Student Research Abstract: Interpreting Energy Profiles with CEGAR

Steven te Brinke
University of Twente – Formal Methods and Tools group – Enschede, The Netherlands
brinkes@cs.utwente.nl

1. INTRODUCTION

There is an increasing demand for reducing the energy consumption of systems that are controlled by software. Energy is one of the resources that should be reduced, but since software often consumes higher-level resources which indirectly consume energy, it is important to model not only energy, but resource consumption in general. To facilitate modular implementation of resource optimization logic, we have proposed [1] 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 as state transition diagrams expressing transitions—triggered by either service invocations or internal events—between states of stable resource consumption. We have shown how to use the CEGAR approach to automatically extract RUMs from existing component implementations.

However, this approach does not measure any energy consumption; it assumes that energy information is available already, e.g.: as annotations in the source code or defined by the specification. Whereas this assumption holds in some cases, it is not applicable in general: Software libraries usually lack energy information. Therefore, to optimize energy consumption effectively, it is necessary that the energy consumption of such libraries can be profiled, so as to add energy information to the RUM.

2. EXTRACTING ENERGY INFORMATION

Our current approach uses CEGAR to extract RUMs from source code, but energy information cannot be derived from only source code, unless it is manually annotated with energy information. We would like to automatically add energy information to RUMs by energy profiling specified execution scenarios. In this research, we will explore to which degree we can use such energy profiles while automatically generating a RUM with the CEGAR approach.

2.1 JouleUnit

JouleUnit [2] is a framework for profiling the energy consumption of Android applications in order to generate energy labels. To provide such labels, the framework supports writing system tests that execute a scenario and measure the power consumption of the full device. Such measurements are performed by special hardware—e.g.: an oscilloscope—after which the framework synchronizes the measured samples with events generated through JouleUnit API calls. Such special hardware is required because energy measurements made by the smart phone itself are too coarse grained, since generally many events are generated per second. The special hardware provides measurements that are fine grained enough to synchronize the measured samples with such events.

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.
SAC'14 March 24-28, 2014, Gyeongju, Korea.
Copyright 2014 ACM 978-1-4503-2469-4/14/03 ...$15.00.

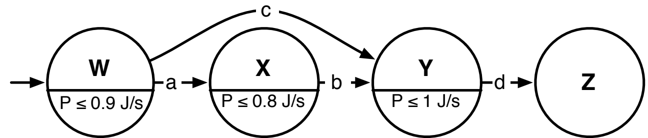


Figure 1: Example behavior

2.2 CEGAR

Counterexample-guided abstraction refinement (CEGAR) is a formal method to refine abstract models when a concrete model is available. First, an initial abstract model is derived, usually by static analysis of the source code. When model checking cannot prove a given property on the abstract model, a counterexample is produced and simulated on the concrete model, which may have two outcomes:

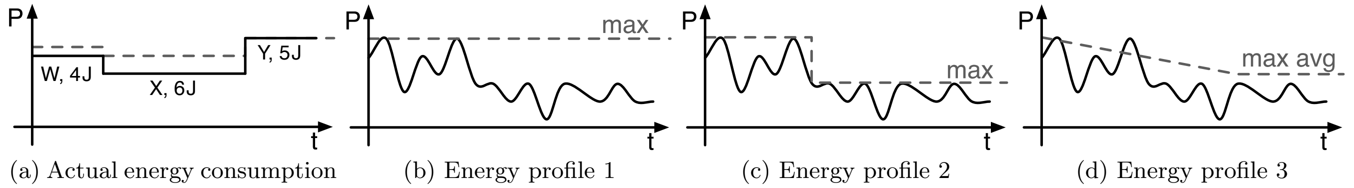
- There is *no* real error, but the abstract model does not include enough information about the concrete behavior; it is called a *spurious* counterexample.
- There *is* a real error. Then an inventive step is needed, because either the application misbehaves or, more likely, the property is a too strict requirement.

In the first case, information can automatically be extracted from the concrete model to make a refined abstract model in which the previous counterexample cannot occur, and then this process must be repeated until the desired property can be proven on the refined abstract model.

In the second case, when we discover that the property is too strict, we should relax the requirements on energy consumption. This can be guided by the counterexample, but is not automatic.

2.3 Generating Test Code

Consider the state chart in Figure 1 the behavior (i.e.: RUM) of an application, on which we run our approach. Deriving this RUM started with specifying a key property, which could be: *reaching Z always consumes less than 16 J*. This key property cannot be guaranteed. Therefore, our approach will give a counterexample, which could be the following sequence that consumes 17.5 J (denoted by which



(a) Actual energy consumption (b) Energy profile 1 (c) Energy profile 2 (d) Energy profile 3

Figure 2: Example energy profiles

states are visited, how long the application stays in these states, and how much energy is consumed in each state): W, 5 s, 4.5 J → X, 10 s, 8 J → Y, 5 s, 5 J → Z.

We must validate that this counterexample is spurious and derive all necessary information missing in the RUM from the application, for which we leverage JouleUnit. From the state chart, we see which services must be invoked to visit the states of the counterexample. Assuming that we have control over executing services a, b, and d, a JouleUnit test case that stays in each state as long as given by the counterexample can be generated by executing these services at the right times. Executing this test shows us whether both the timing and the energy consumption of the counterexample are realistic. Figure 2a depicts a possible output of JouleUnit for this example: It shows the measured energy consumption over time. From this figure we can conclude that in reality, the counterexample did not occur, because both states W and X consume less energy than possible according to the RUM in Figure 1. Thus, the counterexample is spurious and the RUM is refined with the actual energy consumption. On the refined RUM, the above steps are applied again iteratively, which shows that the key property holds.

2.4 Extracting Energy Profiles

The previous subsection gave an example of energy consumption (Figure 2a) that is static during every state. In reality, we expect the energy consumption to be more dynamic, for example as shown by the other graphs in Figure 2. The more precisely we can extract the energy profile, the more useful the result will be. Extracting a precise profile may require correlating changes in energy consumption to execution events, such as service invocation. The *max* lines in Figure 2 show three possible energy profiles which could be extracted from the same measurement.

Figure 2b shows the simplest profile: the energy consumption is always below a constant value. However, this value is often much higher than the actual energy consumption, so it is not a very precise profile.

In Figure 2c the maximum is reduced at some moment. This creates a more precise profile, but requires knowledge about when the maximum energy consumption changes (e.g.: after a certain time or after invocation of a specific service).

The profile shown in Figure 2d is less precise than Figure 2c, but more precise than Figure 2b. Instead of providing a maximum for the consumption at any moment, it provides a maximum for the average energy consumption. Since, in general, we are interested in the total energy consumption, having guarantees on the average consumption is sufficient.

3. CHALLENGES

In this research, we use JouleUnit in an original way, for which it was not specifically intended. This leads to the following challenges:

1. JouleUnit provides system tests, but we want to profile the energy of single components.
2. Generating test code that controls the execution to follow the states from the counterexample is not trivial.
3. JouleUnit only provides outcomes of test cases; it cannot provide guarantees.

Since JouleUnit only provides system tests (Challenge 1), it does not directly provide fine-grained control for extracting the resource consumption of single components, when several components are executing in parallel. Therefore, test cases must be constructed in such a way that either only single components are executed, or the energy consumption of other components should have been measured already, such that the test results can be corrected for possible interference by these components.

For generating test code (Challenge 2), we need to identify which API calls force which state changes and how long such state changes take, so as to issue the calls that generate the desired behavior.

Because the CEGAR approach is based on guarantees, treating the results of test cases in this approach is challenging (Challenge 3). Possible solutions are using a probabilistic model or restricting ourself to boundaries within which we can give guarantees. We could, for example, execute test cases multiple times to provide sufficient certainty about the energy profile. If the test is playing a song, then playing 30 different songs can be used to find an energy profile that holds for all these songs. This may provide enough confidence to conclude that a counterexample is spurious. However, identifying sufficient diverse inputs is also challenging. For example, using 30 different songs provides a better energy profile than using 30 times the same song.

4. EXPECTED RESULTS

With the approach described in this paper, we expect to be able to generate RUMs from source code, augmented with the energy profiles extracted by JouleUnit. Such RUMs should provide enough energy information to allow modular specifying energy optimizations. To validate this, we will write energy optimizations based on the generated RUMs and verify that these optimizations indeed reduce the overall energy consumption of the system.

5. REFERENCES

[1] S. te Brinke, S. Malakuti, C. M. Bockisch, L. M. J. Bergmans, M. Akşit, and S. Katz. A tool-supported approach for modular design of energy-aware software. In *SAC '14*. ACM, Mar. 2014.

[2] C. Wilke, S. Götz, and S. Richly. JouleUnit: a generic framework for software energy profiling and testing. In *GIBSE '13*, pages 9–14. ACM, Mar. 2013.