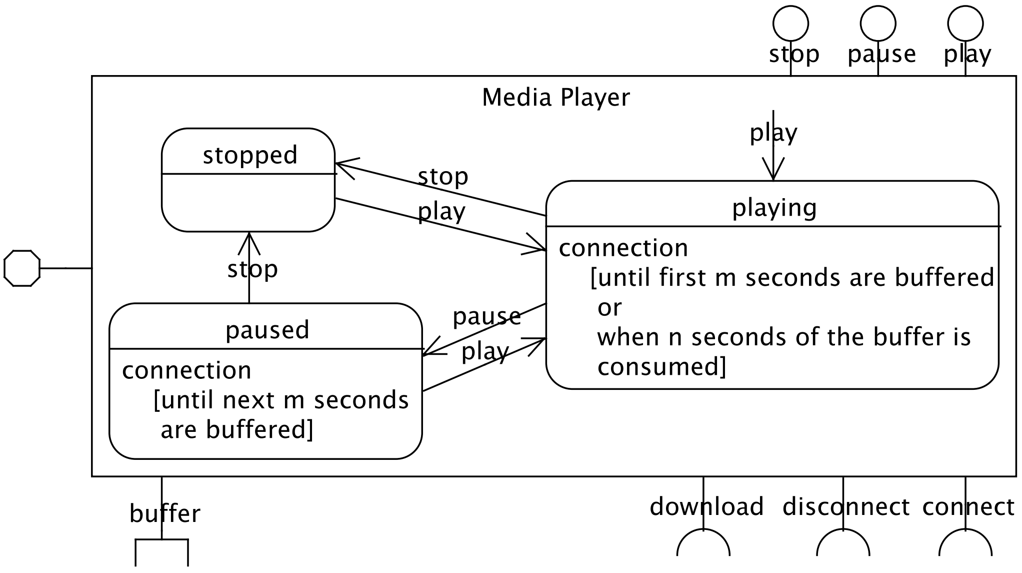

Figure 3: Media player high-level diagram

model shown in figure 4. Such a model can be created by starting with the states of the high-level model and iteratively adding more detail. These details are additional states and transitions that formally describe the resource behavior textually present in the high-level model. Such details are added until all relevant resource behavior is added.

Because this formalization was done using UPPAAL, it was possible to analyze the consistency and completeness of our model using TCTL formulas. With the help of this analysis, we gradually added more detail to the model. This was performed in the following five steps, which are iterative because adding detail to one part of the model might require adding detail to other parts as well.

Specifying the user: Verification requires all components to be specified. The high-level structure diagram did contain most components already, except for the user. Thus, we added a model of a simple, non-deterministic user that might do anything.

Splitting the state diagram into multiple diagrams, which are hierarchically composed: The high-level state diagram (figure 3) that represents the resource behavior of the system, contains various concerns, for example, buffering songs and playing songs. To maintain the modularity of the media player at the detailed level, we need to identify the various concerns and split the state diagram accordingly. To split the state diagram in a *player* and *buffer* process, an interface to communicate between these processes must be defined. For instance, we defined the events *start buffering* and *stop buffering.* The player component can generate these events to influence the buffer process.

Adding detail to the state diagram: The high-level state chart did not contain enough information to analyze the resource consumption. For example, it did not specify in

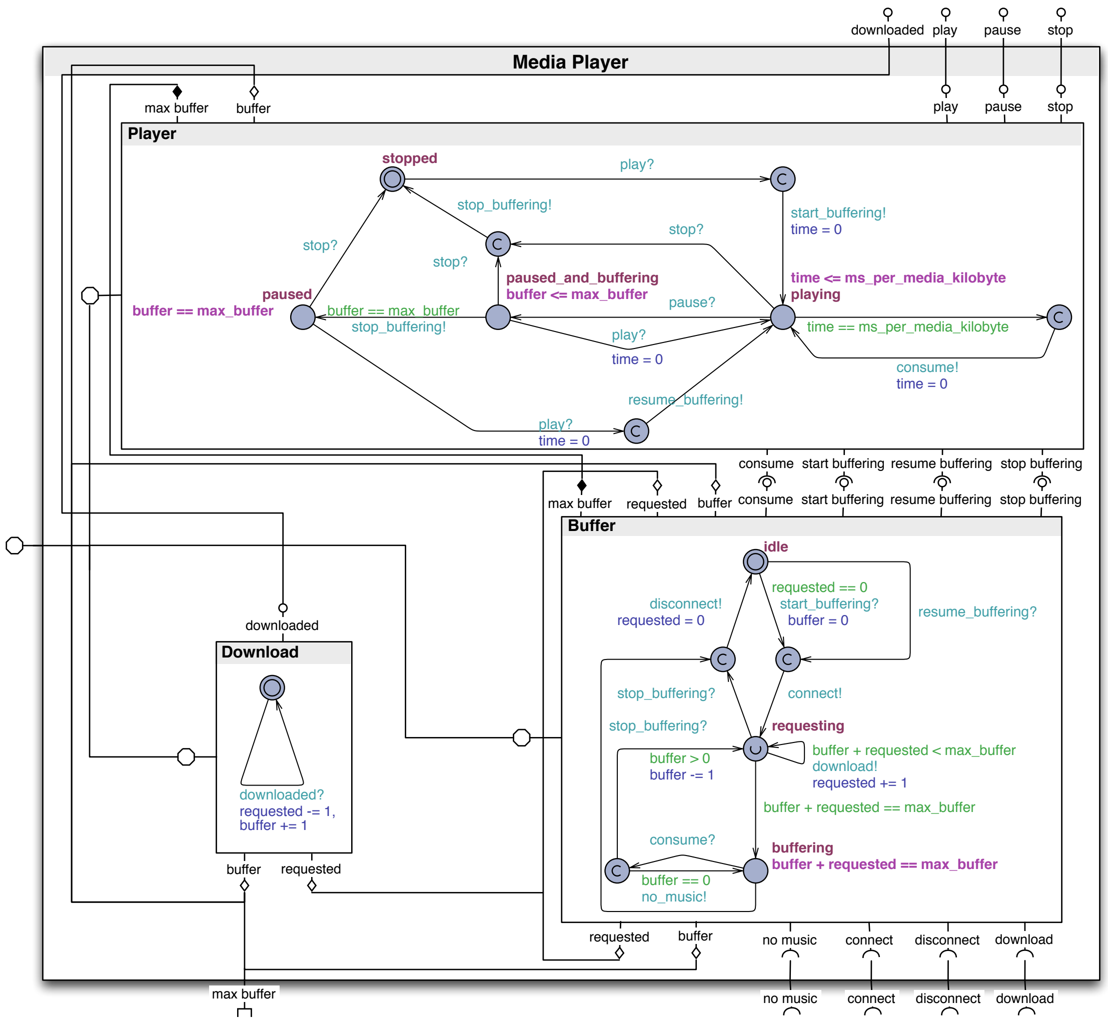

Figure 4: Media player detailed diagram

detail when buffering was performed. Therefore, we added details to the state diagram until we were able to analyze the resource consumption. Most importantly, we had to check that the model was correct and complete. For example, to check that the player actually stops playing when it cannot download music in time and to check that there are no situations for which no possible behavior is defined, i.e., deadlocks.

Adding detail to the interface: You can see that we also augmented the interface of the media player with two services: the required service *no music* and the provided service *downloaded*. These two services do not correspond to the component's functional behavior; instead, they are required to express assumptions about the component's internal behavior that are made by the RUM. Since the RUM is an essential part of the component's interface, it is legitimate and desirable to also promote these details to the component interface in terms of provided and required ports.

The required service *no music* represents the occurrence of an error in the media player component, namely that no music can be played because of the lack of data. This service does not have to be connected to any other component. (Requiring an unconnected service is an error, but since *no music* is an error state anyway, leaving it unconnected does not change the behavior of the media player.) However, explicitly modeling *no music* has the benefit that it provides us the possibility to reason about the situations in which this error condition occurs. Thus, our model of *no music* is a pattern that can be used for modeling error conditions: Model errors by explicitly communicating the error through the interface as a required service.

The provided service *downloaded* is added to model that downloading music is asynchronous. Thus, *downloaded* is the callback event that is fired when a download (started using a *download* event) is finished. Since state machines always run in parallel in UPPAAL, the general pattern for an asynchronous request is to create a state machine that handles this request, is started using a *start?* event, and communicates the result back using a *result!* event.

Selecting the most suitable optimizer component: To test various configurations, we created three different controllers. Such a controller is a mediator between the media player and network manager, as can be seen in figure 5. The three controllers are:

1. Basic controller providing the unoptimized behavior.
2. Burst download controller that downloads as much music as possible at once and then waits until the next burst of music is needed.
3. Fast Dormancy controller that downloads music in bursts and switches the network manager to idle when such a burst download is finished.

These controllers were created in order. When we added the second controller, we only had to replace the controller and connect the previously unconnected resource *buffer*, since the controller uses the knowledge of the buffer size to perform its optimizations. The third controller also did not require any changes to other components. This controller uses the *demote* feature of the network, which was already present, but not yet connected. Thus, adding the third controller also required adding a single connection only.

To test the behavior of the system, we created a scenario (partially shown in figure 6). As adaptation we used the three different controllers described before, while running the scenario. The energy consumption during these runs is shown in figure 7. We see that the first controller consumes most and the third controller consumes least energy.

6 Lessons Learned

6.1 General Approach

One of the challenges, when carrying out the general approach, is how much detail should be added to resource utilization models. Since these models are part of the interface of a component, it is desired to hide some of the complexity of these models. Components should hide their implementation details from other components. Therefore, resource utilization models should also hide all implementation details that are not needed to

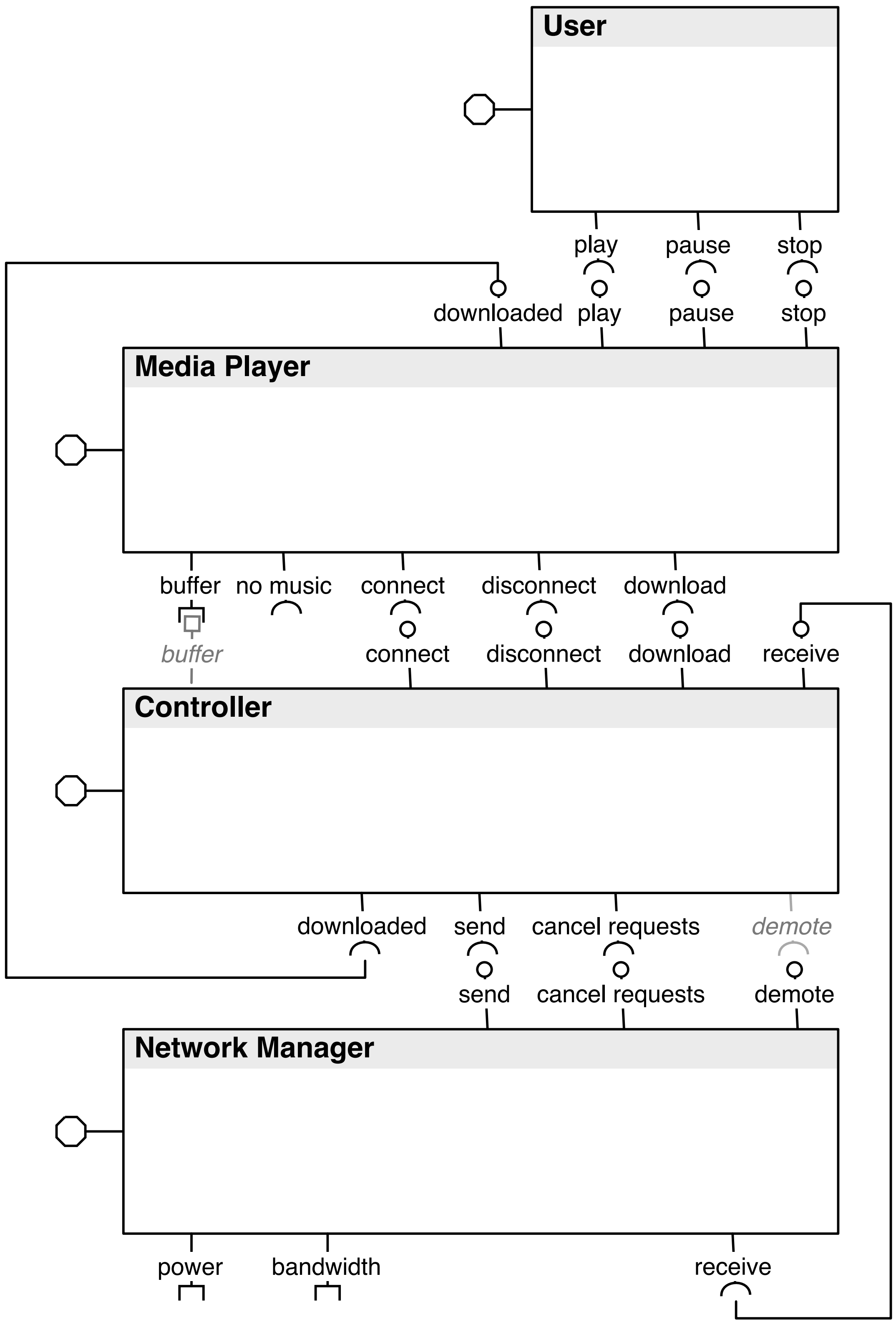

Figure 5: Media Player Overview

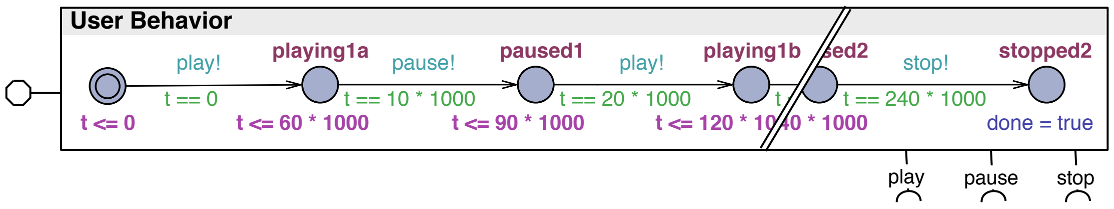

Figure 6: User behavior (partial scenario)

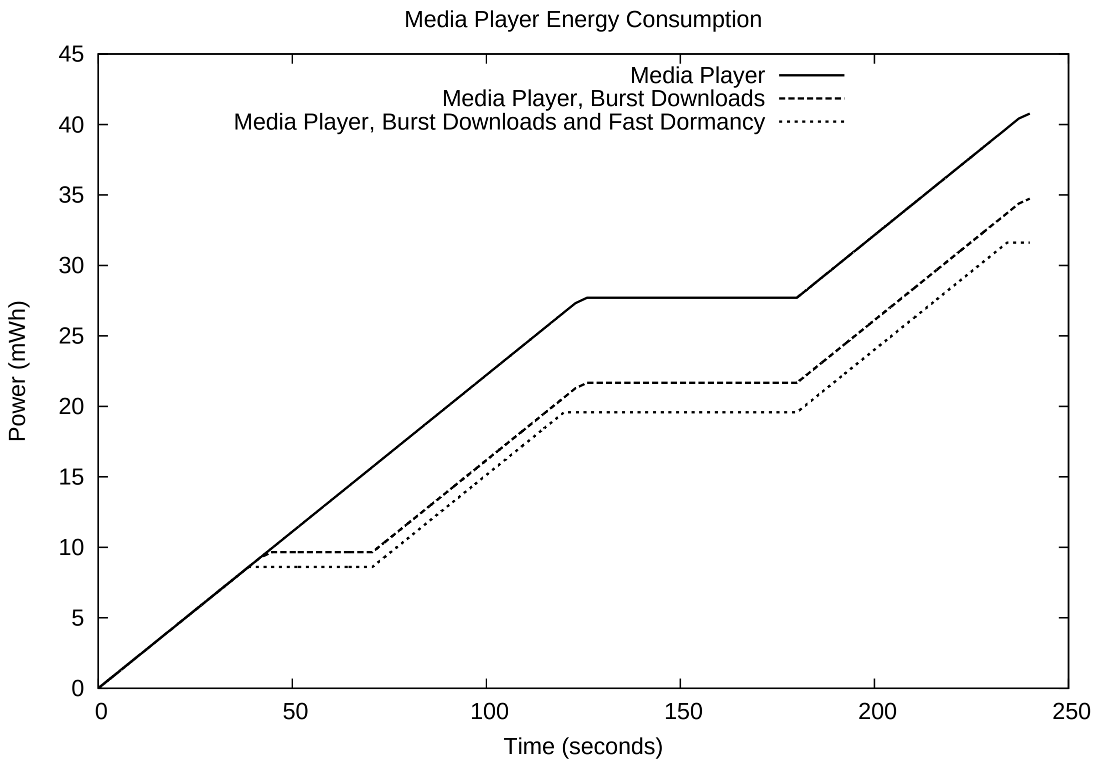

Figure 7: Analysis of scenario

explain the resource consumption. However, in our examples, the resource utilization models express quite some implementation details in order to specify the resource behavior as precisely as possible. 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, for example by specifying only which power states exist, not precisely when these states are activated. In such a case, the activation of the states should be communicated to the controller at runtime in order to allow optimization. This way, different components are more modular, which might be desired even though it could hinder some optimization strategies. We will address the desired level of detail in RUMs in future work.

6.2 Uppaal-based Approach

When performing the case study, we had a large benefit from Uppaal, but we also learned some disadvantages of using Uppaal. Even though Uppaal provides a good interface for inputting state charts, it does not match our notation perfectly, as described in the following three paragraphs.

The models used by Uppaal are not as modular as desired: Models cannot be composed hierarchically and interfaces cannot be visualized. In figure 4 we see that one component (the Media Player) consists of three other components. This hierarchy, including the interfaces of all four components, is drawn by hand. However, it would be possible to automate the conversion from hierarchical models to Uppaal models: David et al. [5] showed such a conversion using a subset of UML for the hierarchical models.

Uppaal does not provide an abstraction for *resources*. Instead, it provides *variables*, which—among other uses—can represent the resource consumption of relevant resources.

Uppaal can analyze the quality of a certain composition, but cannot choose the best composition. Analyzing alternative compositions and deciding which one is best, must be done separately, either by hand or using another tool. We did not look into tools that can facilitate this decision; this will be subject of future research.

7 Related Work

A wide range of techniques and mechanisms are being proposed for making software green. These are usually dedicated solutions or frameworks for facilitating optimizations, for example, at the level of operating systems [14], at the level of compilers [4], or at the system level [13]. Dedicated component models are also proposed [7] to enable modularizing energy-aware software. However, there is a lack of methods and techniques to augment (legacy) with energy-optimization functionality such that the target software is modular. For example, to adopt the solution proposed by Gotz et al. [7] we require to design and implement the software such that it complies with their proposed 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 proposed design model, however, we reused the existing notation and activities to design modular software and extended them with energy-specific notations and activities.

While this paper focuses on green computing, specifically optimization of energy usage, as an application area, our work can more generally be used to specify the software's usage of arbitrary resources. This relates to approaches for modeling the non-functional properties of services. Service-level agreements (SLAs) specify the desired non-functional properties of required services; for instance, Skene et al. [11] define an XML-based language for specifying the SLA between a client and a provider, based on OCL constraints. Zschaler [15] proposes a semantic framework for the specification of non-functional properties (using the Temporal Logic of Actions formalism) in a component-based software development approach. As is the case for other related approaches, Zschaler's framework only supports the specification of non-functional properties for single services; this is sufficient to analyze the non-functional properties of a given component-based architecture as a whole. In contrast, our approach also models the relations and dependencies between all services of a component and, thus, not only expresses its non-functional properties, but also specifies how they can be influenced. This is necessary to enable the modular implementation of optimizers.

Other related work which we will review in future are the works by Albertao [1], Steigerwald et al. [12], and Capra et al. [3].

8 Conclusions and Future Work

This paper discussed the need for designing energy-aware software in a modular way, and proposed a dedicated method for this matter. The paper illustrated a realization of this method by means of UPPAAL tool, and by means of an example showed that adopting this method can lead to modularized energy-aware software.

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.

References

[1] F. Albertao. Sustainable software development. In *Harnessing Green IT*, pages 63–83. John Wiley & Sons, Ltd, 2012.

[2] G. Behrmann, A. David, and K. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 33–35. Springer Berlin / Heidelberg, 2004.

[3] E. Capra, C. Francalanci, and S. A. Slaughter. Is software "green"? Application development environments and energy efficiency in open source applications. *Inf. Softw. Technol.*, 54(1):60–71, Jan. 2012.

[4] M. Cohen, H. S. Zhu, S. E. Emgin, and Y. D. Liu. Energy types. In *Proc. OOPSLA '12*, 2012, to appear.

[5] A. David, M. Möller, and W. Yi. Formal verification of UML statecharts with real-time extensions. In R.-D. Kutsche and H. Weber, editors, *Fundam. Approaches to Softw. Eng.*, volume 2306 of *LNCS*, pages 208–241. Springer Berlin / Heidelberg, 2002.

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

[7] 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.

[8] 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.

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

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

[11] J. Skene, D. D. Lamanna, and W. Emmerich. Precise service level agreements. In *Proc. 26th Int. Conf. on Software Engineering*, ICSE '04, pages 179–188, Washington, DC, USA, 2004. IEEE Computer Society.

[12] B. Steigerwald and A. Agrawal. Green software. In *Harnessing Green IT*, pages 39–62. John Wiley & Sons, Ltd, 2012.

[13] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye. Energy-driven integrated hardware- software optimizations using SimplePower. In *Proc. ISCA '00*, 2000.

[14] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: managing energy as a first class operating system resource. *SIGOPS Oper. Syst. Rev.*, 36(5):123–132, Oct. 2002.

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