Deriving Minimal Models for Resource Utilization

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

Shmuel Katz
Technion – Department of Computer Science – Haifa, Israel
katz@cs.technion.ac.il

Abstract

We show how compact Resource Utilization Models (RUMs) can be extracted from concrete overly-detailed models of systems or sub-systems in order to model energy-aware software. Using the Counterexample-Guided Abstraction Refinement (CEGAR) approach, along with model-checking tools, abstract models can be generated that help establish key properties relating to energy consumption. This approach is illustrated by the concrete example of a network manager sub-system. This work is part of an overall design methodology 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, model checking, CEGAR, resource utilization model, minimal abstraction

1. Introduction

Designing energy-aware software is increasingly important and solving the related challenges is an active field of research. In previous work [12, 13], we describe a methodology for modeling modular, energy-aware software. This approach is based on our notation for representing Resource Utilization Models (RUMs) of components [2], in which components express their resource utilization at their public interface. Energy is an important resource in this respect and is the focus of this paper, but reasoning about energy optimizations requires involving other resources, such as performance, as well.

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 extended components with explicit interfaces specifying the resources that it provides to and requires from its environment, and a RUM which expresses the relation between the component services and resources [2].

A RUM is an enhanced state chart, in which states reflect the various profiles of energy consumption. The transitions reflect service invocations on a component or internal events, which lead to changing the energy-consumption profile of the component (or consume energy per event). Components express their RUM at their public interface. The purpose of exposing the components’ energy usage on their interface is to enable the modular implementation of energy-optimization.

When a RUM is available for every component in the system, it is possible to analyze the overall energy consumption of the system. Such analysis can be performed by (dis)proving properties, such as: the system always consumes less than 1 J/s. Two kinds of properties can be proven: properties that hold for every execution sequence, and properties that hold for some execution sequences. Above is an example of the former; an example of the latter is: for some execution sequence, the system can play 20 s of music while consuming less than 5 J. Such properties can be checked by a model checker; we used the model checker UPPAAL [14, 15].

One of the challenges in our methodology is to define the RUM for a component. We distinguish two kinds of components: (1) components which are currently in development, and (2) components for which we already have an implementation and additional knowledge about resource usage, such as power consumption. For example, when developing an application for playback of media from the Internet, the Network Manager component may already exist while the player is to be newly developed.

For newly developed components, the task of designing the RUM is mainly creative. In our previous paper [12, 13], we give some initial guidelines for designing such RUMs. In this paper, we focus on guidelines for designing RUMs for already existing components.

A RUM must be an abstraction of the energy behavior of the implemented component. This means that behavior expressed by the RUM can be mapped to the behavior of the implementation, even though it does not contain all details of the implementation. Thus, the RUM represents—in an abstracted form—at least all execution sequences of the actual implementation, and possibly more behavior. Therefore, every property that can be proven to be true for every execution sequence in the RUM, is also true for the actual implementation. However, properties that are true for some execution sequence in the RUM, might not be true for the implementation, because these properties might only hold for the additional behavior that is only allowed in the abstraction. Thus, only properties that should be true for all execution sequences can be proven on the abstraction.

1 For clarity, we write energy consumption as J/s (Joules per second) instead of its equivalent W (Watts), because we express the total amount of consumed energy in Joules, not in Watt hours.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © 2013 ACM 978-1-4503-1866-2/13/03...$15.00
To decide which details must be part of the RUM, there are two concerns:

- The RUM must describe the relation between the services and resources that are—required to be—on the interface of its component.
- We must know which properties we want to prove. Therefore, these properties need to be identified first.

With respect to these properties, the RUM should be a minimal abstraction. A minimal RUM has the following advantages:

- Less implementation dependent. The implementation can be exchanged with any implementation that satisfies the RUM. For a minimal RUM, more such implementations exist.
- Easier to understand. A small RUM is easier to fully understand by human developers.
- Faster model checking. Model checking does not scale very well to large models. Therefore, it is only feasible on small models. Thus, keeping the RUM small is important to allow the models to be checked.

The Counterexample-Guided Abstraction Refinement (CEGAR) approach [4] partially automates the creation of such minimal abstractions for concrete models, when a detailed concrete model—which describes both the detailed behavior and the relevant energy consumption of this behavior—exists. For this reason, we propose in this paper to use the CEGAR approach for designing the RUMs of such components.

In the following two sections we will first summarize our proposed methodology for designing energy-aware software and explain the CEGAR approach. In Section 4 we will discuss the application of the CEGAR approach for defining a RUM, illustrated by an example. In Section 6 we consider the limitations of CEGAR and the continued need for originality.

2. Design Method for Resource-Aware Software

Figure 1 is a UML activity diagram that depicts the activities that are performed in our design method for energy-aware software systems with modularly implemented energy optimization [12, 13]. Compared to our previous work, we have added the first task, which is to identify the key properties with respect to which the resource usage abstraction should be minimal. In this paper, we focus on only two kinds of components that are identified as the next step in our methodology:

- The functional components refer to both software and hardware components which form the target system.
- The optimizer components modularize the optimization strategies; for this purpose they interact with functional components. In general, components may be nested and as such an optimizer component may also be contained in a functional component.

After identifying and modeling the components, their interface must be modeled in terms of provided and required ports. To be able to function, components may require services from other components, but they may also require, or consume, resources; energy can be considered as a special kind of resource. Similarly, components may provide services and resources to other components. After these ports have been identified, the resource behavior of components has to be modeled and added as a novel kind of component port, which we call Resource Utilization Model (RUM).

The components resulting from this design process can be represented using the notation depicted in Figure 2 which we have proposed earlier [2] for modeling resource-aware components. A component’s RUM is represented as a state chart in which states are annotated with resource behavior, and invocations on the services of the component or internal events are modeled as transitions (for concrete examples of RUMs see Section 4). The RUM is represented as the light gray box inside the component and exposed through the octagonal port.

Modeling the RUM of components is a challenging task and the expressiveness of the RUMs determines the quality that can be achieved in the subsequent design process. It is the subject of our ongoing research to develop clear guidelines for modeling RUMs appropriately. As the main contribution of this paper, in Section 6, we focus on modeling RUMs for the existing identified
functional components, i.e., components which can be re-used in the design project at hand. Thus, of the activities comprising our design method presented in Figure 1, we mainly focus on those with a grey background.

Based on the modeled resource behavior of the components, our design method considers dedicated checking activities after each modeling step. Typical examples 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.

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’s results. In Section 6, we discuss (among others) the implications of modeling the RUM in the proposed way on the last two activities of our methodology, “Analyze system resource behavior” and “Select most suitable optimizer components”.

3. CEGAR

CEGAR is a way to—partially automatically—refine abstract models based on counterexamples, when a concrete model is available. The purpose of CEGAR is to increase the scalability of model checking by creating and using abstract models that contain the minimum amount of details needed for the desired model checks, instead of using the—generally much larger—concrete models for these checks. It has been implemented in several tools [11, 5], most relevantly in MAGIC for verifying systems composed from multiple parallel components [2, 3]. It has also been applied to events and aspects [6]. CEGAR is relevant when:

- Properties are being checked on abstract models; a violation of such a property leads to a counterexample
- A detailed concrete model is available, from which extra information can be added to the abstract model.

When a problem is detected in trying to prove a property using the abstract models, it could be due to either:

- Not including enough information about the concrete behavior in the abstract model (so the counterexample produced for the desired property in the abstract model does not correspond to an actual error in the concrete system and CEGAR can be used to refine the abstract model)
- Or it is a real error, and then an inventive step is needed, either changing the concrete design, or the specification. These can be guided by the counter-example, but are not automatic.

Using CEGAR requires (1) simulating the steps of the abstract counterexample to see whether they correspond to any execution of the concrete system. If not, (2) (minimal) information can automatically be extracted from the concrete model to make a refined abstract model in which the previous counterexample cannot occur, and then the model checking tool should again attempt to verify the desired property on the new abstract model.

Neither of the key steps in CEGAR is trivial, and they use sophisticated algorithms from model checking. The simulation requires showing that no concrete execution that corresponds to the abstract counterexample is possible, while extracting only the key information that makes the counterexample invalid in the concrete model involves techniques such as finding an interpolant of a complex boolean expression, and in particular, the core conjuncts that make the expression unsatisfiable.

4. Using CEGAR

As an example, we consider designing an application for playing music from the Internet. The application consists of three components: a Media Player, Optimizer, and Network Manager. This application is a concrete example, but our description is applicable to any system that consists of an application component, an optimizer and a low-level—or hardware—component. It can also be generalized to applications that consist of more than three components.

To design this media player, we must design a RUM—an abstraction—for each key property of every component. Once we have such RUMs, we can analyze whether this abstraction entails the desired behavior. If it does, we have—in some sense—defined a generic optimizer: it guarantees the key property for any concrete media player and concrete network manager as long as both satisfy the abstract version on which we performed the checks. However, we are still left with the question how to design these abstractions, which will be elaborated in the remainder of this section.

As noted above, our application has Media Player, Optimizer, and Network Manager components. We also assume that we already have a concrete model of the Network Manager, because this is a fixed low-level component that is already available. Presumably, the designers of the network manager analyzed its resource behavior and created a detailed model of the network manager, which contains at least all information that is relevant to our media player. This concrete model of the network manager might be the actual implementation itself together with the relevant energy consumption of this implementation, or a model that contains too many details to allow practical model checking of key properties. Since we did not design this model for the purpose of designing a media player or guaranteeing its properties, it does contain much more information than needed for this purpose. Therefore, it is not a minimal abstraction, and is not directly useful for the design of our application.

The identification of the minimal abstraction and the degree to which detailed energy information must be included in it is a challenge. Let us start with a very abstract model of the network manager, e.g. just a single state that restricts the power consumption below a value that is at least as high as the maximum power of the concrete implementation (an example is shown in Figure 3), and refine that to a RUM with just the information needed to justify the key properties of the system. This single state must be an over-abstraction of the component itself; that is, even though the model contains too little information to be useful, it must represent all correct behavior of the application.

For example, stating that the power consumption in this single state of Figure 3 is less than 1 J/s is fine, since the network manager never consumes more than 1 J/s. However, stating that the power consumption is larger than 0.5 J/s, for example, is simply incorrect, because in fact the network manager consumes less than that when it is idle.

When we have such an abstraction, we can use key properties to automatically refine this abstraction with the help of CEGAR to something more useful. Key properties are properties that should hold for every execution sequence. That is, any linear temporal logic (LTL) [11] formula can be used to specify a key property. In general, key properties are a combination of both safety and liveness properties. Examples of key properties are: uninterrupted operation (safety), a limited amount of energy consumed (safety),
and pressing the play button should eventually result in playing music (liveness). Note that without liveness properties, totally shutting down the system might be the best optimization: no energy is consumed and nothing can be interrupted. However, a media player that never plays any music is not very useful, therefore key properties are generally a mix of both safety and liveness properties. For the media player, we specify the key property: “in every execution sequence, the media player consumes less than 10 J for playing 20 s of music”.

With the current abstraction (shown in Figure 3), we can only show that for 20 s of music, less than 20 J is required. Thus, consuming 18 J during 20 s of playing music is a counterexample for the given key property, because it does not satisfy the key property and is a possible execution sequence according to the current abstraction. Therefore, to show that the property holds, the abstraction must be refined.

Assuming that the given property is indeed satisfied in the concrete model (because the network manager switches to idle when it is not downloading music) CEGAR can identify that this counterexample is spurious—it does not hold in the concrete model. CEGAR simulates all concrete events that correspond to a self-loop in the abstract state with a power consumption of 0.9 J/s for 20 s and discovers that no concrete execution corresponds to this abstract execution. Based on the concrete model, CEGAR can automatically create a refinement of the abstraction shown in Figure 3.

CEGAR, and in particular its implementation in the MAGIC tool, can identify the key information needed to show the abstract counterexample spurious for the concrete system (so no concrete execution sequence corresponds to it). Then, this information is used to build the refined abstraction shown in Figure 3 by adding more concrete information on the actual power needed during download, and splitting the abstract state into a downloading state, and an idle state with low power consumption when the request has been answered. The predicate done becomes true when the network senses that the request has been answered. This abstraction is detailed enough to show that for playing 20 s of music, less than 10 J is needed: so a new attempt to model check this property will succeed.

5. Related Work

Fleurey et al. [10] have specified a Domain Specific Modeling Language (DSML) for developing self-adapting software. In contrast to our approach, this DSML only models the adaptation logic; it is not a complete design approach.

Götz et al. [8] describe the Energy Auto-Tuning (EAT) approach for optimizing the energy consumption of software. Their approach uses newly designed languages, such as the Quality Contract Language (QCL), whereas we reuse existing languages for expressing properties of the system.

6. Conclusion and Discussion

In this paper, we have shown that by using the CEGAR approach, compact Resource Utilization Models (RUMs) can automatically be extracted from existing functional components. These compact RUMs help establish key properties relating to energy consumption. However, manual effort is still required when key properties cannot be proven, either to resolve an error in any of the components, or to add missing detailed information to the concrete model. Also, creating new optimizations requires an inventive step from domain experts. Thus, our approach does not invent new optimizations, but it can help domain experts to acquire the knowledge needed for designing optimizations.

For the media-player example, we see—once we have refined the abstraction—that the power states of the network manager coincide with its functionality. Based on that, a domain expert can formulate a possible optimization, for example: “downloading in bursts consumes less energy than downloading in trickles”. Such a property can already be checked on the model of the network manager and when it holds, it is an opportunity for optimizing the application.

It could also happen that the power states do not coincide with the functionality, maybe they are even arbitrary. In such a case, a domain expert might not find any opportunities for optimization.

References


