A Tool-Supported Approach for Modular Design of 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

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

ABSTRACT

The reduction of energy usage by software-controlled systems has many advantages, including prolonged battery life and reduction of greenhouse gas emissions. Thus, being able to implement energy optimization in software is essential. This requires a model of the energy utilization—or more general resource utilization—for each component in the system. Optimizer components, then, analyze resource utilization of other components in terms of such a model and adapt their behavior accordingly. We have devised a notation for Resource-Utilization Models (RUMs) that can be part of a component's application programming interface (API) to facilitate the modular implementation of optimizers. In this paper, we present tools for extracting such RUMs from components with an existing implementation.

Categories and Subject Descriptors

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

Keywords

energy-aware software, modularity, model checking, CEGAR, resource-utilization model, minimal abstraction

1. INTRODUCTION

There is an increasing demand for reducing the energy usage of systems; many of these systems are software systems, or use software to control their behavior. Energy is in fact one of many resources that may need to be managed by software, and reducing energy consumption cannot be considered without taking into account the trade-offs with other resources (e.g., memory and bandwidth usage) and services (e.g., delivered quality of audiovisual artefacts).

One possibility to reduce resource usage is to extend the core functionality with optimization logic, which approximates the state, and steers the behavior of a system. Our focus is on the reduction of energy consumption by controlling various external hardware components that are energy intensive. These hardware components are typically represented by dedicated software components, such as device drivers, and can in this way also be controlled by software.

As an example, consider a smart phone running a streaming media player application: a significant part of the energy will be consumed by the network component within the phone. The overall system's energy consumption can be optimized by steering the utilization of the network device so that it can spend more time in a low-power mode.

To facilitate modular implementation of resource optimization logic, we claim that there is a need for defining the resource behavior of components at their interface. For this purpose, we have proposed [22] to use so-called *Resource-Utilization Models* (RUMs), which express the relation between the dynamic behavior of the component and the resources it uses and provides. RUMs are abstractions, expressed as state transition diagrams expressing transitions between states of stable energy consumption. In particular, RUMs explicate internal events leading to state changes.

Considering the RUM as an interface for components also means that it hides implementation details; therefore, concrete implementations can be freely exchanged with others that adhere to the same RUM. This means that on one hand the RUMs of components must be sufficiently detailed to allow reasoning about the relevant resource-utilization properties. On the other hand, it must not be too specific such that exchanging concrete implementations is hindered.

In this paper, we present requirements for our RUM models to enable meaningful formal analyses of the resource consumption of software systems. In particular, we argue that for the purpose of reasoning about the effectiveness and correctness of optimizers, it is necessary to give guarantees about the resource utilization of components. Because defining RUMs can be complex and during software design, the implementation of third party components is often available already, we also present an approach for extracting a RUM from implemented components, based on the formal method of *counterexample-guided abstraction refinement* (CEGAR) [10]. Here we come to a (more) formal specification of how to apply the CEGAR approach, compared to the more intuitive version presented earlier in a workshop paper [4].

This paper is organized as follows. Section 2 outlines the approach and specifies which properties are most important

SAC'14 March 24-28, 2014, Gyeongju, Korea.

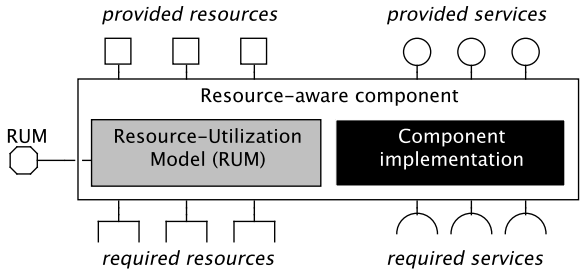

Figure 1: Resource-Aware Component Notation

to be checkable on RUMs and what the requirements for RUMs are. Section 3 explains how the tool MAGIC facilitates generating RUMs and gives an outlook how they can be used to model check system configurations. Section 4 discusses related work. Section 5 concludes this paper and presents future work.

2. DESIGN APPROACH FOR RESOURCE-AWARE COMPONENT-BASED SOFTWARE

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 [27, chapter 5]. To be able to design resource-aware software, components must also have explicit interfaces specifying the *resources* that it provides to and requires from its environment [22]. Additionally, resource-aware components declare their resource behavior, i.e., the dynamic relation between resources and services, in terms of a *Resource-Utilization Model (RUM)*. For example, RUMs 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, or the availability of resources and services in different situations. Figure 1 represents our notation of components.

Our notation for RUMs is based on state machines, because state machines (a) are declarative, (b) can model the internal behavior of a component and its relation with the services provided to and required from the environment, and (c) can conveniently be extended with resource-utilization annotations on states as well as state transitions.

Having a notation is not sufficient, we need to guide designers in the level of detail to be included in RUMs. Being a model at design time means that the RUM of a component must be an abstraction of the concrete implementation. This in turn means that the RUM only specifies implementation details which are relevant for reasoning about the resource utilization and leaves out irrelevant details. In our design approach, we propose to determine *relevant* details by specifying so-called *key properties*, which need to be established for a system. In the following subsection, we elaborate on the nature of key properties we need to express. Afterwards, we discuss what characteristics the RUM must have to support reasoning about models in the desired ways.

2.1 Identifying Key Properties

The key desirable properties, both those relating to the functionality of the system and those relating to resource consumption, can be derived from the requirement specification of the software. Properties of both kinds can be expressed in terms of the possible execution sequences of the system, and are commonly specified using a temporal logic notation [23, 12], although other notations can also be used, and we do not enter into the precise notation here. Which key properties can be checked automatically depends on the formalism in which the properties and the RUM are expressed. To identify which formalism we need, we must first identify which kinds of properties we want to check. We classify multiple kinds of properties into two dimensions:

First dimension: Which execution sequences should be considered? Properties can specify that they hold (1) for *every* execution sequence of the system or (2) for *some* (i.e., at least one) execution sequences.

Second dimension: Is the property (a) a *safety* property that should hold in every state of considered executions (and can be violated in a finite prefix of an execution sequence) or (b) a *liveness* property that ensures reaching desired states in the executions considered (and can only be violated in (usually infinite) execution sequences that do not reach the desired states).

The combinations of these two dimensions are enumerated in the following list, where Φ represents a state predicate.

1. For **every** execution sequence,
   (a) Φ always holds. (Safety)
   (b) Φ will eventually hold in the future. (Liveness)
2. For **some** execution sequence,
   (a) Φ always holds. (Safety)
   (b) Φ will eventually hold in the future. (Liveness)

For example, for a media player we can use the property "music is playing" in place of Φ:

1a. Throughout every system execution, music always plays in the media player.
1b. Pressing the play button always results eventually in playing music.
2a. There are system execution sequences where music is always playing.
2b. For some execution sequence, pressing the play button will eventually result in music playing.

Instead of only specifying functional behavior, properties relating to the amount of resource consumption can also be specified, using state assertions of the following types:

(i) Min/max. "The current resource consumption is within a certain range."
(ii) Total. "The total resource consumption is within a certain range." (This is the integral over the current resource consumption.)

Most critically, we would like to use verification tools to help find optimal execution sequences for a given sequence of user requests and system responses, or for a finite family of such inputs, i.e., (iii) "Provide the sequence for which the total resource consumption is optimal." Example properties are:

- The media player always uses less than 1 J/s (1a, i)

- For every execution sequence, whenever a full song is completed, the energy consumed since the song was requested is less than 10 J (1a, ii)
- For some execution sequence, the media player reaches a state after playing a full song consuming less than 10 J since the song was begun (2b, ii)

Standard temporal logic and model checking do not allow directly answering an optimization request of type (iii). However, the nature of an unsuccessful model check—that it provides a counterexample when the property does not hold for the model—can be exploited. For example, when an assertion of the form "For every execution sequence, whenever three minutes of music have been played, the energy consumption is greater than 10 J" is NOT true of the system, an attempt to model check shows a counterexample where the energy consumption is *less* than 10 J. In this way, an optimized computation (e.g., that buffers network data and works in bursts) can be detected. We intend to explore this possibility for *finding* optimizations in future work. In this paper, we will focus on analyzing systems and optimizations.

For an actual model or implementation, properties 1a, 1b, i, and ii are most important, because 1a and 1b provide guarantees for the model and can be combined with i and ii. During software design, properties 2a, 2b, i, ii, and iii might be quite useful, because 2a, 2b, and iii show what the possibilities of the actual model might be and 2a and 2b can be combined with i and ii. We focus on using actual implementations, so *in this paper we want to be able to verify safety and liveness properties that hold for all execution sequences.*

2.2 Modeling Resource Behavior

In component-based software development, service interfaces of a component are abstractions over the actual implementation of the component, and give guarantees about the functional requirements that are fulfilled by the component. Such guarantees facilitate using components without knowing their implementations, and enable flexibly changing the implementations of components, provided that the interfaces remain intact. Since we consider RUMs as part of the component interface, RUMs must also (1) provide abstraction over the implementation of the component, and (2) provide guarantees on the resource utilization of components.

2.2.1 Abstraction over component implementation

Since RUMs abstract over the implementation of a component, there are two main kinds of abstraction to consider:

Over-abstraction An over-abstraction (also called an over-approximation) specifies a superset of the possible execution sequences of a component. If we verify a property that holds for all execution sequences in the over-abstraction, then it also holds for the execution sequences in the actual concrete system. For example, an over-abstraction can be used to prove an upper bound of the maximum energy consumption of the actual system and a lower bound of the minimum energy consumption. Thus, over-abstractions provide guarantees of the component.

Under-abstraction An under-abstraction (also called an underapproximation) specifies a subset of the possible execution sequences of a component. If we verify a property for some execution sequence in the under-abstraction, then that computation is also in the actual system, and the property holds there too. For example, an under-abstraction can be used to find an execution sequence that gives a lower bound of the maximum energy consumption and an upper bound of the minimum energy consumption of the actual system. Thus, under-abstractions give possibilities of the component.

2.2.2 Guarantees on component resource utilization

The RUM of a component must provide guarantees about resource-utilization requirements of that component. As explained in Section 2.1, in our design method, the resource utilization requirements are formulated as key properties that hold for all execution sequences. Checking that these requirements hold can only be performed on RUMs that are over-abstractions; it would not be possible if RUMs were expressed as under-abstractions. This means *to fulfill most of the typical requirements, a RUM must be an over-abstraction.*

2.3 Analyzing Resource Behavior and Selecting Optimizer

When a software system is designed following the guidelines defined in this section—i.e., using a component model that specifies RUMs as over-abstractions of the components' behavior—the resource behavior of the composed system can be analyzed. The resource behavior of the overall system is determined by the composition of all the separate RUMs. Therefore, the designer can analyze the combined RUMs to understand the resource consumption of the system.

Because the resource utilization is specified for each component modularly, it is easily possible to analyze different compositions and especially to investigate the influence of alternative variants of single components. More specifically, it is possible to design multiple alternative optimizer components and use the analysis to identify which optimization results in the least resource consumption under various usage scenarios. Based on this analysis, designers can select a composition for the final system design using the most suitable one among alternative optimizer components.

As explained in the previous two subsections, it is most relevant to express guarantees of the resource behavior of components. The same argumentation holds for analyses of the composed system. Therefore, it has to be noted that the combination of the individual RUMs is an over-abstraction of the composed system, just like each single RUM is an over-abstraction of a single component. Thus, again, every property which we can prove to hold for every execution sequence in the combined RUMs, also holds for every execution of the concrete system. This means that we can perform an analysis of the properties of type 1a, 1b, i and ii as specified in Section 2.1, i.e., safety and liveness properties specifying ranges and totals of resource consumption for every execution.

3. TOOL SUPPORT

A software system is specified by a composition of components, which easily becomes complex, and beyond the capabilities of existing model checkers to handle directly. Thus it is necessary to create for each desired property an appropriate over-abstraction that is much smaller than the full model, and only includes the information directly needed to establish the property. Manual analysis of such a composition and finding the needed over-abstraction is difficult and

error-prone. Therefore, we claim that the analysis must be automated by means of tools.

This section explains how creating RUMs can be automated by using the tool MAGIC. However, our method does not dictate that this tool must be used. To allow use of other tools that support the same principle, we also explain the underlying mechanism CEGAR on which the tool is built.

As already seen, we consider a smart phone example running a streaming media player application: Significant energy will be consumed by the network component within the phone. The overall system's energy consumption can be optimized by steering the media player to stream the data in bursts operating at full bandwidth and store it in a buffer; when the buffer is filled, the network device can go to powersave mode until the buffer runs empty and another burst data transfer is necessary. For this optimization to be effective, e.g., it must be possible to predict at which rate the buffer is emptied depending on the playback rate, and it must be known how quickly the network device can switch between powersave and active modes, in addition to the amount of power consumed in both modes.

3.1 CEGAR

Counterexample-guided abstraction refinement (CEGAR) is a formal method to—semi-automatically—refine abstract models based on counterexamples, when a concrete model is available. It has been implemented in several tools [3, 7, 8, 11, 14]. In short, the application of CEGAR requires:

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

CEGAR requires that abstract models are over-abstractions of the concrete model. When CEGAR refines an abstract model with details from the concrete model, it ensures that the refined model is also an over-abstraction of the concrete model. Thus, CEGAR generates models that are suitable as RUMs.

Abstract models are refined as follows. First, an initial abstract model is derived, usually by simple static analysis of the concrete model, but guaranteeing that an over-abstraction is used. When CEGAR cannot prove a given property on the abstract model, an abstract counterexample is produced. This failure to prove the desired property could be due to two scenarios:

- There is *no* real error, but the abstract model does not include enough information about the concrete behavior. In this case, the abstract counterexample produced for the desired property does not correspond to an actual error in the concrete system (and it is called a *spurious* counterexample) and CEGAR can be used to automatically refine the abstract model.
- There *is* a real error, and then an inventive step is needed. An option is to weaken the specified desired property. But it may also be the case that an error in the implementation is found, which must be fixed by the responsible developer. This step can be guided by the counterexample, but is 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 tool should again attempt to verify the desired property on the refined abstract model.

The key steps in CEGAR use sophisticated algorithms from model checking. The simulation requires showing that no concrete execution that corresponds to the abstract counterexample is possible. At the same time the key information that makes the counterexample invalid in the concrete model must be extracted. This involves finding the core conjuncts that make a complex boolean expression unsatisfiable. This is today usually done by exploiting the progress made in solvers that determine whether a complex propositional formula with thousands of conjuncts is satisfiable (a SAT solver) or solving extensions to richer formulas known as SMT solvers. The formula constructed is an encoding of an assertion that the steps in the concrete model and steps in the abstract counterexample are possible together. When this is true, the abstract counterexample represents an actual error in the concrete system. When the check fails, the SAT (or SMT) solver provides a so-called *SAT-core expression* that has the fewest conjuncts showing why the formulas cannot be true together (see [11] for more details). From this, the minimal information that should be added to the over-abstraction can be automatically derived, without any user intervention. This gives a refined model for which the previous abstract counterexample cannot occur, and then a new attempt at model checking can be done on that model.

3.1.1 Using CEGAR for deriving RUMs

We experimented with several tools that implement CEGAR. Due to technical difficulties with some of the tools, we have extracted all models with MAGIC.

MAGIC [7, 8] is a tool for automatic verification of C programs against finite state machine (FSM) specifications. MAGIC follows the CEGAR paradigm and uses C source code as concrete model. First, MAGIC extracts an initial finite abstract model from C source code using predicate abstraction and theorem proving [9]. Subsequently, this model is refined until either it contains enough detail to show that the specification holds, or a real counterexample is discovered. MAGIC either outputs a success message or the found concrete counterexample.

MAGIC can output all models it creates, which are over-abstractions of the source code. The last model MAGIC creates in the success case contains sufficient information to prove the key property. Thus, we use this model as RUM. The size of these models created by MAGIC does not directly depend on the size of the source code, it only depends on the size of the code that is concerned with relevant properties. In our example, when reasoning about energy consumption is related to inactivity time, the size of the model depends on the number of locations where the inactivity timer is updated, not on how much data is written to the connection in between.

Figure 2 shows a state of a model generated by MAGIC. Such a state contains a set of predicates, which are all true while the system is in that state. MAGIC uses C code as notation, so the boolean value true is represented by the number 1 and the boolean value false by the number 0. The arrows represent the possible transitions between the states.

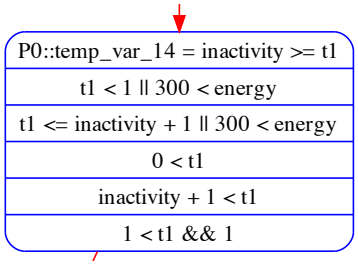

Figure 2: State of refined model generated by MAGIC

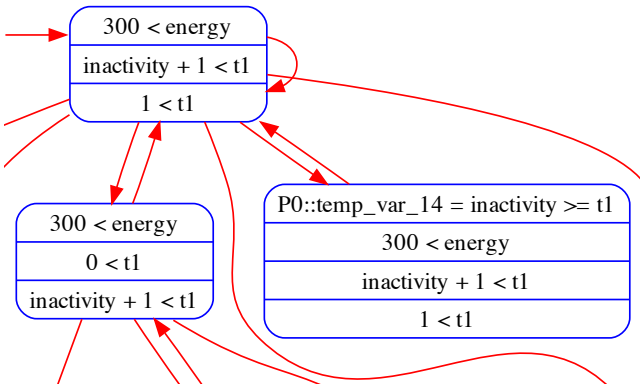

Figure 3: Excerpt of simplified over-abstraction

3.1.2 *Automatically simplifying derived RUMs*

Even though the models created by MAGIC only contain details that are related to the key properties, the models are larger than the size we envision for RUMs. This is mainly because MAGIC only wants to keep the models small enough to perform model checking, but we would ideally keep them small enough to be human readable, which is much smaller. Therefore, we post-process the models generated by MAGIC to reduce their size, using the transformations outlined in the following paragraphs.

First, we decide which variables we consider important. For example, when we want to know the relation between inactivity time t_1 and energy, the variables *inactivity*, *t1*, and *energy* are important. For every state, we remove all predicates that do not contain a variable of interest, to hide unimportant details of the system. This does not reduce the number of states, but—in general—creates many states that are similar because they have the same predicates.

Second, we simplify the predicates of each state. Using the CVC3 theorem prover [2] we identify both (1) predicates that are always true because they are implied by other predicates and (2) parts of disjunctions that are always false because they contradict other predicates, and remove these from the predicates. For example, the state shown in Figure 2 is equivalent to the rightmost state of Figure 3.

Third, we reduce the number of states by merging equivalent states. We do this by calculating the stutter bisimulation quotient [1]. Stutter equivalence considers state changes invisible if the truth value of the predicates does not change, that is, if two states have exactly the same predicates. Thus, the stutter bisimulation quotient is a smaller transition system, because it removes such invisible transitions.

3.1.3 *Application of CEGAR*

The media player application of our example consists of several components, among which is a Network Manager. This Network Manager is not designed specifically for the media player, but is a component that exists already before the media player is designed. Therefore, it is desirable to automatically create a RUM for the Network Manager, based on its source code.

To show that this is possible, we have written a simple network manager in C. The key property we used is that as long as the inactivity timer is less than t_1, the network manager consumes no more than 300 mA. In MAGIC, this key property K can be defined as follows:

```
K = (
  {inactivity = [$0 < t1 && energy <= 300]} -> K |
  {inactivity = [$0 >= t1]} -> K
).
```

Property K states that whenever *inactivity* changes to a value strictly less than t_1, the energy consumption must be less than 300 mA and afterwards K must hold again. If *inactivity* changes to a value greater than t_1, there is no requirement except that K must still hold.

MAGIC can indeed prove that this key property holds and provides us with an over-abstraction of 429 states. We simplify this over-abstraction with a script that performs the simplifications described in the previous subsection. This simplification outputs an over-abstraction of 60 states. Thus, a single key property results in a model that is reasonably small; nearly small enough to be human readable, and we are still investigating further simplifications.

3.2 Model Checking RUMs

After creating the RUMs of all components, the resource utilization of the whole software system is specified by the composition of these RUMs. This composition consists of many parallel state charts, which is too large to easily analyze by hand. Therefore, using a model checker is necessary.

It would be nice if creating RUMs and model checking them could be performed with the same tool, but unfortunately we have not found a tool that facilitates both. MAGIC focuses on automatically extracting models, but does not provide the ability to manually create models for newly designed components, such as the Media Player component. In a technical report [6], we show how UPPAAL can be used to provide additional analysis of RUMs. Figure 4 shows the energy consumptions of three different controllers of a media player, which were analyzed in UPPAAL. This shows that UPPAAL indeed facilitates analyzing energy consumption based on RUMs. However, the RUMs used for this analysis were created manually; converting RUMs extracted by MAGIC to UPPAAL is still ongoing work.

4. 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 [30], at the level of compilers [13], or at the system level [28]. Recently, more emphasis is put on modeling and analyzing energy-consumption at the application level.

The need for reflecting energy consumption of software in design models is also studied by Sahin et al. [26]. However,

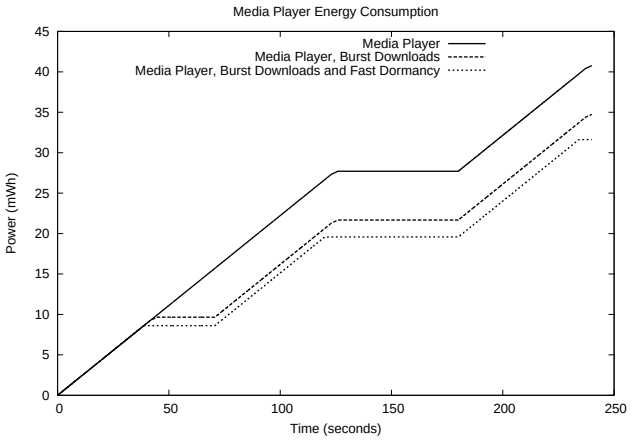

Figure 4: Analysis of scenario

in contrast to our approach, no concrete method or tools are provided to model resource utilization of components, to (automatically) derive RUMs and to analyze them.

Götz et al. [19] propose a model-driven component-based approach for software systems that can be optimized w.r.t. their provided quality and energy consumption at runtime. In this approach, operational modes of components and the resource utilization in each mode are specified as a state machine at the interface level. The optimization problem is defined as an integer linear program (ILP); an ILP solver determines an optimized configuration of software. We identify two difficulties in adopting this approach. First, manual identification of components' resource behavior can be error-prone, and second, this approach assumes that the implementation of components fulfills their specified resource utilization, where this may not be the case in reality. We overcome these by automating the derivation of RUMs from concrete models (e.g. implementations) by means of key properties. Since key properties are checked in the models during the derivation process, it is possible to identify that the concrete models fulfill the desired resource utilization.

Deriving detailed information about the energy modes of applications and the amount of energy consumption in each mode is a challenging task. In our approach, we assumed that information about energy/resource consumption is available in the implementation as annotations over which we can evaluate key properties. Several approaches, discussed below, investigate tools to obtain energy models of software. The accuracy of models depends on the granularity of instrumentations to obtain energy consumption information. Hähnel et al. [20] utilizes Running Average Power Limit (RAPL) energy sensors available in recent Intel CPUs to measure the energy consumption of short code paths (e.g. individual functions). Powerscope [17] facilitates profiling the total energy consumed during a certain time period in each process and/or procedure. Eprof [25, 24] is a fine-grained energy profiler for smart-phone apps, which instruments the app source code for tracing the energy consumption of system calls and application calls. As a result of tracing, Eprof generates a finite state machine depicting various energy modes and energy consumption of apps.

The above-mentioned approaches profile the energy consumption for a subset of the behavior of applications, i.e. the behavior that is executed. Hence, the derived energy models are under-abstractions, and as we discussed in Section 2.2, by adopting under abstractions we cannot provide guarantees over the overall resource utilization of the applications. Therefore, these approaches can be regarded as complementary to ours; we can use the energy models resulting from these approaches to annotate RUMs with more accurate information about the energy consumption of each provided/required service of components.

The approach *eLens* [21] combines per-instruction modeling with program analysis to create fine-grained estimates of energy consumption. However, their approach does not consider abstractions or existence of various power states. Therefore, our approach is more suitable for large software systems, but the fine-grained estimates of *eLens* might be usable as energy input for our approach.

Extending software with energy optimization functionality is a typical way to make software energy adaptive [16, 19, 15]. In contrast to our approach, these approaches fix the optimization functionality, and do not provide means to analyze the impact of various optimizers on the overall energy consumption of software to guide designers to choose suitable optimizers accordingly.

5. CONCLUSION AND FUTURE WORK

In this paper, we have shown how (formal) tool support for defining RUMs can be provided. For giving guarantees, RUMs must be over-abstractions of the actual behavior, so that model checkers can take all possible execution traces into account. We have presented how to use the tool MAGIC and post-process its output to automatically extract RUMs from existing component implementations.

Our approach generates over-abstractions by formally analyzing source code, which is an advantage over testing-based approaches. This is because such approaches can only consider a subset of the possible behavior of software. Thus they cannot give guarantees as we do.

Our ongoing work is completing the full integration of abstract models from MAGIC into Uppaal. We already have intermediate results in transforming the models, mainly lacking a proper transformation of the timing information from the MAGIC to the Uppaal formalism.

In future work, we want to adopt energy profilers to annotate RUMs with more accurate information about the energy consumption of each state. For this purpose, we will investigate combining our approach based on formal analysis with test-based approaches, which physically measure the energy consumption such as JouleUnit [29] or Eprof [25, 24].

The approach presented in this paper focuses on the extraction of RUMs from components with existing implementations. We want to research integrating this approach into a full design method which also includes creating RUMs for newly designed components (i.e., without existing implementation). A preliminary study of this is discussed in previous work [5]. The main challenge for integration is guaranteeing correctness. Since, in general, checking whether a model is an over-abstraction is unfeasible, models created by CEGAR are over-abstractions by construction, whereas manually designed models provide no such guarantee.

We furthermore intend to develop a library of RUMs derived from implementations of standard components, such as fixed network implementations, and power considerations, in order to allow their reuse as various optimization strategies

for different applications are analyzed. Finally, we want to explore the possibility to discover optimization opportunities using model checkers, as outlined in Section 2.1.

6. REFERENCES

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press Cambridge, 2008.

[2] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proc.* 19^{th} *Int. Conf. on Comput. Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[3] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5), 2007.

[4] S. te Brinke, C. Bockisch, L. Bergmans, S. Malakuti, M. Akşit, and S. Katz. Deriving minimal models for resource utilization. In GIBSE '13 [18], pages 15–18.

[5] S. te Brinke, S. Malakuti, C. M. Bockisch, L. M. J. Bergmans, and M. Akşit. A design method for modular energy-aware software. In *Proc. ACM Sympos. Appl. Comput.* ACM, 2013.

[6] S. te Brinke, S. Malakuti Khah Olun Abadi, 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, 2012.

[7] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *Trans. Softw. Eng. (TSE)*, 30(6):388–402, June 2004.

[8] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in Syst. Des. (FMSD)*, 25(2-3):129–166, 2004.

[9] S. Chaki, E. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In D. Geist and E. Tronci, editors, *Correct Hardware Des. and Verification Methods*, volume 2860 of *LNCS*, pages 19–34. Springer Berlin Heidelberg, 2003.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. Emerson and A. Sistla, editors, *Comput. Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer Berlin Heidelberg, 2000.

[11] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS, volume 3440 of LNCS*. Springer, 2005.

[12] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr. 1986.

[13] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. *SIGPLAN Not.*, 47(10):831–850, Oct. 2012.

[14] C. Disenfeld and S. Katz. Specification and verification of event detectors and responses. In *Proc. Int. Conf. on Aspect-Oriented Softw. Devel. (AOSD)*, 2013.

[15] Y. Fei, L. Zhong, and N. K. Jha. An energy-aware framework for dynamic software management in mobile computing systems. *ACM Trans. Embed. Comput. Syst.*, 7(3):27:1–27:31, May 2008.

[16] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc.* 17^{th} *ACM Sympos. on Operating Syst. Principles*, SOSP '99, pages 48–63, New York, NY, USA, 1999. ACM.

[17] J. Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Proc.* 2^{nd} *IEEE Workshop on Mobile Comput. Syst. and Appl.*, pages 2–10, 1999.

[18] *Proc. 2013 Workshop on Green in/by Softw. Eng.*, GIBSE '13, New York, NY, USA, Mar. 2013. ACM.

[19] S. Gotz, C. Wilke, S. Cech, and U. Assmann. Architecture and mechanisms for energy auto tuning. In *Proc. Sustainable ICTs and Manag. Syst. for Green Comput.*, 2012.

[20] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, Jan. 2012.

[21] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proc. 2013 Int. Conf. Softw. Eng.*, ICSE '13, pages 92–101, Piscataway, NJ, USA, 2013. IEEE Press.

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

[23] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., 1992.

[24] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof. In *Proc.* 7^{th} *ACM Eur. Conf. Comput. Syst.*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[25] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proc.* 6^{th} *Conf. Comput. Syst.*, EuroSys '11, pages 153–168, New York, NY, USA, 2011. ACM.

[26] C. Sahin, F. Cayci, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. Towards power reduction through improved software design. In *Energytech, 2012 IEEE*, pages 1–6, 2012.

[27] C. Szyperski. *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

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

[29] C. Wilke, S. Götz, and S. Richly. JouleUnit: a generic framework for software energy profiling and testing. In GIBSE '13 [18], pages 9–14.

[30] 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.