UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

DANILO DA SILVA ALVES

Using Model-based Analysis to Improve Software Energy Efficiency

Master Thesis presented to the Postgraduate Program in Computer Science of the Federal University of Rio Grande do Sul as a partial requirement to obtain the degree of Master in Computer Science

Advisor: Prof. Dr. Lucio Mauro Duarte

Porto Alegre
2022

CIP — CATALOGING-IN-PUBLICATION

Alves, Danilo da Silva

Using Model-based Analysis to Improve Software Energy Efficiency / Danilo da Silva Alves. – Porto Alegre: PPGC da UFRGS, 2022.

62 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2022. Advisor: Lucio Mauro Duarte.

1. Model-based Analysis. 2. Software Energy Consumption. 3. Software behaviour. I. Duarte, Lucio Mauro, orient. II. Título.

To my friends and family. I am nothing without them.

Acknowledgements

To my mother and sisters, who spared no effort to support me.

To my advisor Lucio Duarte, who supported me at all times and helped me to achieve this achievement.

To my friend Oseias, who helped me to consolidate this work and provided me with a sincere friendship.

"It is the responsibility of scientists never to suppress knowledge, no matter how awkward that knowledge is, no matter how it may bother those in power; we are not smart enough to decide which pieces of knowledge are permissible and which are not."

Carl Sagan

Abstract

Recently, energy consumption has become a concern in the software development scenario. This is mainly due to the current different types of platforms where software runs. Studies show that users tend to reject mobile applications that quickly drain battery energy, indicating energy consumption as a relevant aspect. Despite this emerging interest in the software energy consumption metric, developers still lack solid knowledge about how to design, construct and evolve software considering energy efficiency. With the goal of providing some of this necessary support, this work proposes high-level property definitions for the analysis of software energy consumption during all phases of software development. These properties and their analyses rely on a model-based approach, which uses Labelled Transition Systems (LTS) augmented with energy costs and probabilistic information to describe software behaviour. Results of analysing the proposed properties on an LTS model provide useful energy-related information, such as the average energy cost of the system and the probability of occurrence of the most costly execution. We demonstrate how to implement, use and interpret the results of analyses of these properties to create, evaluate and/or evolve software considering energy efficiency. To describe where our work can be applied, we discuss experiments involving the analysis of the proposed properties in different scenarios. Some experiments were performed, involving an analysis of an existent software with a single component, a software evolution and a software with composition of components, and energy efficiency was analysed in all cases. Recommendation of possible actions to adjust energy consumption considering results of property analysis are proposed in a quick guide format, combining energy costs and probabilistic behaviour. This support of property analysis and recommendations constitutes an important step towards helping developers create energy-efficient software.

Keywords: Model-based Analysis, Software Energy Consumption, Software behaviour

Resumo

Recentemente, o consumo de energia tornou-se uma preocupação no cenário de desenvolvimento de software. Isso se deve principalmente aos diferentes tipos de plataformas atuais em que o software é executado. Estudos mostram que os usuários tendem a rejeitar aplicativos móveis que esgotam rapidamente a energia da bateria, apontando o consumo de energia como um aspecto relevante. Apesar desse interesse emergente na métrica de consumo de energia de software, os desenvolvedores ainda carecem de conhecimentos sólidos sobre como projetar, construir e evoluir software considerando a eficiência energética. Com o objetivo de fornecer algum suporte necessário, este trabalho apresenta definições em alto nível de propriedades para a análise do consumo de energia de software durante todas as fases de desenvolvimento de software. Essas propriedades e suas análises dependem de uma abordagem baseada em modelos, que usa *Labelled Transition Systems (LTS)* com o acréscimo de custos de energia e informações probabilísticas para descrever o comportamento do software. Os resultados da análise das propriedades propostas em um modelo LTS fornecem informações úteis relacionadas à energia, como o custo médio de energia do sistema e a probabilidade de ocorrência da execução mais cara. Demonstramos como implementar, usar e interpretar os resultados das análises dessas propriedades para criar, avaliar e/ou evoluir softwares considerando eficiência energética. Para descrever onde nosso trabalho pode ser aplicado, discutimos experimentos envolvendo a análise das propriedades propostas em diferentes cenários. Alguns experimentos são realizados envolvendo uma análise de um software existente de um único componente, uma evolução de software e um software com composição de componentes, e a eficiência energética é analisada em todos os casos. Considerando os resultados análises das propriedades definidas, são propostas recomendações de possíveis ações para ajustar o consumo de energia em um formato de guia rápido, combinando custos de energia e comportamento probabilístico. Esse suporte de análise de propriedades e recomendações constitui um passo importante para ajudar os desenvolvedores a criarem software com eficiência energética.

List of Figures

List of Tables

Contents

1 Introduction

Software is present in all types of devices and platforms and some programs must run with a limited amount of energy due to restrictions imposed by these environments. Research has shown mobile applications that quickly drain battery energy tend to be rejected by users (Khalid; Shihab; Nagappan; Hassan, 2015), indicating that energy consumption is a relevant aspect from the users perspective. Corporations have also come to the conclusion that small inefficiencies in software can significantly affect its operation (Pinto; Castor, 2017a). For this reason, software energy consumption has become an important factor during software development, maintenance and evolution (Albers, 2010) (Li et al., 2016) (Singh; Naik; Mahinthan, 2015a) (Singh; Dutta; VanderMeer, 2013).

It is possible to assume energy will continue to increase its importance when comes to software, becoming as important as it already is in hardware design (Zhang; Sadler; Lyon; Martonosi, 2004) (Wheeldon et al., 2020). As a consequence, we might soon start comparing programs in terms of energy complexity in the same way we now do with time and space. As systems grow in scale and complexity, mostly composed by multiple components that are geographically spread and rely on power supply to keep running, reducing energy consumption might become one of the most important aspects of software design.

Despite the current relevance of energy consumption analyses, there is still little support for designing energy-efficient software. In fact, developers find it still unclear how to produce, evaluate and evolve their software considering energy costs (Pinto; Castor, 2017a) (Pang; Hindle; Adams; Hassan, 2015) (Manotas; Pollock; Clause, 2014) (Pinto; Castor, 2017b). This is mainly due to the absence of combined abstractions and tools to model, measure and analyse energy consumption(Duarte; Alves; Toresan; Maia; Silva, 2019). Hence, finding techniques, tools and processes that could help software developers better understand software energy costs and how to use energy resources has become a major concern (Pinto; Castor, 2017a). With such support, developers would not only be able to identify the costs of executing their software, but also perform analyses in the design phase in order to predict possible inefficiencies, compare different versions in terms of energy consumption and determine possible changes to improve energy efficiency.

There are some available tools to collect software energy consumption such as Gem5 (Binkert et al., 2011) and McPAT (Li et al., 2009), or jRAPL (Liu; Pinto; Liu, 2015) and pyRAPL (Belgaid; d'Azémar; Fieni; Rouvoy, 2019). Whereas McPAT and Gem5 use abstractions to obtain estimates of energy information, simulating an execution, jRAPL and pyRAPL allow the annotation of source code with methods to collect energy

information. Although these tools help the developers measure the energy spent by their software, there is still the need of interpreting what this information means and how to use it.

Research such as presented in (Pereira; Couto; Saraiva; Cunha; Fernandes, 2016) is an example of work on energy consumption analysis. They focused on finding excessive or anomalous energy consumption in software and used a methodology to optimise Java programs and decrease their energy consumption by replacing some data structures for their more energy-efficient alternatives. Although these approaches offer relevant information about the software energetic behaviour, they do not provide any feedback, analysis or guideline on how to improve software energy-efficiency, leaving this task to the developer.

A way to perform energy analysis is using models to obtain an abstract representation of the energy behaviour. In this context, energy analyses could be performed using a model checking tool, such as PRISM (Kwiatkowska; Norman; Parker, 2001) or LoTuS (Barbosa; Lima; Maia; Junior, 2017). LoTuS[1] allows the construction of Labelled Transition Systems (LTS) (Keller, 1976) with energy and probability annotation using an intuitive graphical user interface. However, there is no support for analyses on energy behaviour of the model. As described in (Baier; Dubslaff; Klein; Klüppelholz; Wunderlich, 2014) (Dubslaff; Klüppelholz; Baier, 2014), where a Markov chain can be used to model software considering the probabilistic or stochastic behaviour, transition and state costs could be used to model energy costs, and information about accumulated energy can be obtained using a probabilistic temporal logic. Nevertheless, the user still needs to collect the energy information to be used in the model, and some relevant questions cannot be asked, such as the behaviour that produces the highest energy consumption or the average cost of an execution.

Although there are some tools that are able to analyse software energetic behaviour, it is a difficult task for the developer to determine which information is more relevant and how to describe it in the form of properties to be analysed. Moreover, even after describing and running the analysis of these properties, the developer needs to interpret the results to identify what action - if any - is required. In this work, we propose a set of high-level property definitions for model-based software energy consumption analyses. We provide a guideline on how to perform the analyses and interpret their results such that the software developer can understand how to produce or evolve their software to improve energy-efficiency and we also elaborate a quick guide, in a "Frequently Asked Questions"(FAQ) format, considering different scenarios and suggesting some property analyses to perform in each one. To analyse software using these properties, we model it using LTS, which has a graph-like structure, augmented with information about energy cost and probability of execution of software elements. These proposed properties are

[1] Available in: http://lotus-web.herokuapp.com/

divided into two groups: the first group includes the properties used to perform analyses about energy costs alone, whereas the second group combines analyses of energy costs with probabilistic information to give a more informative scenario about the software behaviour in terms of energy consumption. Defining these two groups is necessary because the probability information is not always available or may not be accurate. Moreover, some times, the necessary information does not require probabilities, such as identifying the behaviour of a system with the highest energy cost.

Considering the results of this work, we can identify 3 possible scenarios of usage:

1. **Software Project Analysis:** With the proposed analyses, a software developer could, for instance, estimate the energy efficiency of a software based only on a blueprint, constructing a behaviour model and collecting energy costs and probability information. This information can come from a specialist, be based on previous studies, follow from previous user's experience or considering available specifications. Based on the analyses, in this scenario, a developer could, for example, verify the adequacy of a software energy consumption to limits of a platform where the software will run;

2. **Existing Software Analysis:** Other possible scenario is the evaluation of an existing software to measure how much energy is spent during its execution and which parts have high consumption. This information allows the user to verify the necessity of changes to reduce energy consumption and to determine which strategies can be employed to achieve a more energy-efficient software;

3. **Software Evolution Analysis:** Finally, these property definitions can be used to verify the positive or negative impacts of software evolution giving to the developer a more informative scenario about the results of code changes in terms of energy consumption. This information can provide support to predict the impacts caused by, for example, the inclusion or removal of software features, components or functionalities or the consequence of applying some refactoring or replacement of data structures.

The experiments performed in this work consists of (i) an evaluation of an implementation of the SMTP protocol, (ii) the analysis of a matrix multiplication system extracted from (Duarte; Alves; Toresan; Maia; Silva, 2019) and (Alves; Ferreira; Duarte; Maia, 2020) and (iii) the analysis of the impact of the combination of a file compression component with an implementation of the FTP protocol. The analysis of the SMTP protocol energy consumption helps understand the application of property analyses in a existing single-component system, the experiment performed with the matrix multiplication system help us observe how to analyse a single-component system evolution, and the analysis of

the network file transfer system combining file compression and the FTP protocol shows important questions about the application of this approach to multiple-component systems. In our experiments, we decided to apply this approach only to existing systems. Although, the approach is applicable to systems in the project stage of the software development, we chose to work with systems for which we could collect energy and probability information, using real values in the models to be analysed.

The main contributions of this work are:

- The presentation of high-level property definitions including energy and/or probability information to obtain a model-based analysis able to identify possible problems in software energy behaviour;
- The demonstration of how to interpret these properties and the results of their analysis;
- The implementation of analyses of some of the proposed properties, demonstrating how the analysis of these properties can be conducted automatically;
- The construction of a quick guide that indicates common questions about software energy behaviour and which property analyses could be used to obtain answers to these questions.

The remainder of this document is divided as follows: Section 2 present some information necessary to comprehend this work; Section 3 discusses the main related work; Section 4 introduces the proposed property definitions, how to interpret the results of their analyses and a quick guide with possible questions for some situations, while Section 5 presents initial results of using the proposed approach in a few experiments; finally, in Section 7 we present our conclusions and outline future research directions.

2 Background

In this section, we present some basic ideas related to energy consumption analysis. Firstly, we discuss how software energy information can be collected and, then, how to use this obtained information in a behaviour model so that analyses can be carried out using this model. We also discuss how to model probability information.

2.1 Collecting Energy Information

The first step to make any analysis about software energy costs is to collect the necessary information. Collecting such information refers to executing the software using some method to measure energy costs of (part of) a system during its execution. Energy values can also be provided by software analysts based on their experience or be given by software/hardware specifications.

There are some available tools able to collect energy information about performed operations, which allow the measurement of energy costs associated to code locations. Examples of such tools are jRAPL (Liu; Pinto; Liu, 2015), which is a framework for profiling energy consumption of Java programs that identifies how much power a code segment is spending, Gem5 (Binkert et al., 2011), which is an architecture simulator able to measure the energy cost of a running software, McPAT (Li et al., 2009), which is a framework for simulating energy consumption and pyRAPL (Belgaid; d'Azémar; Fieni; Rouvoy, 2019), which uses similar concepts to those of jRAPL, but applied to Python programs. While jRAPL and pyRAPL can collect the energy consumption of selected parts of a software, Gem5 and McPAT collect only the global cost of running a program. However, jRAPL only works with some types of architectures and just with Java programs. In (Georgiou; Rizou; Spinellis, 2019), the authors present a detailed comparison about energy collection tools (including those presented here).

Regardless of the tool, collecting energy consumption information involves either running some simulations or executing the program a number of times to obtain a more precise cost information. Moreover, developers still need to understand how such tools can be used to increase energy efficiency and how the identified hotspots affect the overall energy consumption of their software.

2.2 Energy Consumption Modelling and Analysis

Having energy consumption information, it can be inserted in a model to be analysed in some available tool. A widely used model is *Labelled Transition Systems (LTS)* (Keller, 1976). In an LTS, the system behaviour is described by the sequences of actions that it can execute, where actions are defined according to the level of abstraction involved and can represent method calls, variable assignments, task completion, or some other significant event. LTS models are similar to state machines, which makes it easier for any developer to intuitively construct/understand them.

An LTS $M = (S, s_i, \Sigma, T)$ is formally defined as a model where:

- S is a finite set of states,
- $s_i \in S$ represents the initial state,
- Σ is an alphabet (set of action names), and
- $T \subseteq S \times \Sigma \times S$ is a transition relation.

Transitions are labelled with the names of actions from the alphabet that trigger a change from an origin state to a destination state. Therefore, given two states $s_0, s_1 \in S$ and an action $a \in \Sigma$, then a transition $s_0 \xrightarrow{a} s_1$ means that it is possible to go from state s_0 to state s_1 executing action a. A *behaviour* of an LTS M is then a finite sequence of actions $\pi = \langle a_1, ..., a_n \rangle$ such that $a_1, ..., a_n \in \Sigma$. The set $L(M) = \{\pi_1, \pi_2, ...\}$ of all behaviours of M is called its *language*.

Next, we present how an LTS can be used to model both the energy behaviour of a system and probabilistic information about this system.

2.2.1 Modelling Energy Information

In (Duarte; Alves; Toresan; Maia; Silva, 2019), the authors proposed a model-based framework for analysing software energy consumption through the verification of energy-based properties, thus providing a novel approach for the development and evolution of energy-efficient software. They use an LTS to represent the system behaviour and its associated energy costs. To model energy costs, they add cost values to transition labels, thus associating a cost to each action. Hence, a behaviour in this energy-labelled model is a sequence of code elements and their respective costs, where the total cost of a behaviour is the sum of the costs of each element composing this behaviour. As an LTS is a graph-like structure, graph-based algorithms can be used to calculate accumulated costs of behaviours and determine the most/least costly behaviour (i.e., software behaviour). The authors have extended an LTS-analysis tool called LoTuS (Barbosa; Lima; Maia; Junior, 2017). LoTuS

is an open-source tool that allows the graphical modelling of software behaviour using LTS, providing a drag-and-drop GUI to create models. The extension allows the inclusion of energy cost information as part of transition labels. However, there is no support for any type of analysis using this information.

Representing energy in a behaviour model enables the application of techniques to find out relevant information that can help understand how software is consuming energy and which parts have been influencing the most this consumption. An example of this representation can be observed in Figure 1, which presents an LTS model where energy costs are shown between braces. A, B, C and D are transitions that represent an action occurrence in the software execution. In this example, a transition from state 0 to state 1 has a cost of 1 energy unit (e.g. Watts, Joules and others). With this model, it is possible to observe that state 0 is the initial state and, it is possible to reach state 3 by two executions, $A - C$ or $B - D$, with an energy costs of 4 and 3, respectively.

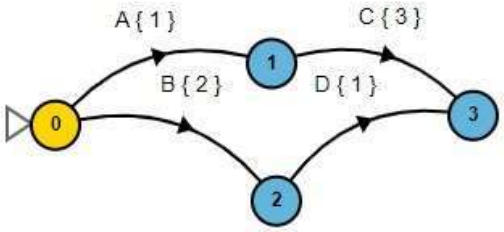

Figure 1 – LTS with energy costs

2.2.2 Modelling Probabilistic Information

In (Duarte; Maia; Silva, 2018), the authors used LTS models to analyse probabilistic behaviour. In this context, probabilistic information is inserted in an LTS, originating a *Probabilistic Labelled Transition System (PLTS)*, which is an LTS that also contains a probability value for each transition. Hence, a Probabilistic LTS (PLTS) is a structure $P = (S, s_i, \Sigma, T, lambda)$, where:

- S, s_i, Σ and T have the same definition as in an LTS,
- $\lambda : \Gamma \rightarrow [0, 1]$ is the transition probability function which assigns a positive real number less or equal to 1 for each transition such that the sum of the probability of all transitions leaving the same state is 1.

A PLTS example can be observed in Figure 2, where the probability of each transition is presented between parentheses.

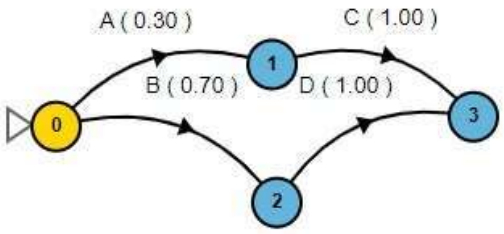

Figure 2 – LTS with probability information

2.2.3 Combining Energy Costs and Probabilities

In this work, we combine the PLTS as proposed in (Duarte; Maia; Silva, 2018) with the LTS with energy costs presented in (Duarte; Alves; Toresan; Maia; Silva, 2019) to obtain a model where probabilistic properties involving energy consumption can be analysed. In this model, the energy cost of each transition is represented by the number between curly brackets and the probability of execution is represented by the number between parentheses. Figure 3 shows an example of this model.

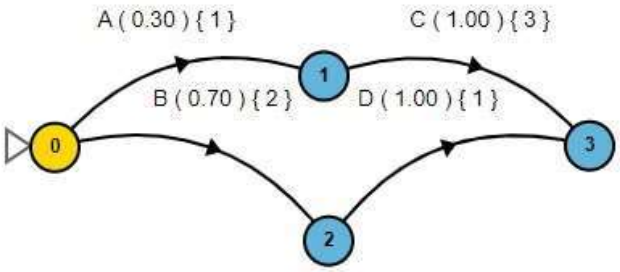

Figure 3 – Example of an LTS model that contains energy costs and probability values.

In this example, we can observe that the operation associated to transition A has 30% of probability of occurrence and consumes 1 energy unit, whereas the operation associated to transition C has 100% of chance of occurrence and consumes 3 energy units. Thus, the behaviour involving operations A and C has a 30% probability of occurring and consumes a total of 4 energy units.

3 Related Work

In this chapter we present the related work, divided into the group of those focused on measurement and/or direct analysis (i.e., analysis of raw data) of energy consumption and the group focused on model-based analysis.

3.1 Energy Measurement and Analysis

Software energy consumption research has so far focused mainly on the measurement of energy costs. The work described in (Singh; Dutta; VanderMeer, 2013) concentrated on detecting excessive or anomalous energy consumption in software, with the goal of optimising energy consumption of IT resources knowing how much power an application is consuming. Other authors analyse the influence of data structures on energy consumption (Pereira; Couto; Saraiva; Cunha; Fernandes, 2016) (Hasan et al., 2016) (Oliveira; Oliveira; Castor; Fernandes; Pinto, 2019), introducing a methodology to optimise Java programs and decrease their energy consumption by replacing data structures by a more energy-efficient alternative.

The research presented in (Schubert; Kostic; Zwaenepoel; Shin, 2012) described the development process of a profiler for measuring the energy consumption of source code points. In (Pereira, 2017), critical energy areas were identified using a statistical method to associate responsibility for energy consumption to source code components. There have also been studies focusing on estimating energy costs with the goal of extending battery life in embedded systems (Brandolese; Fornaciari; Salice; Sciuto, 2002) (Jayaseelan; Mitra; Xianfeng Li, 2006) and mobile devices (Hao; Li; Halfond; Govindan, 2013) (Couto; Carção; Cunha; Fernandes; Saraiva, 2014) (McIntosh; Hassan; Hindle, 2019).

The work described in (Singh; Naik; Mahinthan, 2015b) shows how some decisions during the software development process can influence energy consumption. In their experiments, they performed energy measurements of some Java Application Programming Interface (API), used for operations like file reading, file copy, file compression, and file decompression. They found out that using APIs with different buffer sizes it is possible to reduce energy consumption. Regarding user decisions, the work described in (Zhang; Hindle; German, 2014) shows how user requirements and choices directly impact on software energy consumption.

Some studies show how to decrease energy consumption during software development by using a search-based modification of the software system as an instance of Genetic Improvement (Petke et al., 2018)(Pereira; Couto; Saraiva; Cunha; Fernandes, 2016). In

this metric, it is desired to adapt the program and generate some related versions that hold some properties and improve others. The work described in (Lima et al., 2016) focused on observing the implications of development decisions in software energy consumption. In this study, they analysed a subset of decisions in Haskell applications and reported that some decisions could save up to 60% of energy resources.

None of the above studies employs a model as basis for their analyses as we propose here, which limits their analysis capacity, since abstractions enable analyses that could not be - or would hardly be - carried out directly on the actual software. For instance, without a model supporting the analysis, it would be hard to verify some cost characteristics such as the most costly execution or the cost of the most probable execution. With a model, it is also possible to compare different software versions, performing analysis before and after an evolution.

3.2 Model-based Analysis

A great part of the difficulty of producing and evolving energy-efficient software comes from the absence of software abstractions and tools (Pinto; Castor, 2017a). A way of analysing energy costs is through a behaviour model of the system, allowing a set of analyses that could improve the developer's understanding of their software energy behaviour.

Considering energy-cost modelling, some other approaches, such as (Baier; Dubslaff; Klein; Klüppelholz; Wunderlich, 2014) (Dubslaff; Klüppelholz; Baier, 2014), model energy costs using Markov chains and use PRISM (Kwiatkowska; Norman; Parker, 2001) to run quantitative analyses based on probabilistic information. In the approach described in (Baier; Dubslaff; Klein; Klüppelholz; Wunderlich, 2014), the costs/rewards feature of PRISM is used to assign costs to states/transitions. A limitation of the aforementioned approaches is that analyses about paths are not supported, due to the type of logic adopted to describe properties. Hence, questions that either require producing sequences of actions or evaluating specific executions (e.g., the most/least costly behaviour) could not be easily executed, if at all. In the LoTuS tool, we have an interface that provides a simple way of constructing models and including costs and probabilities, but its analysis capability is limited.

None of the above studies perform a model-based analysis with combined information about energy costs and probability of execution. Although, the study described in (Baier; Dubslaff; Klein; Klüppelholz; Wunderlich, 2014) employ the energy cost in money units, enabling the possibility of using energy units instead (probably requiring some adaptation). Moreover, they do not define what properties could provide relevant information about energy consumption nor do they present any guideline for software

developers on how to interpret analysis results and how to use these results to improve software energy efficiency. Without direction and orientation, the software developer does not have the information about what to look for and which changes could be made to enhance energy efficiency and may not get any improvement at all.

4 Property-Based Software Energy Analysis

Model-based analysis (Magee; Kramer, 2006) is an important technique for understanding software behaviour and identifying potential and real problems. One way of carrying out this analysis is using requirements (properties) to guide the process, so as to obtain objective results that can be used to support development and evolution decisions. For instance, using model-based analysis, it is possible to check impacts caused by a software modification, to identify violations of access to shared resources and to evaluate resource usage as RAM, time, power and others.

In this work, we consider the analysis of Labelled Transition Systems (LTS) (Keller, 1976) models augmented with energy (Duarte; Alves; Toresan; Maia; Silva, 2019) and probability information (Alves; Ferreira; Duarte; Maia, 2020), and propose high-level definitions of energy-related properties to help the analysis of software energy efficiency and offer support to decisions regarding energy consumption. We use the term "high-level" due the fact that these properties are not defined using any type of formal logic and we would like them to have a practical interpretation and usage, regardless of the target application and of their specific implementation.

The proposed properties involve the analysis of energy costs and probability values associated to executions of a software. We define an *execution (trace)* based on the definition presented in (Jr; Grumberg; Kroening; Peled; Veith, 2018), where an execution e of a deterministic LTS $M = (S, \Sigma, T, s_i)$ is a sequence of actions $a_1...a_n$, such that $n = |e|$ and $a_1, ..., a_n \in \Sigma$. An execution e always starts in the initial state s_i and proceeds according to the transition function T. In a simplified notation, a transition can be described by $s_j \xrightarrow{a_j} s_k \in T$, where $s_j, s_k \in S$ and s_j enables action $a_j \in \Sigma$ of the sequence. The execution always ends in a state $s_f \in S$, such that $s_n \xrightarrow{a_n} s_f$, where $s_n \in S$ and $a_n \in \Sigma$ is the last action of the sequence. As a consequence of this definition, an execution is a unique finite sequence of actions of a model M.

As it is known, some parts of the software involving loop structures can generate infinite executions. However, our definition of an execution assumes it is finite. For this reason, we consider a model-based testing coverage criterion called *one-loop path coverage* (Utting; Legeard, 2010) for the analyses of the properties. This coverage criterion considers at most one occurrence of a loop in each possible execution. Applying this idea makes it possible to obtain a total coverage of the model and guarantees the analysis of finite sequences only.

As commented before, to enable the property analyses proposed here, the LTS model must contain at least an energy cost value associated to each of its transitions,

similar to the model used in (Duarte; Alves; Toresan; Maia; Silva, 2019). To execute analyses considering probabilistic information, there must also be a probability value associated to each transition, as presented in (Alves; Ferreira; Duarte; Maia, 2020).

The results of these analyses on executions provide relevant information about software energy behaviour. Based on this information, a developer can better understand how the analysed software uses energy resources during its operation, so that they can make adjustments, if necessary. Moreover, the analyses can lead to the location of anomalous points of high consumption and support the investigation of what changes could be made to improve energy efficiency.

4.1 Property Definitions

The set of property definitions is divided into two groups:

- Energy properties: includes properties involving only the analysis of energy costs; and
- Energy-Probability properties: includes properties involving the analysis of the combination of energy costs and probabilities.

These properties cover the most important types of energy consumption analysis based on the framework proposed in (Duarte; Alves; Toresan; Maia; Silva, 2019) and the extended set of properties described in (Alves; Ferreira; Duarte; Maia, 2020). During the experimental phase, the set of properties was frequently modified and refined as some property definitions seemed to be redundant or not useful in practice and others had to be created to cover some information not provided by the current set of properties.

The two sets of properties are presented in the subsequent sections, as well as instructions on how to interpret their results.

4.1.1 Energy Properties

The set of energy properties is presented in Table 1. These properties involve only energy costs, and are defined based on situations where the probability information is not present or not available. Analyses based only on energy information can, in many cases, provide sufficient results to support software evaluation.

Figure 4 presents a model that will be used to describe how these properties can be applied to analyse energy efficiency. The energy cost associated to each transition is given in some energy unit[1] and is expressed by the number between braces. We represent this

[1] Here, we will express energy costs in Watts, following the unit provided by tools such as jRAPL, but any other energy unit could be used as long as all costs in the model consider the same unit.

Property ID	**Property Name**	**Input**	**Output**
`ExecCost`	Execution Cost	An execution trace	Energy Cost
`PossExecsCost`	Possible Executions Cost	Target state	List of possible executions in increasing order by energy total cost
`MaxCostExec`	Most Costly Execution	None	The execution that more spend energy
`ExecCostsList`	Execution Cost List	None	List of all possible executions in increasing order by energy total cost
`LimitSat`	Energy Limit Satisfaction	An execution and a boolean expression	`true` or `false`, according to the satisfaction or not, respectively, of the boolean expression by the execution cost value
`LimitsSatList`	Limit Satisfaction List	Inferior or superior energy limits	List of executions satisfying the criteria
`AvgConsumption`	Average Energy Consumption	None	Average energy value of all executions

Table 1 – Energy-based properties

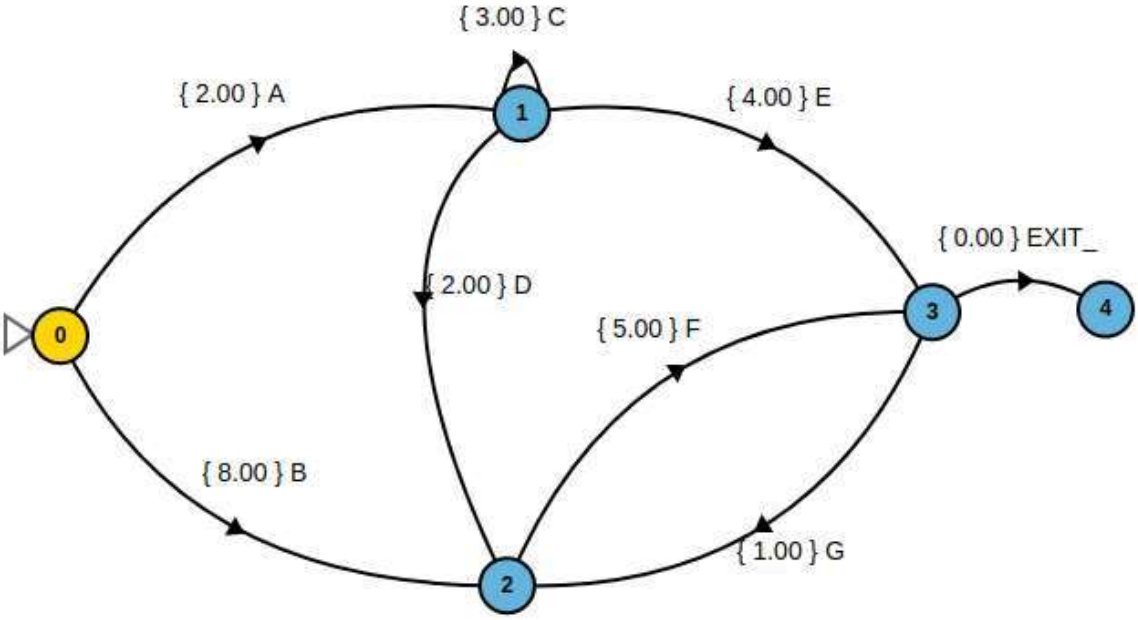

Figure 4 – Example model

cost using a function $cost(s, l)$, which takes a state identifier s and a label l of a transition originated in s as input and returns the energy cost associated to this transition. For instance, $cost(1, D) = 2.00W$ and $cost(3, G) = 1.00W$.

Execution Cost (`ExecCost`)

This property requires as input a specific execution and its analysis can be carried out by travelling the model accumulating the energy cost of each transition of the given execution, obtaining the total cost of the particular execution. Considering the model of Figure 4, we present the results of analysing this property for a few executions, presented in Table 2. Taking the input $A-C-E$, for instance, we obtain as a result the sum of the costs of the transitions that belong to the input execution, resulting in $2.00+3.00+4.00 = 9.00W$.

Input	Cost(W)
B-F	13.00
A-C-E	9.00
A-C-D-F-G-F	18.00

Table 2 – `ExecCost` property analysis

The result of an analysis of this property provides to the developer the cost of a specific execution. With this, they can, for example, check whether a change is required for this behaviour and can compare costs of selected executions to identify paths with higher energy consumption.

Possible Executions Cost (`PossExecsCost`)

To execute the analysis of this property, a target state is provided as input. The analysis occurs traversing the model to obtain all possible executions from the initial state of the model to the provided target state. As output, the analysis generates the list of executions and their respective total energy cost.

Applying an analysis of property `PossExecsCost` in the model of Figure 4 with state 3 as target state, the corresponding output is displayed in Table 3, where execution $A-E$, with cost of $6.00W$, has the lowest energy cost and $B-F-G-F$, with $19.00W$, is the most costly execution.

Input Target State	Output Execution	Cost(W)
3	A-E	6.00
	A-C-E	9.00
	A-D-F	9.00
	A-C-D-F	12.00
	A-E-G-F	12.00
	B-F	13.00
	A-D-F-G-F	15.00
	A-C-E-G-F	15.00
	A-C-D-F-G-F	18.00
	B-F-G-F	19.00

Table 3 – `PossExecsCost` property analysis

In a scenario where a new feature is developed (system evolution), new states and/or actions may be included in the model. This property analysis can be useful to determine the possible costs of executing this new feature in a software artifact.

Most Costly Execution (`MaxCostExec`)

The analysis of this property gives to the developer the execution in the system that spends more energy resources, starting from the initial state of the model. To obtain this result, we need to verify the cost of each possible execution and compare the results to determine the execution with the highest total cost value.

This analysis uses `PossExecsCost` to obtain all possible executions by the combination of the initial state and all other existing states as target states and identify their corresponding energy costs. With this information, we only need to compare costs to identify the most costly.

Applying the `MaxCostExec` analysis in the model of Figure 4, we obtain the information that execution $B - F - G - F$, with cost of $19.00W$, is the most costly in terms of energy consumption. This information tells the developer the highest possible cost of a system execution. Based on this, a developer could verify whether their system's energy consumption complies with a platform's upmost energy usage limitation.

Execution Cost List (`ExecCostsList`)

The analysis of this property consists in traversing the model using all possible executions starting in the initial state and leading to all other states of the model. The cost of an execution corresponds to the sum of the costs associated to each transition in the execution. This is similar to what happens in the analysis of `MaxCostExec` but, in this case, the analysis returns a list containing all possible executions starting from the initial state and the energy cost of each of them ordered by energy costs, from the lowest to the highest.

Table 4 displays the results obtained by the application of the `ExecCostsList` property analysis to the model presented in Figure 4. With this analysis, a broader scenario of the system energy consumption is provided, giving the software developer information about all behaviours that can occur. This information allows the developer to identify which executions consume more energy and, therefore, require more attention in case of a system change/evolution.

Energy Limit Satisfaction (`LimitSat`)

The analysis of this property requires an energy cost, a comparison operator (<, <=, >, >=, or =) and an execution. The result is the verification of whether the cost of

Output	
Execution	Cost(W)
A	2.00
A-D	4.00
A-C	5.00
A-E	6.00
A-C-D	7.00
A-E-G	7.00
B	8.00
A-C-E	9.00
A-D-F	9.00
A-C-E-G	10.00
A-D-F-G	10.00
A-C-D-F	12.00
A-E-G-F	12.00
B-F	13.00
A-C-D-F-G	13.00
B-F-G	14.00
A-D-F-G-F	15.00
A-C-E-G-F	15.00
A-C-D-F-G-F	18.00
B-F-G-F	19.00

Table 4 – `ExecCostsList` property analysis

the informed execution satisfies or not the boolean expression composed of the operator and the provided energy cost, returning `True` in case of satisfaction and `False`, otherwise.

Input			Output
Op.	Value	Execution	
<	5	A-C	`False`
<=	5	A-C	`True`
>	15	B-F-G-F	`True`
>=	15	B-F	`False`
=	10	A-D-F-G	`True`

Table 5 – `LimitSat` property analysis

Table 5 shows results of analyses of this property in the model of Figure 4. As an example, the analysis returns `False` when the cost of execution $A - C$ (5W) is verified against the limit < 5 and returns `True` when verified for the input $<= 5$.

This property allows the user to check whether specific executions are in conformance with a hardware energy limit. For instance, performing this property analysis giving as input the energy limit of the target platform where the software will run, it is possible to verify that an informed execution can run with energy consumption within that limit.

Limit Satisfaction List (`LimitsSatList`)

To perform this property analysis, it is necessary to inform as input an inferior and/or a superior energy limit. With this information, an analysis is carried out considering

all possible executions, returning a list of executions that satisfy the informed limits.

Performing the `LimitsSatList` analysis in the model presented in Figure 4, it is possible to obtain the results shown in Table 6. The table presents a list of all executions with energy cost between $5W$ and $10W$, executions with energy cost above $10W$ an executions with energy cost under $5W$.

Input		Output	
Inf. limit	Sup. limit	Execution	Cost(W)
5	10	A-C	5.00
		A-E	6.00
		A-C-D	7.00
		A-E-G	7.00
		B	8.00
		A-C-E	9.00
		A-D-F	9.00
		A-C-E-G	10.00
		A-D-F-G	10.00
10	none	A-C-E-G	10.00
		A-D-F-G	10.00
		A-C-D-F	12.00
		B-F	13.00
		A-C-D-F-G	13.00
		B-F-G	14.00
		A-D-F-G-F	15.00
		A-C-D-F-G-F	18.00
		B-F-G-F	19.00
none	5	A	2.00
		A-D	4.00
		A-C	5.00

Table 6 – `LimitsSatList` property analysis

`LimitsSatList` aids the developer to perform an analysis to verify and find executions that may not be in conformance with the expected energy costs or that are above a platform's consumption limits. Thus, the developer can determine which behaviours should be modified to fit the environment energy requirements.

Average Energy Consumption (`AvgConsumption`)

This property analysis consists in obtaining the average amount of energy consumed by the system under analysis. This information is calculated by the sum of all execution costs (obtained calculating `ExecCostsList`) divided by the number of possible executions, respecting the one-loop path coverage, previously described.

Performing this property analysis in the model presented in Figure 4, we obtain the information that the average energy cost of the analysed system is, approximately, $8.52W$. This information is useful to verify the standard energy cost of the system, i. e, the cost that will occur in most executions and should then be the expected frequent energy consumption of the system. Having this knowledge about all programs running in a

platform, a developer could estimate the overall average cost required to run all necessary applications.

4.1.2 Energy-Probability Properties

Here we describe a combination of the presented energy properties with probability information that represents the chance of occurrence of each software component represented in the model. This information is used to propose another set of properties that can provide valuable information to developers to support decisions regarding energy efficiency. These properties are presented in Table 7.

Combining energy properties with probabilistic information, the developer can conduct a more thorough analysis about components and specific executions. This analysis is important because some points of the code can have a large energy consumption, but with minimal chance of execution. Furthermore, this can generate unnecessary refactor work to modify parts that are rarely executed and, thus, have a small impact on the overall energy consumption. Looking from another perspective, some code parts can present a lower energy cost, but with a higher probability of execution. In case of loop structures, which can consume more energy depending on the cost of each iteration and the number of iterations, if the loop has a high chance of occurring, it could be a point of interest.

To exemplify how the analysis of these properties work, the model previously presented has been modified with the addition of values of probability of execution to each transition, such as in a PLTS, but keeping the energy cost values. This model can be seen in Figure 5, where the energy cost of each transition is represented by the number between curly brackets and the probability of execution is represented by the number between parentheses. This information can be originated, for instance, from a specialist's knowledge or from usage profiles analysis. The probability of a transition is given by a function $prob(t)$, which returns the probability associated to a transition t, where t follows the definition of the transition relation of an LTS presented in Section 2.2. For instance, considering the model in Figure 5, $prob(1 \xrightarrow{D} 2) = 0.25$ and $prob(3 \xrightarrow{G} 2) = 0.40$.

Property ID	**Property Name**	**Input**	**Output**
`ProbCostExec`	Probability and Cost of an Execution	Execution trace	Probability and energy cost value
`ProbOfMaxCostExec`	Probability of Most Costly Execution	None	Probability value and an execution trace
`CostOfMaxProbExec`	Energy Cost of Most Probable Execution	None	Energy cost value and an execution trace
`ExceedProb`	Probability of Exceeding	Superior Energy Thresholds	Probability value
`AvgProbableCost`	Average System Cost	None	Energy cost value

Table 7 – Energy-probabilistic properties

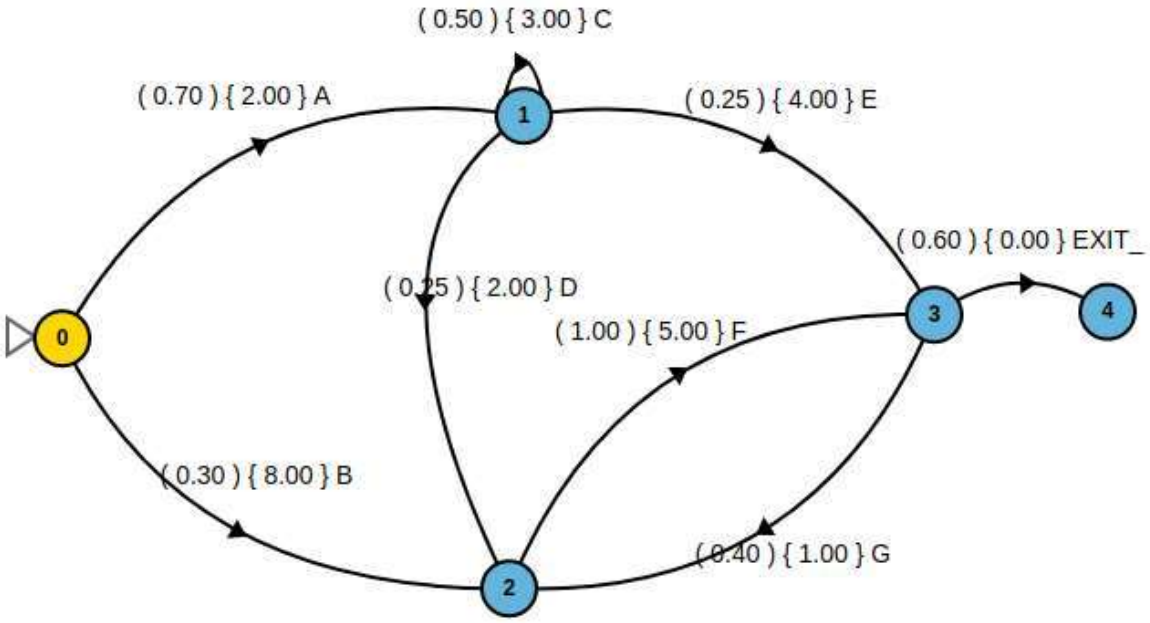

Figure 5 – Example model with probabilities

Next, each energy-probability property is described in detail, we discuss possible uses and present how they can be applied to our model of Figure 5.

Probability and Cost of an Execution (`ProbCostExec`)

The analysis of this property requires an execution as input and determines both the energy cost and the probability of occurrence of this execution. The probability of an execution is obtained by calculating the product between the probability of each transition that belongs to the execution.

Applying `ProbCostExec` in the model presented in Figure 5, we obtain the results presented in Table 8 for some selected executions. These results show, for example, that executions $B - F$ and $A - C - D - F - G$ have the same total cost, but the former has a probability of 30% of being executed against 3.5% of the latter. Also note that the execution with the highest cost ($A - C - D - F - G - F$) is not the one with the highest probability of occurring. Thus the importance of combining costs and probabilities to decide possible changes.

The analysis of `ProbCostExec` provides more information about specific system executions, allowing a developer to decide where they should change a software to improve energy efficiency and what strategies should be used to achieve this. This analysis could, for example, be used to support refactoring or evolution decisions.

Input	Output	
	Cost(W)	Probability
B-F	13.00	30%
A-C-E	9.00	8.75%
A-C-D-F-G	13.00	3.5%
A-C-D-F-G-F	18.00	3.5%

Table 8 – `ProbCostExec` property analysis

Probability of the Most Costly Execution (`ProbOfMaxCostExec`)

Analysis of this property combines the identification of the most costly behaviour that can be executed during a software run (similarly to `MaxCostExec`) with the probability of executing this behaviour. The probability of the execution is determined by the multiplication of the probability values associated to each transition that belongs to the execution (as it occurs in `ProbCostExec`).

After performing the analysis of `ProbOfMaxCostExec` in the model of Figure 5, we obtain the information that execution $B - F - G - F$ is the most costly execution, consuming $19.00W$, and have 12% of chance of being executed. This information is useful to verify whether the behaviour with the highest energy cost also has a high probability of execution, indicating a point in the software to be improved to get a better usage of energy resources. On the other hand, if the probability is low, the developer can decide whether it is worth changing the software or not, considering the required work and the chance of that specific execution occurring.

Energy Cost of the Most Probable Execution (`CostOfMaxProbExec`)

Similarly to the previous property, the analysis of `CostOfMaxProbExec` also requires exploring the whole model but, this time, with the aim of finding the behaviour with the highest chance of being executed. After finding this execution, a complementary analysis, similarly as it occurs in the analysis of `ExecCost`, identifies its respective energy cost.

Applying the analysis of the `CostOfMaxProbExec` property in the example model of Figure 5, we obtain the information that execution $B - F$ is the most probable to occur during a software run and has $13.00W$ of energy cost. This information helps identify the most critical point (i.e., the most frequent behaviour) in the software under analysis and the energy cost of running it. This information is useful to verify that the energy cost of the most probable execution - and the most probable energy consumption - is in conformance with a platform's energy limit.

Probability of Exceeding (`ExceedProb`)

The analysis of this property requires as input a superior energy threshold to perform a system model verification, identifying all the executions trespassing the imposed energy limit. After this, we calculate the chance of the energy cost not being in conformance with this limit by finding the highest probability value among those of the executions previously identified. This strategy of finding the execution with the highest probability of exceeding the limit is used because the analysis works on executions with variable sizes and, due to this fact, performing the sum of the probabilities of all executions could result in a value superior to 100%. This information is useful to the software developer because it shows the maximum chance of a behaviour that trespasses the limit to occur.

As an example of usage, applying the analysis of property `ExceedProb`, it is possible to obtain the information that the software represented in our example model has a 70% chance of consuming more than $10W$. This information can be useful to verify the possibility of the consumption becoming higher than that supported by the hardware. In this situation, a software developer can decide what changes are necessary to reduce this probability or this consumption.

Average System Cost (`AvgProbableCost`)

This property analysis consists in traversing all possible transitions of the model performing the multiplication between the energy cost and probability of occurrence of each possible execution. Accumulating this costs, it is possible to obtain the average cost of an execution of the system.

This information is useful, for example, to measure the impact of including or removing features or components during a software evolution. With this information, the developer can define strategies to decrease the average energy cost, since this value is a direct indicator of energy efficiency, being an approximation of the expected energy cost of executing the analysed software.

4.1.3 Property Combinations

The proposed set of property definitions, composed of a combination of analyses of energy costs and probability information, gives a developer a better view of their software energy behaviour, supporting the developer, for instance, when it is necessary to verify the compatibility between the software energy consumption and a platform where it has to run or when the developer needs to evaluate impacts of an evolution process.

With the results provided by the analyses of the proposed properties, the developer can understand more about software energy behaviour, observing what should be a concern

when an improvement of software energy efficiency is desired and become more apt to define changing strategies to achieve this goal.

4.2 Quick Guide to Property Usage

In this section, a quick guide in the form of a "Frequently Asked Questions"(FAQ) is presented. This guide has been constructed in order to provide a quick view about some situations where the analysis of the presented properties can contribute with useful information about software energy behaviour to handle software developers' decisions. It also indicates which property(ies) could be analysed in each case and how to interpret and use the results. The quick guide was constructed thinking in different scenarios where it is interesting for the developer to know energy properties of its software and also are based on empirical information acquired from our previous studies. The set of questions might not be complete for all scenarios, but it is intended as a guide for frequent questions about energy consumption.

Q1. What is my software energy consumption?

This question is potentially the most asked by software developers that desire to evaluate their software energy consumption. The answer to this question depends on the context where the software is running and what is relevant to the developer. Due to this fact, the answer to this question directly depends on the interpretation the developer will give to the obtained values. However, the software developer can make informed decisions based on results from the analysis of properties such as:

- `ExecCostsList`, that makes possible to identify costs for all possible executions, allowing the developer to obtain a range of the energy cost of their software, including the minimum and maximum execution cost;
- By using `AvgConsumption` property analysis, the software developer can obtain the mean cost spent by software executions;
- Finally, using the analysis of `AvgProbableCost`, it is also possible to access the mean energy cost of the software but, this time, considering the probability information (if available) about each software component.

Based on the knowledge obtained with these analyses, a developer can define strategies (such as refactoring, evolution or structure replacements) to improve energy efficiency if the current energy consumption is not in conformance with the analysed scenario or the developer's expectations. This information is also useful to know the consumption requirement needed to run the software on a platform.

Q2. Is the energy consumption of my software compatible with some energy limit?

In this situation, it is necessary to check the maximum amount of energy the software can spend during an execution, which means finding the most costly execution. This information can be obtained by analysing property `MaxCostExec` or, if the probability information is present in software model, the software developer can apply the analysis of property `ProbOfMaxCostExec` and obtain the most costly execution and the probability of its occurrence. The result of this analysis allows to identify the executions where the software spends more energy. With this information, it is possible to perform changes in the code to reduce its consumption or, if necessary (and possible), make changes in the hardware where software will run to allow a higher consumption limit.

It is important to highlight that this question considers only one software execution in isolation. For a long-run software, consider question Q3.

Q3. What is the cumulative energy consumption of a long-run software?

In case of a long-run software (i.e. a system that executes in a loop), it would be necessary to know the mean time spent to perform one execution and the time period that is desired to observe. With this information, it is possible to execute the property analyses presented in Q2. Also is possible to perform analysis such `AvgConsumption` if the probability information are present or `AvgProbableCost` in cases where not. Having this information, the developer can perform a simple calculation as demonstrated below:

$$\frac{T^{Total}}{T^{Run}} \times Cost$$

For instance, take a software that performs a complete run in 15 seconds and has an average energy cost of $12.5W$, found by analysis of `AvgConsumption` or `AvgProbableCost`. In a scenario that this software will run continuously for 500 seconds, we have the equation presented below:

$$\frac{500}{15} \times 12.5$$

As a result, we obtain an approximately amount of energy consumption of $416,67W$.

Q4. What is the probability of the system consumption trespassing an energy limit?

To obtain the answer to this question, the software developer needs to run an analysis of property `ExceedProb`, informing the desired energy limit. With this, the probability of the software energy consumption getting above this limit is calculated.

After this, the software developer can execute an analysis of other properties, such as `MaxCostExec`, to get the most costly behaviour, and `ProbOfMaxCostExec`, to obtain its probability of occurrence. In this case, if the probability of execution of the most costly behaviour is considered high, the software developer can refactor, replace structures or modify some features to reduce the energy cost of this behaviour. Another option is to perform the `ExecCostsList` property analysis to verify the cost of all executions and, with this, check which software parts could be modified to reduce energy consumption.

Q5. Where do I have to modify my software to improve its energy efficiency?

To improve the software energy efficiency, first we need to identify the points that require more attention (i.e. points where are more energy consumption). This information can be obtained by the analysis of some properties.

It is possible to use `ExecCost` to check the consumption of specific executions. The user also can perform the analysis of `MaxCostExec` to verify the execution that have more energy consumption or, to have a more complete scenario, by use `ExecCostsList` the developer can check the cost of all executions and verify that ones that have more consumption.

If the probability information is present in the analysed model, it is possible to perform the analyses of `ProbOfMaxCostExec` and `CostOfMaxProbExec` to obtain the probability of most costly and the cost of most probable executions. This executions require some attention due to the fact that they are the ones that can cause more impact in energy consumption. With this information, the software developer has a better idea about where they have to make changes in their software to improve its energy efficiency.

Q6. How does the inclusion of a new feature impact my software energy consumption?

To check this scenario, it is possible for the software developer to:

- Use the analysis of `ExecCost` property to compare the energy costs of impacted executions before and after the system evolution.
- Use the analysis of `PossExecsCost` property to check how the system evolution generates new behaviours, the costs of these behaviours and their impact compared to the previous behaviours.
- Check the maximum of energy consumption, using `MaxCostExec` before and after changes to check if the software consumption upperbound is still in conformance with the platform's energy limit.

- Finally, it is possible to execute `AvgConsumption` in cases where the probability information are not present, or `AvgProbableCost` if are present, before and after the evolution to evaluate the impacts of perform it by changes in mean of energy consumption during software execution.

All this information can show how the new feature/system evolution impacts the software energy efficiency providing the costs before and after the changes. With this, it is possible to visualise the impacts caused by system evolution and this gives to the software developer a more informative scenario to make decisions.

Q7. Which parts of my software consume more energy?

To have this information, the user can analyse property `MaxCostExec` to obtain the most costly execution. To have a more complete information, the user can combine this result with an analysis of property `AvgConsumption`, to obtain the average cost, and property `LimitsSatList`, applying this obtained cost as input to obtain the costs that are above the average. Another (and simpler) approach would be to analyse property `ExecCostsList` to determine a list of all possible system executions and, with this, observe which executions consume more energy. With this information, it is possible to define which software parts cause more impact in energy efficiency and, consequently, require more attention.

Q8. What is the average cost of executing my software?

If the probability values are known, this information can be accessed using the analysis of property `AvgProbableCost`. In cases where the probability information is not available, then it is possible to analyse property `AvgConsumption`. Both analyses return the approximate energy consumption in most software executions, based on the provided information. If desired or necessary, the software developer can execute `ProbOfMaxCostExec` and `CostOfMaxProbExec` to identify software points where performing changes can result in an energy efficiency enhancement.

Q9. What is the relation between the most costly executions and the most frequent executions?

To have the answer to this question, the software developer can analyse property `ProbOfMaxCostExec` to obtain the most costly execution and its probability of occurrence. To have a complementary information, it is possible to perform an analysis of property `CostOfMaxProbExec` to get the information about the most probable execution.

With the information resulting from these analysis, it would be possible to evaluate the most costly and the most probable execution. If they are the same, this means that part of the software is the most executed and also has the highest energy

cost, thus representing a part of the code to be further investigated. Otherwise, it is important to verify the energy cost of other executions and this can be made by using `ProbCostExec` to identify executions with elevated energy costs and high probabilities of occurrence. This executions are potentially relevant for software energy efficiency and should have more attention from the developer.

Q10. How much energy-efficient is my software?

Similarly to Q1, the answer to this question depends on the context where software under analysis is involved and which information is relevant to the developer. Due to this fact, this question does not have an objective answer, but there is some information that may indicate how efficient the software is in terms of energy consumption, such as:

- Total energy consumption, as demonstrated in Q1 and Q3 by analysing properties `ExecCostsList`, `AvgConsumption` and `AvgProbableCost`;
- Average energy consumption, as explained in Q1 and Q8 by analysing properties `AvgConsumption` and `AvgProbableCost`;
- The most probable energy cost, obtained through the analysis of properties `PCM`, `CostOfMaxProbExec` and `ProbCostExec`, as discussed in Q9;
- The highest software consumption, which can be verified by analysing properties `MaxCostExec` and `ExecCostsList` or by the combination between `AvgConsumption` and `LimitsSatList`, as demonstrated in Q7;
- The software adherence to an imposed limit, answered in Q2 and Q4 through the analysis of properties `MaxCostExec`, `ProbOfMaxCostExec` and `GCE`.

All these questions can be answered by analysing the indicated properties of the proposed set, thus providing clues to the software developers about their software energy-efficiency. Although answers might not be direct or objective, the developer can gather enough information to move towards the appropriate answer, considering their context, environment limitations and application domain.

5 Results

In this section, experiments involving the proposed properties and our preliminary results are presented[1]. Model construction can be carried out manually, based on the knowledge of the developer, or using some model extraction approach, such as the one presented in (Duarte; Kramer; al., 2017). We emphasise that the precision of the analyses depends directly on how well the model represents the software behaviour and the costs and probabilities associated to each transition. For the experiments, we used the LoTuS tool (Barbosa; Lima; Maia; Junior, 2017) to perform a visual and intuitive modelling, and energy costs were collected using the jRAPL framework (Liu; Pinto; Liu, 2015).

5.1 Implementation of Property Analysis

Here, we describe the implementation[2] of the analyses of the defined set of properties. This implementation was developed in the Python language[3], mainly due to our familiarity with this programming language and its processing speed. Moreover, some consolidated libraries, such as NetworkX[4], are available to be readily used with Python. NetworkX allows a developer to construct and operate in graph structures and its derivations and, thus, it was widely used in this work.

As a first step, we had to define how to deal with loop structures. As we know, loop structures can generate infinite executions and, as mentioned in Chapter 4, our definition of an execution assumes it is finite. For this reason, we used the *one-loop path coverage* (Utting; Legeard, 2010) to avoid infinite executions. Hence, in our implementation, we perform a modification in the model structure to produce a model that follows the one-loop path principle. We call this process *loop flattening* and it has been designed to achieve a model without loops, but which preserves the energy and probabilistic behaviours of the original model in terms of finite executions. To make this process possible, we need to follow a sequence of well-defined steps, which will be described next. To exemplify how this process occurs in an LTS model, consider the example model presented in Figure 6.

To describe how our one-loop path algorithm works, some roles need to be defined:

- *Initial state*: the state which is target of a backward transition; i.e., a transition, that goes back in the model, returning to a previously visited state. For example,

[1] Data for these experiments can be found at https://github.com/danilodsa/
[2] The algorithm can be found at https://github.com/danilodsa/
[3] https://www.python.org/
[4] https://networkx.org/

transition C in Figure 6. [5];

- *Loop state*: the state which is origin of a backward transition;
- *Intermediary state*: the state(s) that is(are) in the path between the Initial and the Loop states;
- *Final state*: the state(s) that is(are) the target of non-backward transitions from the the Loop state;
- *Link state*: the state(s) that is(are) created to simulate a one-loop execution.

The Loop and Initial states are unique for each loop. The set of intermediary states can be empty, whereas the set Link state and the set of Final states are non-empty.

The first step of the process is to identify all loop occurrences in the model using the NetworkX library features. In our example, shown in Figure 6, a loop occurs in execution $0-1-2-0$. When we assign the roles mentioned before, state 0 is the Initial state, state 1 is an Intermediary state, state 2 is the Loop state and state 3 is a Final state.

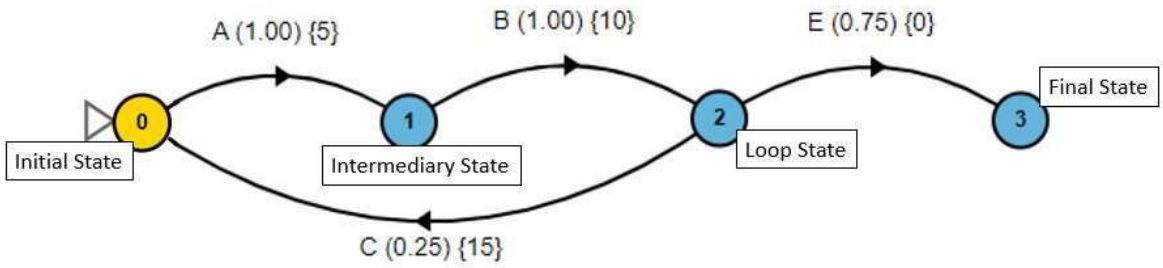

Figure 6 – Model before applying the *loop flattening* process

For each loop, we store all information about transitions that compose it (i.e., Loop state, Initial state, Intermediary states, Final states, labels, costs and probability information) and then eliminate the backward transition. For instance, transition C is removed from the model in our example, as shown in Figure 7, since it returns from state 2 to state 0.

The next step consists in creating new states called *Link states* for each removed loop (states 4, 5 and 6 in the model of Figure 8), which will be used to simulate one loop occurrence. To do so, we recall the information about each removed transition, stored in the first step, and use it to connect the involved states. We create one link state for each

[5] In our LTS representation, states are presented in order, from the start state. Hence, a backward transition is a transition originated in a state with a higher identifier and with destination to a state with a lower identifier.

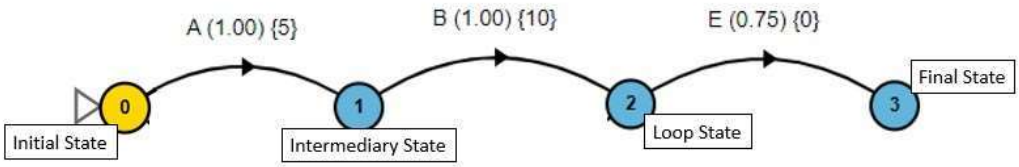

Figure 7 – Backward transition removal

state involved in the loop. In our example, three link states are required to reproduce the loop because there were 3 states involved (0, 1 and 2). This re-connection step preserves the original costs but creates only two possible executions from the Initial state to the Final state: one considering an execution without a loop ($0-1-2-3$) and another considering one occurrence of the original loop ($0-1-2-4-5-6-3$). Also it is possible to see that some transitions, such as E, had its probabilities recalculated to keep consistent the system probabilistic behaviour. This occurs because in the original model (see Figure 6), reaching the state 2, there was 25% of chance of performing the loop. After that, reaching state 2 another time would mean again that there was a 25% of chance of a new execution of the loop. However, as our execution can only allow one occurrence of the loop, after returning to state 2 (simulated by state 6 in the resulting model), we can only move to the Final state.

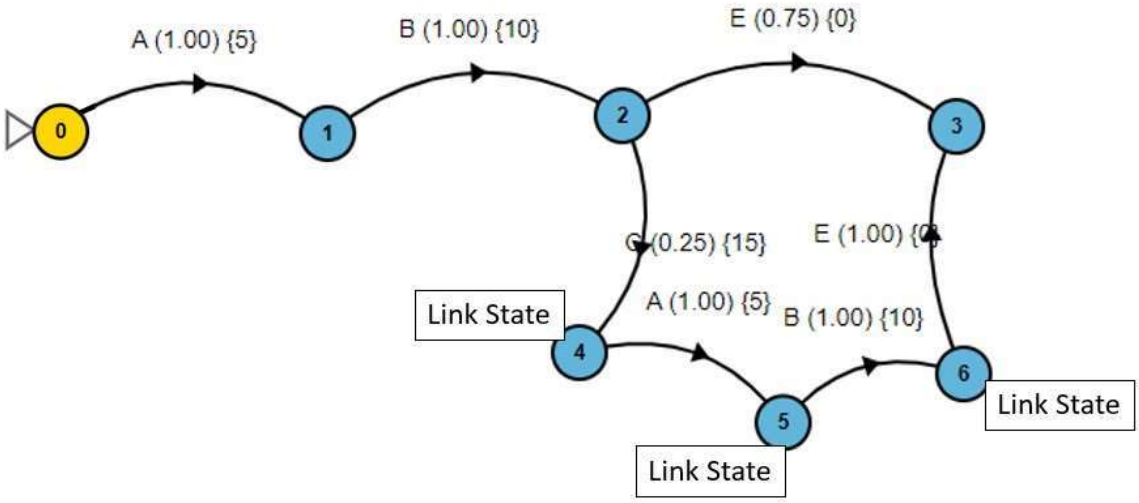

Figure 8 – Link state creation

After performing the presented steps, we have a *loopless model* containing only finite executions of the original model, respecting the one-loop path rule. With this model, it is possible to perform the analysis of the proposed properties without worrying about infinite executions. Moreover, some information can be lost once each loop execute only one time. This information can include, for instance, a decision structure that define if the loop will occur more times.

5.1.1 Limitations of the Implementation

Our implementation does not work well when the model contains nested loops. In these cases, during the experiments, we observed that NetworkX gives a list of all loops but does not identify those inside other loops. This limitation can make the analysis of models with nested loops not as accurate as with models with only simple loops. The set of not yet implemented properties consists in `ExceedProb`, `LimitSat` and `LimitsSatList`. This properties were not implemented due to this complexity and time necessary to perform this implementation. Furthermore, we intend to implement them in future work.

Also, the one-loop path coverage imposes a limitation on the path length and some information can be lost. For example, decision structures that are dependent on loop occurrences, as in a WHILE loop that only continues depending on an inside loop variable value.

5.2 Experiments

To demonstrate an application of the proposed properties, we conducted some experiments performing the modelling and evaluation of the energy behaviour of some software. These experiments are divided in three categories, following the scenarios of application described in Chapter1: a single component system analysis, a single component evolution and finally, a system constructed by a component combination.

In these experiments, we focused on performing the analysis of existent software. We intend to demonstrate how our research, the proposed set of properties and the quick guide can be applied in some cases that can occur in the real world, where a software developer have to make decisions about its software performance and target platforms. The experiments were executed in a computer with an Intel Core i5 with RAPL enabled and 16GB of RAM and the unit used was `Watt`, as used in the experiments performed in (Liu; Pinto; Liu, 2015).

5.2.1 SMTP Protocol

In the first experiment, we performed the modelling and evaluation of the software energy behaviour of the SMTP Protocol class, part of the SMTP implementation of the ristretto benchmark[6]. The model presented in Figure 9 was obtained using the model extraction process described in (Duarte; Kramer; al., 2017). In this process, we used the test cases provided with the benchmark for this class to execute an annotated version of the code and produce traces, which were used as input to the LTSE (Duarte; Kramer; al., 2017)(Duarte; Maia; Silva, 2018) tool to construct the LTS model. The probability

[6] http://columba.sourceforge.net/ristretto-1.0-docs/

information was obtained by analysing how many times a transition between a pair of states was executed according to what was recorded in the traces and the number of occurrences of the origin state. The energy values represent the average consumption based on 1000 executions for each method of the class.

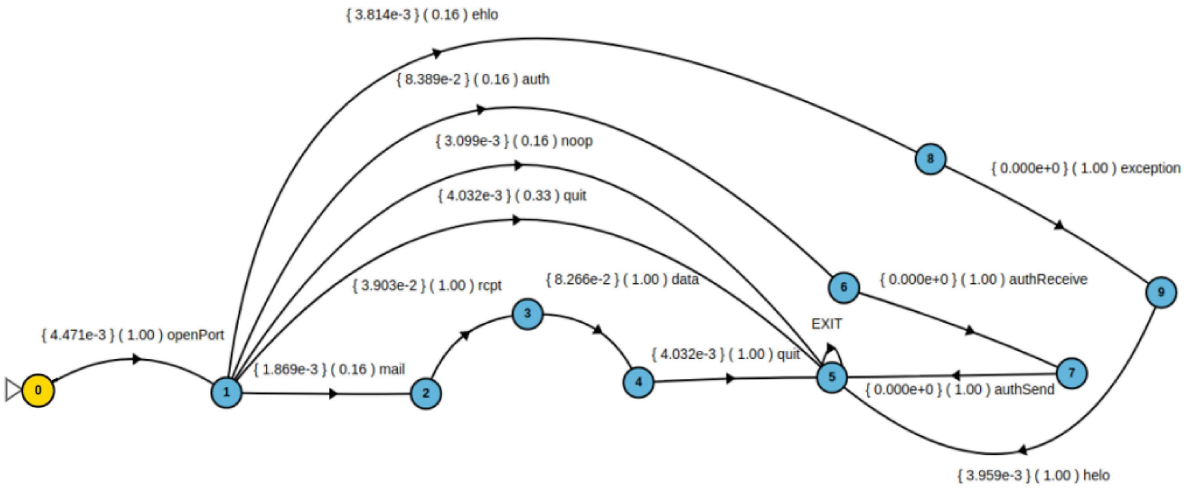

Figure 9 – Model of the SMTPProtocol class

We explore a scenario where we need to use the SMTP to transfer e-mail messages from an embedded, battery-powered system. Supposing that there is a limitation of the amount of energy that this task can spend of 0.05W per operation, is this implementation of the SMTP suitable for the presented scenario?

To answer this question, we can use our quick guide (presented in the Section 4.2) and follow the suggestion described in Q4. As a result, we can have the information that an execution of this implementation of the SMTP has a chance of 32.32% of trespassing the limit of 0.05W. In this case, we should use another implementation of this algorithm or we could make some changes in the source code to improve energy efficiency.

If we take the second option, to identify where changes should be made to reduce the software energy consumption, the quick guide shows in Q5 the use of `ExecCostsList` to return the costs of all possible executions. With this information, it is possible to observe that *openPort-mail-rcpt-data-quit* has the highest energy cost. This information can also be obtained using the `MaxCostExec` property, as mentioned in quick guide, but the `ExecCostsList` will give to the developer a broader scenario, more useful in this case of analysis. The developer can choose between these approaches by the effort that will be required to apply them.

In this system, the probability information is present, which allows us to get more information about the software behaviour. We can ask, for instance, what is the chance of the most costly execution occur (Q9) and receive the answer of 16.66% of chance. With this information, the developer can determine whether this value is high according to the

particular application. If that is the case, this execution requires some attention and a modification could reduce the software energy consumption. On the other hand, if the resulting chance is significantly low, the effort to perform a refactoring in this execution may not pay off.

Execution	Cost (W)
openPort	0.004471
openPort-ehlo	0.008285
openPort-auth	0.088361
openPort-noop	0.00757
openPort-quit	0.008503
openPort-mail	0.00634
openPort-ehlo-exception	0.008285
openPort-auth-authReceive	0.088361
openPort-mail-rcpt	0.04537
openPort-ehlo-exception-helo	0.012244
openPort-auth-authReceive-authSend	0.088361
openPort-mail-rcpt-data	0.12803
openPort-mail-rcpt-data-quit	0.132062

Table 9 – `ExecCostsList` analysis for the SMTP model

Also, it is possible, as shown in the quick guide, to calculate the average cost of the system (Q8). Analysis of the `AvgConsumption` property shows us that the average consumption of a run of the SMTP implementation is about 0.023W. This information gives to the developer the expected energy cost of executing the software, which can help the developer make decisions considering the platform where the system will run. With the probability information, a more precise analysis can be done with the `AvgProbableCost` property, that shows us the expected value is 0.15W based on the cost and probability provided information.

5.2.2 Matrix Multiplication Evolution

In software engineering, the term *software evolution* describes the process of changing a software in one or more attributes or characteristics, adding new features or removing obsolete functionalities (Lehman; Ramil, 2003). These software changes can cause alterations in the system behaviour and, consequently, modify the software energy consumption.

The proposed set of properties also can be used to evaluate how the system energy consumption is affected by the evolution of the software. The experiments described below show how to measure the impact of a system evolution in energy consumption by the comparison of energy properties of the system before and after the evolution.

To demonstrate how to employ our work during a software evolution phase, we use an experiment involving a matrix multiplication system. We performed the analysis of a system considering a possible evolution. This analysis was conducted on a matrix

multiplication algorithm[7], in which the matrices to be multiplied contain values that can be of type `Short`, `Int`, `Long`, `Float` or `Double`. The algorithm works receiving as input a selected data type, which determines the type of matrices that will belong to the operation through the transitions **opt0** for `Float` matrices, **opt1** for `Short` matrices, **opt2** for `Double`, **opt3** for `Int` and finally, **opt4** for `Long` type. Two matrices are randomly created with size 1000 x 1000, containing only values of the selected type, and then multiplied. Figure 10 shows how this system can be modelled as an LTS.

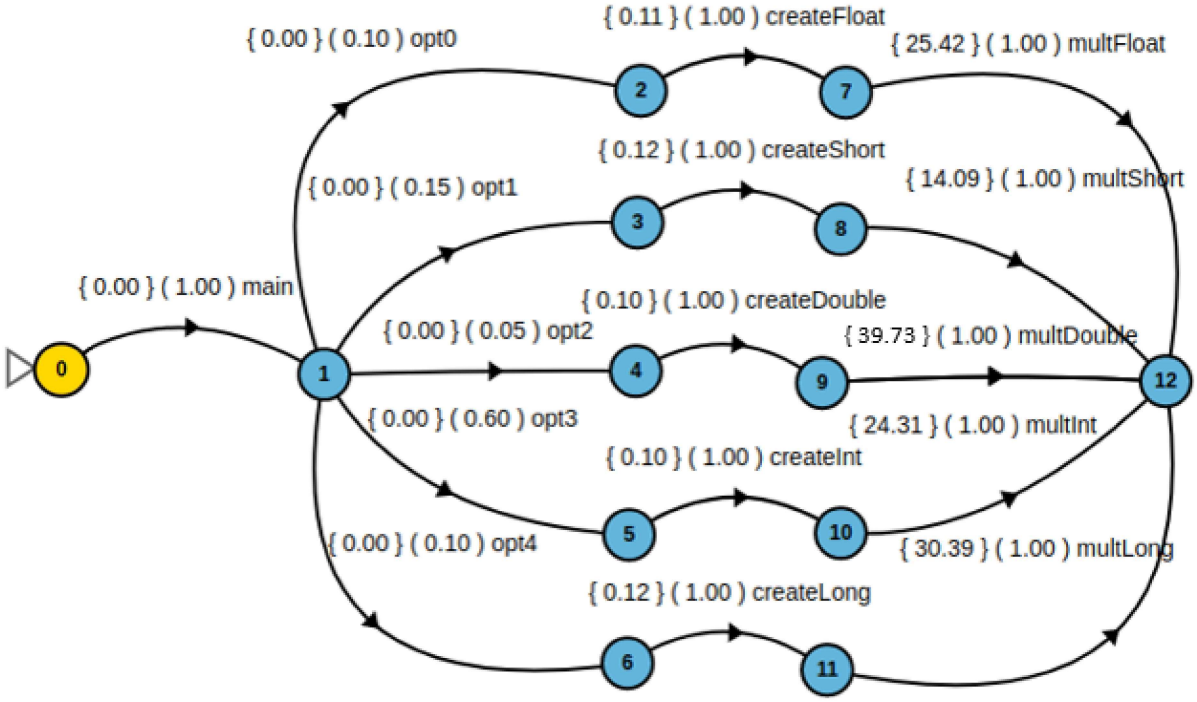

Figure 10 – Model of the matrix multiplication system

Considering some of the possible questions from the quick guide, we can get information about the software energy behaviour and, with this, it is possible to compare the results before and after the evolution to analyse the impacts on the energy behaviour. Evaluating a software before and after an evolution can be important when the system runs in a energy limited platform and the developer needs to keep the energy consumption under this limitation. Another possible case occurs when the developer needs to decide between storage structures, for example, and wants to verify the trade off of realising this evolution and evaluate the impact in complexity, time or capacity.

For instance, if the developer wants to know the part of the code with the highest energy consumption in this system (Q7), it is possible to observe that the execution that operates with matrices with Double values (*main-opt2-createDouble-multiplyDouble*) consumes 39,83W and represents the highest energy consumption in this system before the evolution.

[7] Adapted from jRAPL benchmarks available at <http://kliu20.github.io/jRAPL>.

As mentioned before, the probability information gives us the possibility to better understand the software behaviour. With this, we can ask about the most probable execution (Q9) and obtain the execution *main-opt3-createInt-multiplyInt* having the highest chance of occurrence (60%). Applying the suggestions presented in Q8 of the quick guide, we were also able to obtain that the average cost of this system is 24.37W.

Working in a specific context, matrix operations are part of several computing processes, including graphic processing (Jodra; Gurrutxaga; Muguerza, 2015). In addition to multiplication, matrix transposition is another operation that is frequently used in this area (Baqais; Assayony; Khan; Al-Mouhamed, 2013). Hence, we could implement an evolution to include this new operation. The model obtained after this process of evolution is presented in Figure 11. In state 13, there are new transitions to represent the matrix transposition after the matrix creation and two transitions that permit the system to realise a matrix transposition and return to matrix type selection. After constructing the model, we can ask: how does the inclusion of this new feature impact my software energy consumption?

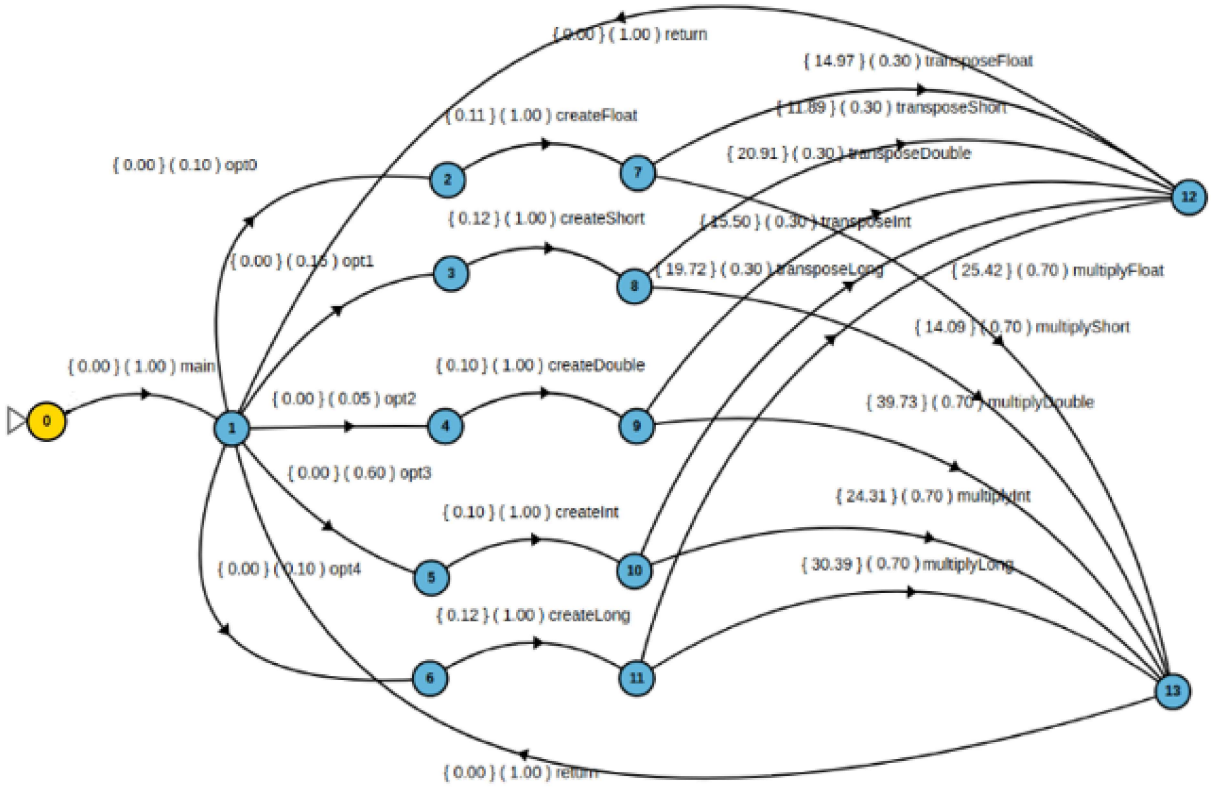

Figure 11 – Model of the matrix multiplication system evolution

Before the evolution, the average cost of this system was 24.37 watts and, now its average cost is 21.77 watts. The average cost was reduced by the inclusion of the new feature because the multiplication consumes more energy than transposition. Before, the multiplication had 100% of chance of occurrence and now, its chance is divided with the

	Before	After
Average cost	24.37	21.77
Most costly path probability	5%	3.5%
Probability of exceeding 30W	20%	10.5%
Most probable path	main-opt3-createInt-multiplyInt	main-opt3-createInt-multiplyInt
Most probable path probability	60%	42%

Table 10 – Impacts of the evolution

transposition operation. We also can observe that the most costly path is still the same, but now we have a 3.5% chance that the system will execute a Double matrix multiplication. Analysing the probability of the system exceeding 30W before the evolution provides a chance of 20%, now, this chance went down to 10,5%. This comparison can be better observed in the Table 10.

Now, the system also has the possibility of returning to the matrix type selection to execute another operation with another type of matrix. We can check, for example, the cost and probability of executing an Int matrix multiplication and then performing a transposition of a Short matrix. This analysis results in an energy cost value of 36.42 watts with a 1.8% of chance. This analysis could be interesting to evaluate loop structures in the system and its result is useful to analyse the energy consumption growth during an execution and how the probabilities influence such consumption.

By evolution of the matrix multiplication system, the developer would obtain an improvement in understanding their system energy behaviour. This enhancement could be useful if the developer needs to adjust the software to a platform with energy limitations.

5.2.3 FTP Protocol + Compression

We performed an experiment composing two systems to originate a new software. We work on a hypothetical scenario, where the software developer has to work with file transfers using the File Transfer Protocol (FTP) (Postel; Reynolds, 1985) and would like to know whether there is an energy gain involved in compressing the files before transmission.

The FTP involves sending files between a client and a server. To establish communication, the user provides their credentials, including identification and password. With the user successfully authorised, it is possible to transfer files between both sides (Kurose; Ross, 2007). The Apache Commons Library[8] is used in this experiment and an adaptation of the code of the FTP implementation was created from a repository of Java codes and

[8] https://commons.apache.org/

```
1public class FTP {
2  public void submitFTP(localFile,remoteFile){
3    ...
4    FTPClient ftpClient = new FTPClient();
5    try {
6      ftpClient.connect(server, port);
7      ftpClient.login(user, pass);
8      ...
9      inputStream = new FileInputStream(localFile);
10     if(firstApproach){ //using an InputStream
11       System.out.println("Start uploading file");
12       done=ftpClient.storeFile(remoteFile,inputStream);
13       inputStream.close();
14       if (done) {
15         System.out.println("Uploaded successfully.");
16       }
17     }
18     else { // using an OutputStream
19       inputStream = new FileInputStream(localFile);
20       outputStream=ftpClient.storeFileStream(remoteFile);
21       byte[] bytesIn = new byte[4096];
22       int read = 0;
23       while ((read = inputStream.read(bytesIn)) != -1) {
24           outputStream.write(bytesIn, 0, read);
25       }
26       inputStream.close();
27       outputStream.close();
28       completed=ftpClient.completePendingCommand();
29       if (completed) {
30           System.out.println("Uploaded successfully.");
31       }
32     }
33   } catch (IOException ex) {
34     System.out.println("Error: " + ex.getMessage());
35   } finally {
36     try {
37       if (ftpClient.isConnected()) {
38         ftpClient.logout();
39         ftpClient.disconnect();
40       }
41     }catch (IOException ex) {
42       ex.printStackTrace();
43     }
44   }
45 }
46}
```

Figure 12 – FTP program source code

tutorials[9], as shown in the partial source code in Figure 12. and the model obtained is shown in Figure 13. In this experiment, an upload of a 15MB file is executed from a client to a server, given the IP address, port number, username, and password.

The source code of the program that performs the data compression and calls the FTP to perform the transmission is displayed in Figure 14 and the model obtained can be observed in Figure 15. It is important to observe that the costs to perform the transmission in the composed model are different from the previous model. This occurs due to the compression, which reduces the file size and, consequently, reduces its transmission cost.

[9] Available at <https://www.codejava.net/java-se/ftp/java-ftp-file-upload-tutorial-and-example>

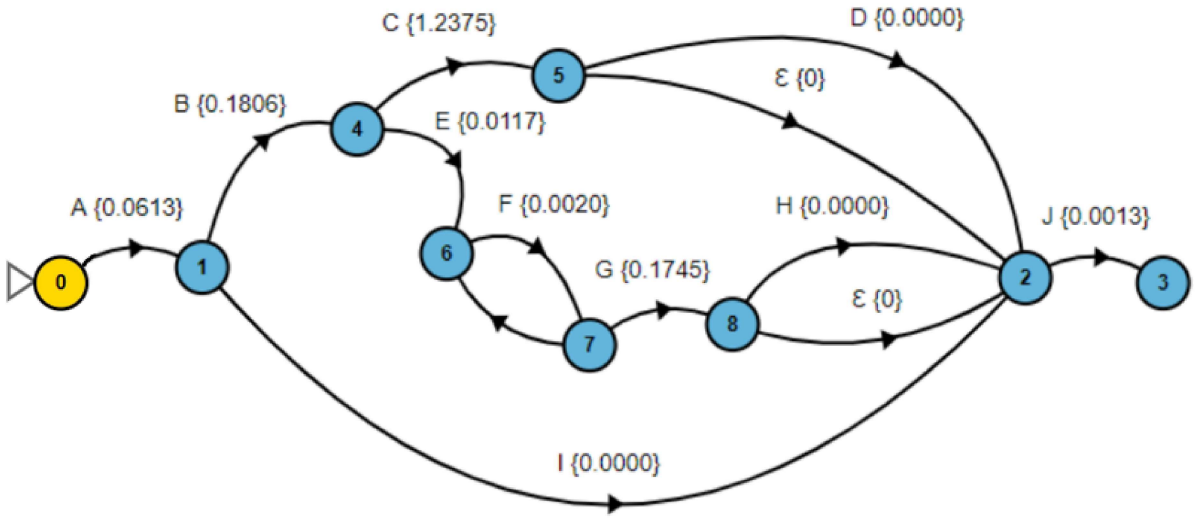

Figure 13 – FTP protocol system modelling

```
1  public static void compress(String file){
2    try{
3      byte[] buffer = new byte[256000];
4      FileOutputStream fos = new FileOutputStream("file.zip");
5      ZipOutputStream zip = new ZipOutputStream(fos);
6      FileInputStream in = new FileInputStream(file);
7      int len;
8
9      while((len = in.read(buffer)) > 0) {
10       zip.write(buffer, 0, len);
11     }
12
13     in.close();
14     zip.closeEntry();
15     zip.close();
16
17   }catch(IOException ex){
18     ex.printStackTrace();
19   }
20  }
21}
```

Figure 14 – Compression system source code

Taking this scenario, with the software developer having the energy consumption of each software part and an LTS model that represents this software, it is possible to use the quick guide to get some information, asking, for example, "How does the inclusion of a new feature impact my software energy consumption?"(Q6). In this case, the new feature is the data compression.

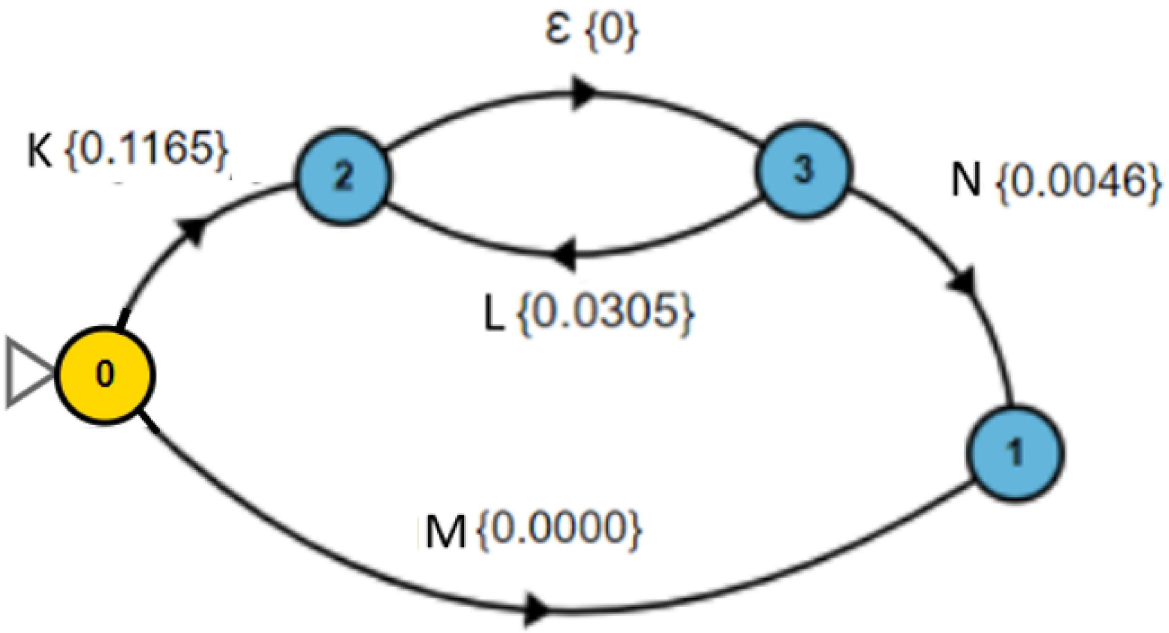

Figure 15 – Compressor model

```
1  public class main {
2    public static void main (String arg[]){
3      String file = 'home/Documents/file.pdf'
4      CompressZip compressZip = new CompressZip();
5      compressZip.compress(file);
6
7      FTP ftp = new FTP();
8      ftp.submitFTP(file,nameOfRemoteFile);
9    }
10 }
```

Figure 16 – Compressor + FTP system source code

The FTP energy cost, for this experiment, results in a value of $1.3478W$ without performing the compression. However, executing the file transfer performing the file compression before calling the FTP results in a total energy cost of $2.5728W$ (resulting file with 9Mb); i.e., $1.2250W$ more than the transmission without the compression. Hence, the energy cost to compress the file before sending it is higher than that of simply sending the original file.

Having this information in hands, the developer can direct its attention to experiment other compression methods, file sizes or other file types, for instance. Also, other transmission algorithms and protocols can be used with the objective to find the less expensive in terms of energy consumption or the developer can simply decide to perform

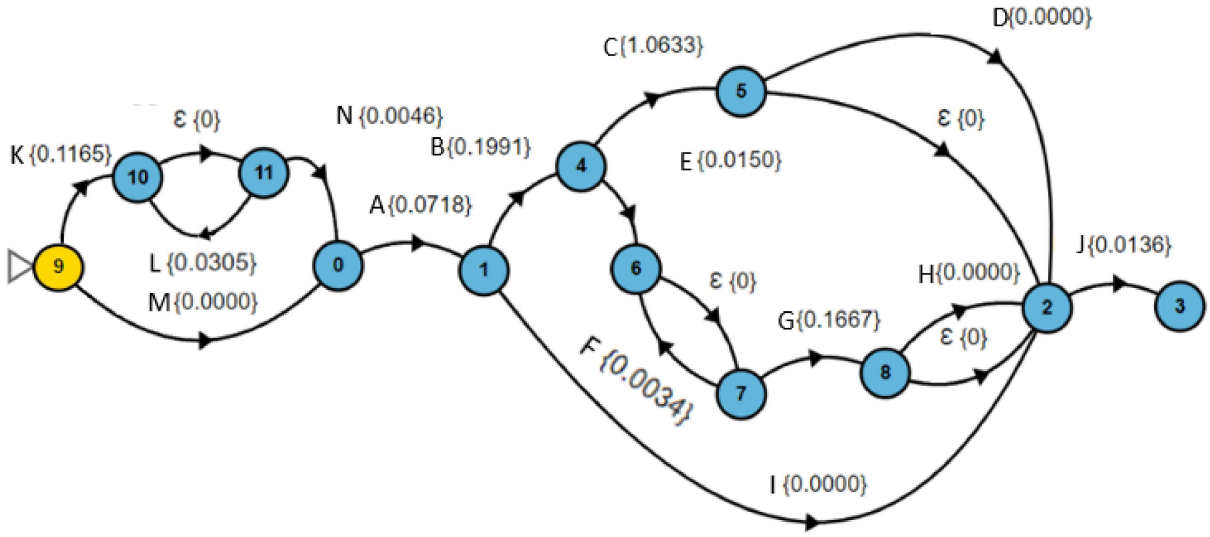

Figure 17 – System evolution resultant model

the transmission without compressing the file, that is, this experiment opens several options that the developer can choose to adequate the system energy consumption to their need.

This experiment showed us how the utilisation of the proposed set of properties and the quick guide can support the developer when the objective is analysing the energy consumption of a software before and after an evolution. Hence, to conduct a more precise analysis in this experiment, it would be necessary to evaluate different file sizes and types to check if is worth the effort to compress.

5.3 Discussion

All of the answers obtained with the experiments can provide information to support a software developer regarding choices that can be made in order to enhance their software performance in terms of energy consumption or in order to adapt the platform to run the analysed system.

Our idea was to present examples simple to understand and that are extracted from the real world. The same process to evaluate the STMP Protocol and the matrix system can be applied to large systems to perform a general analysis or to its isolated components to make a local evaluation. These analyses can be performed in any LTS model with costs and probability information. This can be used in software still in the design phase, if probabilities and costs of each part are previously known or estimated, to determine that the software under development attends the limitations of the platform where it will run. As demonstrated, this property analysis can also be applied to evaluate the energy consumption of an existing system or in a software evolution, where it is desirable to assess the impact of the modification on the energy consumption factor.

Even though our results show important information about energy behaviour analysis of the programs, we still need to further evaluate this analysis in most robust and multi-component systems. In the experiments, models were constructed either by hand or by using a model extraction tool and, the quality of the analysed model (i.e, the correct representation of the system behaviour) can have a big impact on the produced results. For this reason, models should have some type of conformance with the software behaviour. The origin of energy and probability values also is a relevant factor, since the analyses directly depend on these values. We obtained the probability information by the set of tests in the first experiment, and this can cause a bias depending on how the tests were constructed and their purpose. However, in the context of this work, our intention was just demonstrating the applicability of our proposed analyses and not having a precise model.

6 Application Areas

In this section, we discuss some possible application areas for the proposed property definitions and their corresponding analyses, highlighting how it can help users to address the inherent challenges of each area.

6.1 Component/Service-based development

More recently, microservices (Newman, 2015) have been gaining attention from both academia and industry, particularly due to its efficient manner to scale computational resources at runtime, thus becoming a trend as an architectural style for cloud-native applications (Kratzke, 2018). A challenging and yet not addressed question regards how to build energy-efficient systems upon a set of components or (micro)services. In this direction, developers could benefit from our set of property definitions to model and exercise the possible architectural configurations based on the energy consumption and probability of execution of each available component.

6.2 Refactoring

In the context of energy analysis, a developer could use the results of analyses of the defined properties to evaluate how applying a given refactoring (Fowler, 1999) would affect their local and/or overall energy consumption. Property analysis can support the decision-making process by providing results that can help weigh possible conflicting aspects. Different versions of a software could be compared and multiple changes considered.

6.3 Energy Optimisation

Ideas about energy complexity analysis have already been presented (Demaine; Lynch; Mirano; Tyagi, 2016), discussing how to determine, for instance, the minimum energy required to execute an algorithm. Using the results from analysing our properties, developers could compare the energy efficiency of their solutions with possible lower/upper bounds. It would also be possible to discover what parts of the code are more inefficient and explore possible solutions to improve efficiency.

6.4 Self-adaptive systems

Self-adaptive systems are able to adjust their behaviour or structure in response to their perception of the environment and the system itself (Cheng; Lemos; al., 2009). In this realm, energy consumption arises as a possible aspect that can be observed and lead a system to adapt at runtime in order to choose the most appropriate component to keeps the system overall energy consumption below a established acceptable threshold.

6.5 Embedded Systems

If we can measure and analyse energy costs for a given software, we can do this process using different hardware components and configurations to see how they affect energy-efficiency. In particular, embedded systems (Jayaseelan; Mitra; Xianfeng Li, 2006) are systems where hardware and software are combined and, therefore, energy consumption should account for both types of components. Using simulators of architectures and microprocessors, one could execute a program using different settings and analyse how they influence energy efficiency in this composed scenario.

7 Conclusions and Future Work

The work described here aimed to show the importance of energy consumption analysis during the software development process. We proposed a set of high-level property definitions that consider the energy cost and the probability of occurrence of each software part combined in an LTS model, which allows a developer to analyse their software energy consumption. To achieve the proposed objective, an implementation of the one loop path technique was proposed. Although the implementation presents some weakness in some nested loops and massive systems due to this experimental phase, it showed its efficiency on allowing the use of classic graph algorithms to support energy consumption analysis. We also help the developer understand how to enhance their software energy efficiency by presenting a list of possible real situations in a quick guide format, present common situations that the developer can encounter and how to proceed in each case.

The experiments conducted in this study showed how the analyses of these properties can be used to support the software developers on evaluating the energy consumption of their systems. We performed experiments modelling and evaluating the energy behaviour of a single component system, a single component evolution and finally, a system constructed through a component combination, presenting preliminary results obtained by the application of the properties proposed and following the quick guide instructions. To perform the experiments, an implementation of the one-loop path coverage and analyses of some of the proposed properties were developed.

As future work, we intend to study techniques and heuristics that are able to enhance the one-loop path implementation and generate model traces to perform analysis in systems with a substantial number of components and events with less effort and, with this, increase the set of implemented property analyses. We also intend to research and perform experiments in different scenarios with different limitations to enhance the actual set of properties and increase the number of possible analyses. With this, a new version of the quick guide can be proposed, containing more situations and questions to help the developer.

References

Albers, S. Energy-efficient algorithms. *CACM*, ACM, New York, NY, USA, v. 53, n. 5, p. 86–96, maio 2010. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1735223.1735245>.

Alves, D. da S.; Ferreira, O. A.; Duarte, L. M.; Maia, P. H. Probabilistic model-based analysis to improve software energy efficiency. In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2020. p. 132–136.

Baier, C.; Dubslaff, C.; Klein, J.; Klüppelholz, S.; Wunderlich, S. Probabilistic model checking for energy-utility analysis. In: _____. *Horizons of the Mind. A Tribute to Prakash Panangaden: Essays Dedicated to Prakash Panangaden on the Occasion of His 60th Birthday*. Cham: Springer International Publishing, 2014. p. 96–123. ISBN 978-3-319-06880-0. Disponível em: <https://doi.org/10.1007/978-3-319-06880-0_5>.

Baqais, A.; Assayony, M.; Khan, A.; Al-Mouhamed, M. Bank conflict-free access for cuda-based matrix transpose algorithm on gpus. In: *International Conference on Computer Applications Technology*. [S.l.: s.n.], 2013. p. 160.

Barbosa, D. M.; Lima, R. G. de M.; Maia, P. H. M.; Junior, E. C. Lotus@runtime: A tool for runtime monitoring and verification of self-adaptive systems. In: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. Piscataway, NJ, USA: IEEE Press, 2017. (SEAMS '17), p. 24–30. ISBN 978-1-5386-1550-8. Disponível em: <https://doi.org/10.1109/SEAMS.2017.18>.

Belgaid, C.; d'Azémar, A.; Fieni, G.; Rouvoy, R. *pyRAPL: A software toolkit to measure energy in python language*. 2019. <https://pypi.org/project/pyRAPL/>. Accessed: 2021-03-16.

Binkert, N. et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, ACM New York, NY, USA, v. 39, n. 2, p. 1–7, 2011.

Brandolese, C.; Fornaciari, W.; Salice, F.; Sciuto, D. The impact of source code transformations on software power and energy consumption. *Journal of Circuits, Systems, and Computers*, World Scientific, v. 11, n. 05, p. 477–502, 2002.

Cheng, B. H. C.; Lemos, R. de; al. et. Software engineering for self-adaptive systems: A research roadmap. In: _____. *Software Engineering for Self-Adaptive Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 1–26. ISBN 978-3-642-02161-9. Disponível em: <https://doi.org/10.1007/978-3-642-02161-9_1>.

Couto, M.; Carção, T.; Cunha, J.; Fernandes, J. P.; Saraiva, J. Detecting anomalous energy consumption in android applications. In: Springer. *Brazilian Symposium on Programming Languages*. [S.l.], 2014. p. 77–91.

Demaine, E. D.; Lynch, J.; Mirano, G. J.; Tyagi, N. Energy-efficient algorithms. In: *ITCS '16*. New York, NY, USA: ACM, 2016. (ITCS '16), p. 321–332. ISBN 978-1-4503-4057-1. Disponível em: <http://doi.acm.org/10.1145/2840728.2840756>.

Duarte, L. M.; Alves, D. da S.; Toresan, B. R.; Maia, P. H.; Silva, D. A model-based framework for the analysis of software energy consumption. In: ACM. *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. [S.l.], 2019. p. 67–72.

Duarte, L. M.; Kramer, J.; al. et. Using contexts to extract models from code. *Software and Systems Modeling*, v. 16, n. 2, p. 523–557, 2017.

Duarte, L. M.; Maia, P. H. M.; Silva, A. C. S. Extraction of probabilistic behaviour models based on contexts. In: *Proceedings of the 10th International Workshop on Modelling in Software Engineering - MiSE '18*. New York, New York, USA: ACM Press, 2018. p. 25–32. ISBN 9781450357357. ISSN 02705257. Disponível em: <http://dl.acm.org/citation.cfm?doid=3193954.3193963>.

Dubslaff, C.; Klüppelholz, S.; Baier, C. Probabilistic model checking for energy analysis in software product lines. In: *Proceedings of the 13th International Conference on Modularity*. New York, NY, USA: ACM, 2014. (MODULARITY '14), p. 169–180. ISBN 978-1-4503-2772-5. Disponível em: <http://doi.acm.org/10.1145/2577080.2577095>.

Fowler, M. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-48567-2.

Georgiou, S.; Rizou, S.; Spinellis, D. Software development lifecycle for energy efficiency: Techniques and tools. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 52, n. 4, p. 1–33, 2019.

Hao, S.; Li, D.; Halfond, W. G. J.; Govindan, R. Estimating mobile application energy consumption using program analysis. In: IEEE Press. *35th International Conference on Software Engineering (ICSE 2013)*. [S.l.], 2013. p. 92–101.

Hasan, S. et al. Energy profiles of java collections classes. In: ACM. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. [S.l.], 2016. p. 225–236.

Jayaseelan, R.; Mitra, T.; Xianfeng Li. Estimating the worst-case energy consumption of embedded software. In: IEEE. *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. [S.l.], 2006. p. 81–90.

Jodra, J. L.; Gurrutxaga, I.; Muguerza, J. Efficient 3d transpositions in graphics processing units. *International Journal of Parallel Programming*, Springer, v. 43, n. 5, p. 876–891, 2015.

Jr, E. M. C.; Grumberg, O.; Kroening, D.; Peled, D.; Veith, H. *Model checking*. [S.l.]: MIT press, 2018.

Keller, R. Formal Verification of Parallel Programs. *CACM*, v. 19, n. 7, p. 371–384, July 1976.

Khalid, H.; Shihab, E.; Nagappan, M.; Hassan, A. E. What do mobile app users complain about? *IEEE Software*, v. 32, n. 3, p. 70–77, May 2015. ISSN 0740-7459.

Kratzke, N. A brief history of cloud application architectures. *Applied Sciences*, v. 8, n. 8, 2018. ISSN 2076-3417. Disponível em: <https://www.mdpi.com/2076-3417/8/8/1368>.

Kurose, J.; Ross, K. Computer networking: A top-down approach (6th edition). In: . [S.l.: s.n.], 2007.

Kwiatkowska, M.; Norman, G.; Parker, D. PRISM: Probabilistic symbolic model checker. In: Kemper, P. (Ed.). *Proc. of the Tools Session of Aachen 2001 Intl Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*. [S.l.: s.n.], 2001. p. 7–12. Available as Technical Report 760/2001, University of Dortmund.

Lehman, M. M.; Ramil, J. F. Software evolution—background, theory, practice. *Information Processing Letters*, Elsevier, v. 88, n. 1-2, p. 33–44, 2003.

Li, D. et al. Software energy consumption estimation at architecture-level. In: *2016 13th International Conference on Embedded Software and Systems (ICESS)*. [S.l.: s.n.], 2016. p. 7–11.

Li, S. et al. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. [S.l.: s.n.], 2009. p. 469–480.

Lima, L. G. et al. Haskell in green land: Analyzing the energy behavior of a purely functional language. In: IEEE. *2016 IEEE 23rd international conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.], 2016. v. 1, p. 517–528.

Liu, K.; Pinto, G.; Liu, Y. D. Data-oriented characterization of application-level energy optimization. In: Egyed, A.; Schaefer, I. (Ed.). *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. p. 316–331. ISBN 978-3-662-46675-9.

Magee, J.; Kramer, J. State models and java programs. *wiley Hoboken*, v. 10, p. 332036, 2006.

Manotas, I.; Pollock, L.; Clause, J. Seeds: a software engineer's energy-optimization decision support framework. In: ACM. *Proceedings of the 36th International Conference on Software Engineering*. [S.l.], 2014. p. 503–514.

McIntosh, A.; Hassan, S.; Hindle, A. What can android mobile app developers do about the energy consumption of machine learning? *Empirical Software Engineering*, Springer, v. 24, n. 2, p. 562–601, 2019.

Newman, S. *Building Microservices*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2015. ISBN 1491950358, 9781491950357.

Oliveira, W.; Oliveira, R.; Castor, F.; Fernandes, B.; Pinto, G. Recommending energy-efficient java collections. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2019. p. 160–170.

Pang, C.; Hindle, A.; Adams, B.; Hassan, A. E. What do programmers know about software energy consumption? *IEEE Software*, IEEE, v. 33, n. 3, p. 83–89, 2015.

Pereira, R. Locating energy hotspots in source code. In: *39th International Conference on Software Engineering (ICSE 2017)*. [S.l.: s.n.], 2017. p. 88–90.

Pereira, R.; Couto, M.; Saraiva, J.; Cunha, J.; Fernandes, J. P. The influence of the java collection framework on overall energy consumption. In: ACM. *Proceedings of the 5th International Workshop on Green and Sustainable Software*. [S.l.], 2016. p. 15–21.

Petke, J. et al. Genetic improvement of software: A comprehensive survey. *IEEE Transactions on Evolutionary Computation*, v. 22, n. 3, p. 415–432, June 2018. ISSN 1941-0026.

Pinto, G.; Castor, F. Energy efficiency: A new concern for application software developers. *CACM*, ACM, New York, NY, USA, v. 60, n. 12, p. 68–75, December 2017. ISSN 0001-0782.

Pinto, G.; Castor, F. Energy efficiency: a new concern for application software developers. *Communications of the ACM*, ACM New York, NY, USA, v. 60, n. 12, p. 68–75, 2017.

Postel, J.; Reynolds, J. K. *RFC 959: File Transfer Protocol*. 1985. Obsoletes RFC0765. Updated by RFC2228 . Status: STANDARD. Disponível em: <ftp://ftp.internic.net/rfc/rfc2228.txt,ftp://ftp.internic.net/rfc/rfc765.txt,ftp://ftp.internic.net/rfc/rfc959.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc2228.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc765.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc959.txt>.

Schubert, S.; Kostic, D.; Zwaenepoel, W.; Shin, K. G. Profiling software for energy consumption. In: *2012 IEEE International Conference on Green Computing and Communications*. [S.l.: s.n.], 2012. p. 515–522.

Singh, J.; Naik, K.; Mahinthan, V. Impact of developer choices on energy consumption of software on servers. *Procedia Computer Science*, v. 62, p. 385 – 394, 2015. ISSN 1877-0509. Proceedings of the 2015 International Conference on Soft Computing and Software Engineering (SCSE'15). Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050915025582>.

Singh, J.; Naik, K.; Mahinthan, V. Impact of developer choices on energy consumption of software on servers. *Procedia Computer Science*, Elsevier, v. 62, p. 385–394, 2015.

Singh, V. K.; Dutta, K.; VanderMeer, D. Estimating the energy consumption of executing software processes. In: *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. [S.l.: s.n.], 2013. p. 94–101.

Utting, M.; Legeard, B. *Practical model-based testing: a tools approach*. [S.l.]: Elsevier, 2010.

Wheeldon, A. et al. Learning automata based energy-efficient ai hardware design for iot applications. *Philosophical Transactions of the Royal Society A*, The Royal Society Publishing, v. 378, n. 2182, p. 20190593, 2020.

Zhang, C.; Hindle, A.; German, D. M. The impact of user choice on energy consumption. *IEEE software*, IEEE, v. 31, n. 3, p. 69–75, 2014.

Zhang, P.; Sadler, C. M.; Lyon, S. A.; Martonosi, M. Hardware design experiences in zebranet. In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. [S.l.: s.n.], 2004. p. 227–238.

APPENDIX A – Resumo Estendido

Software está presente em todos os tipos de dispositivos e plataformas. Alguns programas dispõem de uma quantidade limitada de energia, devido às restrições impostas pelos ambientes onde são executados. Pesquisas mostram que aplicativos móveis que esgotam rapidamente a bateria do aparelho tendem a ser rejeitados pelos usuários (Khalid; Shihab; Nagappan; Hassan, 2015), indicando que o consumo de energia é um aspecto relevante do ponto de vista de quem está utilizando um programa. Grandes indústrias e corporações também chegaram à conclusão de que a ineficiência do consumo de energia em software pode afetar significativamente sua operação (Pinto; Castor, 2017a). Por esta razão, o consumo de energia de software tornou-se um fator importante durante o desenvolvimento, manutenção e evolução de software (Albers, 2010) (Li et al., 2016) (Singh; Naik; Mahinthan, 2015a) (Singh; Dutta; VanderMeer, 2013).

Apesar da relevância atual das análises de consumo de energia, ainda há pouco suporte para a construção de softwares energeticamente eficientes. Na verdade, os desenvolvedores ainda não sabem como produzir, avaliar e evoluir seu software considerando os custos de energia (Pinto; Castor, 2017a) (Pang; Hindle; Adams; Hassan, 2015) (Manotas; Pollock; Clause, 2014) (Pinto; Castor, 2017b). Isso se deve principalmente à ausência de abstrações e ferramentas combinadas para modelar, medir e analisar o consumo de energia(Duarte; Alves; Toresan; Maia; Silva, 2019). Assim, prover técnicas, ferramentas e processos que possam ajudar os desenvolvedores de software a entender melhor os custos de energia de software e como usar os recursos energéticos tornou-se uma grande preocupação (Pinto; Castor, 2017a). Com esse suporte, os desenvolvedores poderiam não apenas identificar os custos de execução de seus sistemas, mas também realizar análises na fase de projeto para prever possíveis ineficiências no consumo, comparar diferentes versões em termos de consumo de energia e determinar possíveis alterações para melhorar a eficiência energética.

Existem algumas ferramentas disponíveis para coletar o gasto de energia do software, como Gem5 (Binkert et al., 2011) e McPAT (Li et al., 2009), ou jRAPL (Liu; Pinto; Liu, 2015) e pyRAPL (Belgaid; d'Azémar; Fieni; Rouvoy, 2019). Pesquisas como as apresentadas em (Pereira; Couto; Saraiva; Cunha; Fernandes, 2016) são um exemplo de trabalho de análise de consumo de energia. Eles se concentraram em encontrar consumo excessivo ou anômalo de energia em software e usaram uma metodologia para otimizar programas Java e diminuir seu consumo de energia substituindo algumas estruturas de dados por alternativas mais eficientes em termos de energia. Embora essas abordagens ofereçam informações relevantes sobre o comportamento energético do software, elas não fornecem nenhum *feedback*, análise ou orientação sobre como melhorar a eficiência energética do

software, deixando essa tarefa para o desenvolvedor.

Uma maneira de realizar a análise de energia é usar modelos para obter uma representação abstrata do comportamento energético. Nesse contexto, análises de energia podem ser realizadas usando uma ferramenta de verificação de modelos, como PRISM (Kwiatkowska; Norman; Parker, 2001) ou LoTuS (Barbosa; Lima; Maia; Junior, 2017). LoTuS[1] permite a construção de Labeled Transition Systems (LTS) (Keller, 1976) com anotação de custo de energia e probabilidade usando uma interface gráfica intuitiva. No entanto, não há suporte para análises sobre o comportamento energético do modelo e o usuário ainda precisa coletar as informações de energia a serem utilizadas no modelo, e algumas questões relevantes não podem ser feitas, como o comportamento que produz o maior consumo de energia ou o custo médio de uma execução.

Embora existam algumas ferramentas capazes de prover algum suporte à análise de comportamento energético de software, é ainda uma tarefa difícil para o desenvolvedor determinar quais informações são mais relevantes e como descrevê-las na forma de propriedades a serem analisadas. Além disso, mesmo depois de descrever e executar a análise dessas propriedades, o desenvolvedor precisa interpretar os resultados para identificar qual ação – se houver – é necessária.

Neste trabalho, é proposto um conjunto de definições de propriedades de alto nível para análise de consumo de energia de software baseadas em modelos. Tamém é fornecida uma orientação sobre como realizar as análises e interpretar seus resultados para que o desenvolvedor de software possa entender como produzir ou evoluir seu software para melhorar a eficiência energética e também elaboramos um guia rápido, em "Perguntas Frequentes"(FAQ), considerando diferentes cenários e sugerindo algumas análises de propriedades para realizar em cada um. Para analisar software usando essas propriedades, modelos foram criados usando LTS, que tem uma estrutura tipo grafo, acrescido com informações sobre custo de energia e probabilidade de execução de elementos de software. Essas propriedades propostas são divididas em dois grupos: o primeiro grupo inclui as propriedades usadas para realizar análises apenas sobre custos de energia, enquanto o segundo grupo combina análises de custos de energia com informações probabilísticas para fornecer um cenário mais informativo sobre o comportamento do software em termos de energia consumo. Definir esses dois grupos é necessário porque as informações de probabilidade nem sempre estão disponíveis ou podem não ser precisas. Além disso, algumas vezes, as informações necessárias dispensam probabilidades, como identificar o comportamento de um sistema com maior custo energético.

Realizamos experimentos avaliando o comportamento energético de um sistema de um único componente, uma evolução de um sistema e, por fim, um sistema construído através de uma combinação de componentes, apresentando resultados preliminares obtidos

[1] Disponível em: http://lotus-web.herokuapp.com/

pela aplicação das propriedades propostas e seguindo as instruções do guia rápido. Para realizar os experimentos, uma implementação da cobertura de caminho denominada de *one-loop path coverage* e algumas análises de propriedades foram implementadas. Os experimentos realizados neste trabalho consistiram em (i) uma avaliação de uma implementação do protocolo SMTP, (ii) a análise de um sistema de multiplicação de matrizes extraído de (Duarte; Alves; Toresan; Maia; Silva, 2019) e (Alves; Ferreira; Duarte; Maia, 2020) e (iii) a análise do impacto da combinação de um componente de compactação de arquivos com uma implementação do protocolo FTP. A análise do consumo de energia do protocolo SMTP ajuda a entender a aplicação de análises de propriedades em um sistema de componente único existente, o experimento realizado com o sistema de multiplicação de matrizes nos ajuda a observar como analisar a evolução de um sistema de componente único e a análise do sistema de transferência de arquivos combinando a compressão de arquivos e o protocolo FTP mostra questões importantes sobre a aplicação desta abordagem para sistemas de múltiplos componentes. Os experimentos realizados mostraram como as análises dessas propriedades podem ser utilizadas para auxiliar os desenvolvedores de software na avaliação do consumo de energia de seus sistemas.

A principal contribuição deste trabalho foi a apresentação de definições de propriedades de alto nível incluindo informações de energia e/ou probabilidade para obter uma análise baseada em modelo capaz de identificar possíveis problemas no comportamento energético do software. Também demonstramos como interpretar estas propriedades e os resultados da sua análise. A implementação de análises de algumas das propriedades propostas foi feita para mostrar como a análise dessas propriedades pode ser realizada de forma automática. Por fim, a construção de um guia rápido, que indica questões comuns sobre o comportamento energético de software e quais análises de propriedades podem ser utilizadas para obter respostas para essas questões, completa o conjunto de contribuições que visam a permitir que desenvolvedores criem software mais energeticamente eficientes.