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

OSEIAS AYRES FERREIRA

Modelling Software Energy Consumption for Energy-Efficiency Analysis

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Lucio Mauro Duarte

Porto Alegre
May 2022

CIP — CATALOGING-IN-PUBLICATION

Ferreira, Oseias Ayres

Modelling Software Energy Consumption for Energy-Efficiency Analysis / Oseias Ayres Ferreira. – Porto Alegre: PPGC da UFRGS, 2022.

70 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. Behavior model. 2. Model construction. 3. Software energy consumption. I. Duarte, Lucio Mauro. II. Título.

“What you focus on, expands.“

— JOSEPH SUGARMAN

AGRADECIMENTOS

À Deus que sempre me abençoou nesta caminhada tão difícil.

Aos meus pais, Aurea Ayres Rodrigues Ferreira e Juarez Alves Ferreira, que sempre me incentivaram a me dedicar aos estudos e me apoiaram financeiramente para que este mestrado pudesse ser concluído.

À minha noiva, Juliana Gonçalves Camilo Peres, pela sua paciência no meu tempo fora em Porto Alegre para realizar o mestrado e, por todo seu amor, carinho e palavras de conforto diante a todas as dificuldades.

Ao meu orientador professor Lucio Mauro Duarte, que com sua orientação, sempre feita com grande maestria e excelência pude ir além do que jamais imaginei chegar.

Ao meu amigo e companheiro de laboratório Danilo Alves, que colaborou e incentivou diretamente neste trabalho.

À Universidade Federal do Rio Grande do Sul e ao Instituto de Informática, que proporcionaram grandes ensinamentos por meio de todos os professores altamente qualificados.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pela financiamento desta pesquisa.

ABSTRACT

Software energy consumption is becoming an essential issue during software development and evolution, in particular, due to several restrictions imposed by platforms and application requirements. However, still little support exists to aid developers to understand how some small factors can affect software energy efficiency. This mainly happens because of the absence of appropriate abstractions to model and analyse software energy behavior. This work proposes an approach to support the construction of energy behavior models from code. For this, we developed a model called ELTS (Energy Labelled Transitions System), which is a Labelled Transition System (LTS) augmented with energy information. To build this model from Java code, we created the concept of basic energy unit (BET) that enables to associate parts of the code to elements of the ELTS. With this, we aim to guarantee traceability, which enables the identification of possible hotspots of energy in the code after an analysis of the model. We represent the semantics of the code in the model through relations between BETs, namely sequence, conditional and iteration. These relations enable a better understanding of the behavior when analyzing the model and represent the connections of its composing BETs, also facilitating model construction. In addition, we propose how to locally change the abstraction level of the model. Thus, a part of the model is only expanded if necessary, keeping the rest of the model as it is. We describe three experiments to demonstrate how to model programs using our approach, including basic model composition. This modelling strategy makes it possible to improve the analysis of energy consumption, thus possibly leading to better decisions regarding software energy efficiency.

Keywords: Behavior model. Model construction. Software energy consumption.

Modelando o Consumo Energético de Software
para Análise de Eficiência Energética.

RESUMO

O consumo de energia de software está se tornando uma questão essencial durante o desenvolvimento e evolução de software, em particular, devido às várias restrições impostas pelas plataformas e requisitos de aplicativos. No entanto, ainda existe pouco suporte para ajudar os desenvolvedores a entender como alguns pequenos fatores podem afetar a eficiência energética do seu software. Isso acontece principalmente devido à ausência de abstrações apropriadas para modelar e analisar o comportamento relacionado ao consumo de energia de um software. Este trabalho propõe uma abordagem para a construção de modelos de comportamento energético a partir de código. Para isso, propomos um modelo chamado ELTS (*Energy Labeled Transition System*), que é um *Labeled Transition System* (LTS) que contém informações de custos de energia. Para construir este modelo a partir de código Java, criamos o conceito de unidade básica de energia (BET), que permite associar partes do código a elementos do ELTS. Com isso, visamos a garantir a rastreabilidade, que possibilita a identificação de pontos específicos de gasto de energia no código após uma análise do modelo. Representamos a semântica do código no modelo por meio de relações entre BETs, definidas como sequência, condicional e iteração. Essas relações possibilitam um melhor entendimento do comportamento ao analisar o modelo e representam as conexões das BETs que o compõem, facilitando também a construção do modelo. Além disso, propomos como alterar localmente o nível de abstração do modelo. Assim, uma parte do modelo só é expandida se necessário, mantendo o restante do modelo como está. Descrevemos três experimentos para demonstrar como modelar programas usando nossa abordagem, incluindo uma composição básica de modelos. Essa estratégia de modelagem possibilita melhorar a análise do consumo de energia, possivelmente levando a melhores decisões em relação à eficiência energética do software.

Palavras-chave: Modelos de Comportamento. Construção de Modelos. Consumo de energia de software..

LIST OF ABBREVIATIONS AND ACRONYMS

LTS	Labelled Transition System
ELTS	Energy Labelled Transition System
PLTS	Probabilistic Labelled Transition System
BET	Basic Energy Unit
API	Application Programming Interface
FSM	Finite State Machine
FTP	File Transfer Protocol
CPN	Coulored Petri Nets
PMC	Probabilistic Model Checking
SPL	Software Product Lines
MDP	Markov Decision Processes
PEPA	Performance Evaluation Process Algebra

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Software is present in many devices and platforms that can bring with them restrictions regarding energy consumption, such as data centers, embedded systems and mobile applications. Some of these applications may consume so much energy that batteries are quickly drained, leading to their rejection by users (KHALID; SHIHAB; NAGAPPAN; HASSAN, 2015). For this reason, energy consumption has become an important aspect to be analyzed during software development and maintenance (ALBERS, 2010; LI et al., 2016; SINGH; NAIK; MAHINTHAN, 2015).

Software energy efficiency has recently gained the attention of the research community (KHALID; SHIHAB; NAGAPPAN; HASSAN, 2015; LI; HALFOND, 2014; LIU; PINTO; LIU, 2015; PINTO; CASTOR, 2017). The work described in (PEREIRA; SARAIVA; TEC; LINCS; FERNANDES, 2016) focuses on finding excessive or anomalous energy consumption in software. They use a methodology to optimise Java programs and decrease their energy consumption by replacing some data structures by their more energy-efficient alternatives. The work presented in (SINGH; NAIK; MAHINTHAN, 2015) shows how some choices of API (Application Programming Interface) during software development can influence energy consumption, possibly saving up until 76% in common operations, such as file reading. Therefore, developers should have more knowledge about the energy consumption of their software so as to being able to modify their code to improve energy efficiency (LI; HALFOND, 2014).

In spite of the aforementioned work, among others that can be found in the literature, energy-consumption analysis has still little support, which makes it difficult to produce and evolve systems based on energy costs. This happens, essentially, because of the absence of software abstractions and tools (PINTO; CASTOR, 2017). It is even worse when systems size and complexity increase, which may prevent the identification of problems and potential improvements.

One possible way of analysing energy costs, whilst providing more support for program understanding, is using behaviour models (UCHITEL; KRAMER; MAGEE, 2003). Using models, it is possible to analyze different components with different levels of abstraction and to check whether properties of the system satisfy some requirements. Models also allow for a more easily comparison of two versions of a system in case of possible changes, even before implementation. In addition, behavior models may be used to document the system's behaviour, which can be used for many other analyses. All these

possible analyses on an abstract model would be difficult to be carried out directly on the source code due to its complexity (LUDEWIG, 2004; CORBETT et al., 2000).

Behaviour models have been used to abstract and analyze software energy consumption in the work by Duarte, Alves, Maia e Silva (2019) and in the research by Baier, Dubslaff, Klein, Klüppelholz e Wunderlich (2014). However, the authors, in both cases, do not indicate how to build such a model from the source code that represents the actual energy behaviour of the system. Knowing how to construct an intended model is essential to guarantee more reliable results, as any analysis on a model that does not represent the real behaviour of a system may be misleading (JACKSON; RINARD, 2000). Hence, an appropriate representation of the behaviour of a system in terms of energy consumption can directly affect the results of any analysis on this model. For this reason, there should be some guidance on how to properly represent code elements as model constructs and their corresponding energy costs.

1.1 Problem Definition

The model-based framework for analysing software energy consumption proposed in (DUARTE; ALVES; MAIA; SILVA, 2019) consists on collecting energy information using one of the available tools (LIU; PINTO; LIU, 2015; LI et al., 2009; BINKERT; BECKMANN; AL., 2011), adding this information to a Labelled Transition System (LTS) (KELLER, 1976) and analysing properties of interest using an existing tool. The framework is composed of four phases: i) behaviour model specification; (ii) energy consumption measurement; (iii) model annotation; (iv) energy-based property verification.

At the first step of the framework, a model must be constructed. The authors associate an energy cost to each transition of the LTS. Thus, a transition in the LTS model contains a label with the name of a program element and the energy information to represent the energy cost of executing that specific element. To model and visualise the LTS, the authors used a tool called LoTus (BARBOSA; LIMA; MAIA; COSTA, 2017), which has been extended to include the representation and analysis of energy information. However, there is no definition on how to carry out this model construction with energy costs, being the quality of the model essential for the subsequent steps of the framework.

The model construction is made in an *ad hoc* way and there is no guidance for the software developers. Hence, each developer creates their own model, according to their knowledge and defining what is relevant to be a modeled. According to the developer's

choices, the energy modeling may have inconsistencies with the real energy behavior. Consequently, the produced model may generate an inaccurate representation, affecting energy analyses based on it. Therefore, to reach reliable results, we need to construct an accurate model; i.e., a closer representation of the real code, on the way that the results are trustworthy to be analyzed and indicate actual hotspots of energy consumption. To do so, it is necessary to provide directions to developers on how to build these models and use them to support better decisions to achieve energy efficiency.

Without some guidance and a structured way of building a model, there are many problems that can occur and prevent the adequate representation of the software energy behavior, in particular related to the developer's lack of experience with models and/or energy behavior. These problems affect the analyses and, thus, should be avoided. Some work, such as (ALVES; FERREIRA; DUARTE; SILVA; MAIA, 2020), (DUARTE; ALVES; MAIA; SILVA, 2019) and (ALVES; FERREIRA; DUARTE; MAIA, 2020), highlight the need for steps to construct an energy behavior model to enhance energy-based property verification. Based on the experiments conducted in these previous studies, we have identified some issues that can affect analysis results:

1. **Inclusion of unnecessary information.** A software developer may include behaviours not needed for the analyses or that do not add much information. Similarly, developers may not model important behaviours, which can lead to incorrect results during model analysis;
2. **Traceability between model and code may be hard to guarantee.** A developer may construct a model that does not represent a precise relation between code and model. It is crucial for any analysis to understand exactly to which code element each model element corresponds. A model that does not correspond to the source code may lead, for example, to the possible identification of false hotspots of energy;
3. **Lack of a semantic relation between model elements and source code structures.** The semantics of the code is essential to its behavior representation. If the semantics is not considered, the corresponding model would be only a sequence of actions, ignoring points of choice and iteration, which does not represent the flow of execution of the code. Depending on what is executed, the overall energy consumption can be affected, thus modelling the correct semantics has an important effect on the energy behavior representation;
4. **Impossibility of changing the level of abstraction of specific parts of the model,**

if necessary. The level of abstraction of the model can facilitate the interpretation of data, which makes it easier to understand the energy behavior looking in detail statements that may possibly affect energy consumption as a whole. In addition, there should be a way of controlling multiple levels of abstraction using the same model, what would make it possible to analyse only a specific part of the model. This way, the developer would not have to change the level of abstraction of the whole model, but could just investigate a part of interest.

As mentioned before, all these problems happen, essentially, due the absence of some guidance on how to map source code to an energy behavior model. To overcome these problems, this mapping should guarantee the inclusion of all necessary information and traceability between model and code, consider the source code semantics and allow local change of level of abstraction. Hence, given all the difficulties related to building an accurate behaviour model with energy costs, our research question is: *How to allow even developers with little or no previous experience with models to build a model from an existing source code which is as close as possible to the software's energy behavior?*

1.2 Proposed Approach and Our Contributions

This work describes an approach to guide a software developer, even with little or no knowledge of modelling or energy behavior, to construct a behavior model with energy cost information by defining *basic energy units (BETs)* to represent parts of the code and their respective energy consumption. In addition, we determine a way to described relations between these basic energy units, where these relations are used to represent the semantics of the code. Moreover, traceability between code and model is easily achieved, allowing that potential problems identified during analysis of the model can be traced back to the corresponding element of the code, so that appropriated changes can be carried out. Our modelling approach also offers the possibility of changing the level of abstraction of parts of the model, supporting a better understanding of the energy behavior of the system.

We conducted 3 experiments to demonstrate our approach. The two first experiments model isolated components, one to send a file using the FTP protocol and another that focuses on a file compressor. In the last experiment, we combined these two models and evaluate the impact on energy costs. In all these experiments, we applied the step-by-

step proposed in our approach, which enables us to show how to construct a model from source code. The constructed models were useful to indicate the main energy hostposts, allow traceability to the code and enable easy change of local abstraction level.

The main contributions of this work are:

- Demonstration of how to build a model with energy costs step-by-step based on the definition of basic energy units (BETs) and their relations, consequently, generating a model that describes as close as possible the real energy behavior of the system;
- Traceability between model elements and their respective constructs in the source code, making it possible to easily identify which part of the software needs to be modified to improve energy efficiency;
- Possible change of abstraction level of a part of the model to better understand the energy consumption of the corresponding part of the code, identifying possible points of interest and where to modify the code to improve energy efficiency.

1.3 Outline

The remainder of this dissertation is organized as follow. In Chapter 2, we present the background theory about behaviour models, discussing tools and models to work with software energy consumption. In Chapter 3, we discuss our approach to build behaviour models with energy costs using different levels of abstraction. We evaluate and discuss the model constructions presenting the results of some experiments in Chapter 4. Finally, we discuss the related work in Chapter 5 and we present conclusions and possible future work in Chapter 6.

2 BACKGROUND

This chapter presents the background theory. Firstly, we describe the main tools used to collect energy information, and next, the concepts related to behavior models and how they can be used to model systems considering energy costs. The last part contains background information about software energy analyses.

2.1 Collecting Energy Information

Nowadays, to aid the developers to collect the software energy consumption of their systems, there exist some tools such as Gem5 (BINKERT; BECKMANN; AL., 2011), McPAT (LI et al., 2009), jRAPL (LIU; PINTO; LIU, 2015), and pyRAPL [1]. These tools return quantitative values that describe the consumption of a part of the system or that of the entire program.

Gem5 is an architecture simulator that enables the software developer to measure the energy cost of a running software. It works with Python and C++ programming languages. The McPAT tool is a framework used to simulate an execution, using abstractions to estimate the energy consumption. It uses an XML-based interface to configure and specify the target clock frequency, the area, and power deviation of the optimization function and other architectural/circuit/technology parameters. Both tools work on the energy consumption of the entire program.

Tools such as jRAPL and pyRAPL enable the developers to annotate on the source code which parts they would like to measure energy costs. Hence, it is possible to choose the point of interest and determine how much energy a code segment is spending. The jRAPL works exclusively with Java programs and some architectures, and pyRAPL works with Python programs and also only for a specific architecture. These two tools are simpler to use compared with Gem5 and McPAT, since it is not necessary to specify low-level parameters, only to determine which lines to collect the energy information from.

Each of these tools mentioned depends on the desired application. If it is necessary to measure the consumption of the system as a whole, tools such as Gem5 and McPAT support this goal. However, in case the software developer needs to obtain more details on how much each part of the code is consuming, jRAPL and pyRAPL are better options.

Using the available tools, it is possible to obtain the energy consumption of the

[1] https://pyrapl.readthedocs.io/en/latest/.

software and to provide to software developers a way to identify possible hotspots. The tools only show how much is consumed, hence it is the software developer who has to decide what should be done. Therefore, simply collecting the energy information may not be enough, becoming crucial to use other artifacts to understand the energy behavior of the system, and to visualize it in a clear way how some evolution/refactoring could affect the consumption. One possibility of doing this is by using behavior models.

2.2 Behavior Models

A behavior model is an abstract representation used to describe the expected behavior of a system. Therefore, it is possible to a software developer to construct a model leaving out unnecessary information and focusing only on important issues to facilitate the analysis of a system. The level of abstraction can be changed by including or excluding some information, depending on the modeling purpose. Using behavior models, it is possible to accomplish some analysis that would be difficult or impossible directly on the source code, possibly reducing future costs of fixing the system later on (UCHITEL; KRAMER; MAGEE, 2003).

Generally, the way of modelling behaviours is through finite-state machines. A finite-state machine (FSM) is composed by a finite set of states $Q = \{q_0, q_1, ..., q_n\}$, where each q_i state, for $0 \leq i \leq n$, represents a set of possible concrete states $S = \{s_1, s2, ..., s_n\}$ of a system, and a set T of transitions connecting these states. For example, a transition $t \in T$ from a state q_0 to a state q_1 may represent a transition from a set of states S_1 to a set of states S_2 in the actual program. FSMs have mathematical foundations that enable the analysis of several properties of a system (CLARKE; WING, 1996).

FSMs are the basis for multiple models, such as StateCharts (HAREL, 1987), UML state diagram (SEIDL, 2015) and CFG (BöHM; JACOPINI, 1966), which can be used to model the behaviour of a system. These models have the advantage of having a graph-like structure, thus enabling the use of known graph-related algorithms to analyse software behavior. Behaviour models based on state machines can be used not only for the design and analysis of systems, but also during maintenance and evolution to enable a better visualization of which parts of the system will be modified to obtain the desired behaviour.

One model based on FSMs that is commonly used to model behaviour is Labelled Transition System (LTS) (KELLER, 1976). An LTS is a type of action-based model

(HANSEN; VIRTANEN; VALMARI, 2003) whose behaviour is given by the sequence of actions that the system executes, where each transition of the model is associated with one action.

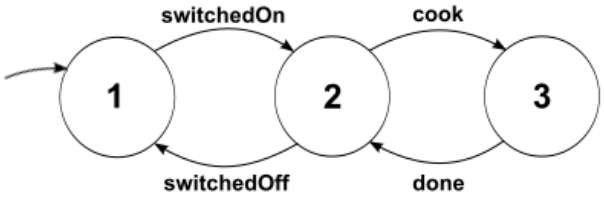
Definition 2.2.1 (*Labelled Transition System.*)**.** A *Labelled Transition System (LTS)* $M = (S, s_i, \Sigma, T)$ is a model where:

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

Given two states $s_0, s_1 \in S$ and an action $a \in \Sigma$, then $s_0 \xrightarrow{a} s_1$ describes a transition from state s_0 to state s_1 through the execution of an action with name a. Thus, when an action occurs, a new transition is triggered, changing the current state.

An example using an LTS model is presented in 2.1. It models a microwave oven system where actions *switchedOff* and *swicthedOn* are used to symbolized when the system is on or off, respectively, while action *cook* indicates that some food is being cooked and action *done* represents that the cooking process is finished. The leftmost transition, without a label, points to the initial state of the model.

Figure 2.1: A LTS model that represents a microwave oven.



Adapted from: (DUARTE, 2007).

The states in the LTS model represent stopping points while the next event does not occur. A transition indicates the occurrence of an event (action) and the change from one state to another. Hence, the behavior is given by the sequence of occurred actions.

An advantage of using an LTS is the possibility of modeling concurrent and distributed behaviours and analysing systems with these behaviours. This is possible using some type of parallel composition (MILNER, 1989).

The abstraction level is related to the amount of information represented by the model. This information can be added or removed with the goal of enhancing the modeling representation. The change of abstraction level can be done in states or transitions,

where the information modeled by these elements can be subdivided, creating new states/-transitions. If we decrease the abstraction level, more information is added, which makes the model larger. On the other hand, when we increase the abstraction level, we reduce the information represented, creating a more compact model. Therefore, it is important to analyze which is the best level of abstraction to represent our system, because a model with too much details can be large and difficult to analyze, and a too abstract model can miss relevant information.

2.3 Software energy consumption analysis

Some studies, such as (DAYARATHNA; WEN; FAN, 2016) and (SINGH; NAIK; MAHINTHAN, 2015), analyze how some decisions in design and coding stages may generate a relevant software energy consumption, indicating that some small changes can be performed to improve energy efficiency, and thus, achieve high performance with lower costs. In (SINGH; NAIK; MAHINTHAN, 2015) they show how the increase of the amount of data dealt with in a server-side impacts the software energy consumption and how not only hardware setups can bring benefits, but also, software decisions contribute to energy costs. These studies show how some changes in the source code can impact energy consumption. However, it is still unclear to software developers how to produce, evaluate and evolve their software considering energy costs.

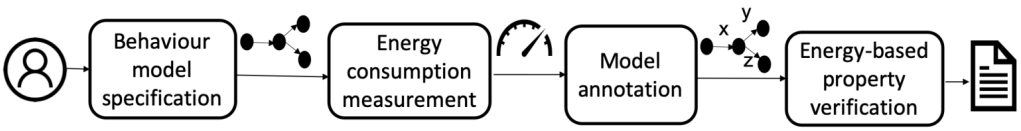
Duarte, Alves, Maia e Silva (2019) suggested a way to collect and analyze the software energy consumption, where the energy cost is collected using some tool, such as those mentioned in Chapter 2.1, and inserted into an LTS model, associating an energy cost value to each transition. Figure 2.2 shows the flow of each phase proposed by the framework. They used the constructed model to understand where an energy bottleneck could be. This approach consists of four phases:

1. **Behavior model specification:** They propose to associate each transition of the LTS with a code element, where each element has an energy cost. Hence, a path of execution would be a sequence of transitions showing the behavior of the system with respect to energy consumption. The total cost of a path would be the sum of the costs of its composing transitions. Model construction could be done manually or using some model extraction approach as proposed in (DUARTE; KRAMER; UCHITEL, 2017), (WALKINSHAW; TAYLOR; DERRICK, 2013) or (MARIANI;

PEZZè; SANTORO, 2017);

2. **Energy consumption measurement:** This phase indicates the use of a tool that enables annotating directly the source code to select the desired elements to collect the energy costs to be used in the model;
3. **Model annotation:** With energy costs collected in the previous phase, it is possible to annotate the model with these values, associating each transition with its energy cost. From this, we have an LTS model with transitions labelled with values representing energy costs;
4. **Energy-based property verification:** The last phase comprises carrying out analyses on the model constructed in the previous phases, such as the most costly path of execution, list of executions within a threshold of consumption, the average execution cost, among others.

Figure 2.2: The flow of model-based framework.



Source: (DUARTE; ALVES; MAIA; SILVA, 2019)

Based on the framework proposed by (DUARTE; ALVES; MAIA; SILVA, 2019), work such as (ALVES; FERREIRA; DUARTE; SILVA; MAIA, 2020) and (ALVES; FERREIRA; DUARTE; MAIA, 2020) were developed using LTS models annotated with energy costs. In (ALVES; FERREIRA; DUARTE; SILVA; MAIA, 2020), it is modelled the energy consumption of Java implementations of the Bubble Sort, Insertion Sort and Selection Sort algorithms and performed a comparison of these different algorithms in terms of energy costs. This analysis showed how to combine LTS and energy consumption to help developers produce energy-efficient software.

In the research of Alves, Ferreira, Duarte e Maia (2020) it is proposed a set of properties mixing the energy consumption with probabilities of execution. Depending on the probability of a component occurring, the energy costs may be irrelevant if the probability is very small.

These works above show the usability of LTS to model the energetic behavior of a system. However, all this research highlights how the appropriate construction of a model to describe the energy consumption is of paramount importance, because a misleading

analysis might indicate false hotspots and lead to unnecessary changes.

The framework does not specify how to construct the LTS model with energy information; i.e, the software developer has no guidance nor a structured way of building a model. Without this support, there are many problems that can occur and prevent the adequate representation of the software energy behavior (for instance, the ones mentioned in 1.1). These problems affect the analyses and, thus, should be avoided. For this reason, in this work we address how to construct a behavior model based on LTS that includes energy information. We concentrate on phases 1 and 3 of the framework, considering energy information gathering (phase 2), with the aim of producing a better model for energy-based analyses (phase 4).

3 MODELLING SOFTWARE ENERGY CONSUMPTION

In the work by Duarte, Alves, Maia e Silva (2019), the authors propose a model-based framework to analyse software energy consumption. The framework is composed of four phases: (i) Behaviour model specification; (ii) Energy consumption measurement; (iii) Model annotation; and (iv) Energy-based property verification. Hence, the goal of the framework is to enable model-based analysis of energy consumption. For the analysis of software energy consumption, the authors suggest using one of the available tools to collect energy information, such as (LIU; PINTO; LIU, 2015), (LI et al., 2009) and (BINKERT; BECKMANN; AL., 2011), and then adding this information to an LTS model.

To build this model, the authors propose to associate an energy cost to every transition of the LTS. Thus, each transition in this LTS model contains a label with the name of an action, representing the execution of an element of the code, and a label with the energy cost of executing that specific action. Hence, the proposed idea is simply associating an energy cost to each transition. However, there is no guidance on how to construct an LTS model from the source code considering its association with energy information.

Creating a model that adequately represents the energy behavior of the code guarantees that results of analyses on the model can be assumed to reflect an analysis on the actual code. Moreover, when mapping code to model, it is essential to keep track of which element of the code corresponds to each element of the model. Without this clear relation, a developer could have difficulties identifying the actual points of possible energy bottlenecks or hotspots after some analysis of the model.

Towards providing support for model-based energy analyses, we focus specifically in the behaviour model specification and model annotation phases of the framework proposed by Duarte, Alves, Maia e Silva (2019), but also involving the energy consumption measurement and with the goal to use the produced model in the energy-based property verification phase. We propose a step-by-step approach to model systems using an LTS model augmented with energy information, denominated in this work *Energy Labelled Transition System (ELTS)*. We introduce a mapping from elements of the code to elements of the model to create an appropriate representation of energy behavior of the code, which also enables analyses on the model using different levels of abstraction. This mapping between code and model and the use of different levels of abstraction allow the identification of points of interest in the model and to trace it back to the corresponding

elements in the source code. Thus, a developer can understand which parts of the code could be improved in terms of software energy consumption and where to locate them.

An overview of the approach is presented in Figure 3.1[1]. The gray rectangles represent the requirements of a step. This approach consists of four steps: i) Definition of energy elements; (ii) Energy cost gathering; (iii) Model construction; (iv) Model analysis. The first step of the approach is to define the basic elements of the model, which we call *basic energy units (BET)*. A BET is used to refer, in a generic way, to a code element which spends a certain amount of energy to execute. Hence, BETs constitute the building blocks of an ELTS and are based on specific characteristics of the particular programming language used. With the identification of the necessary energy elements, a tool should be used to collect the energy information corresponding to each one of them. Having the energy elements and their respective energy costs, an ELTS is built following some rules based on a set of relations between energy elements in the code. After the model has been built, a developer can carry out the model analysis step. In this step, a model evaluation is executed aiming to identify possible energy hotspots or to check energy properties. In the case where there is any part of the model that needs to be further investigated, it is possible to change the level of abstraction of that part of interest by repeating the process considering only the specific part of the model. The process ends when the developer determines that no other analyses are required or when it is no longer possible to change the abstraction level.

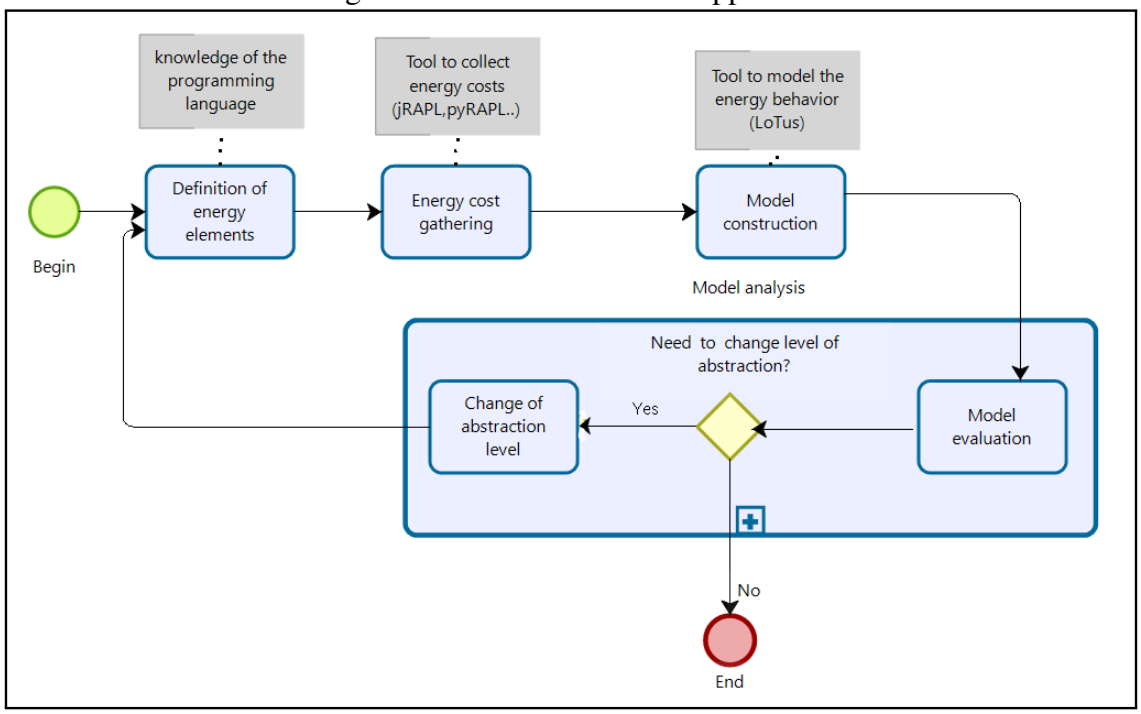
All these steps are presented in more detail next. We use the source code in Figure 3.2 as a running example while explaining each step. This program enables the user to type in a sentence and checks whether there are more than 10 words in the sentence, in which case the count stops. This example is only to show the basic ideas of our step-by-step approach to build an energy behavior model.

3.1 Step 1: Definition of Energy Units

As an input to this step, it is necessary to have the source code, from which the developer will construct the model. To perform an accurate model construction we propose a generic way to refer to a code element denominated *basic energy unit (BET)*, which enables the software developer to identify the points of interest to collect and model the

[1]The signal "+" was placed by the modeling tool used to create the flowchart and it indicates the existence of a subprocess (in this case, within the model analysis step).

Figure 3.1: Flow chart of our approach.



Source: Author.

Figure 3.2: Running Example.

```
1  Scanner text = new Scanner(System.in);
2  System.out.print("Enter with a text: ");
3  if(text != null){
4       BufferedWriter buffWrite = new BufferedWriter();
5       while(text.hasNext()){
6            buffWrite.append(text+"\n");
7       }
8  }
9  else{
10      System.out.print("Empty string");
11 }
12 text.close();
```

Source: Author.

energy consumption of their system. Thus, the purpose of using BETs is to help identify possible hotspots of energy costs, keeping a clear mapping from code elements to model elements.

A BET is defined as a sequence of one or more consecutive lines of a program code, such that a line cannot belong to more than one sequence (i.e., there is no overlap of BETs). Moreover, to identify BETs on the source code, we must consider the semantics of the programming language used to build the code. This is necessary because the seman-

tics of the language is directly linked to the energy behaviour of the program. We cannot, for example, divide blocks of code such as selection and repetition statements, blocks of statements, etc., which need to be analysed as a single BET. Otherwise, we could be altering the semantics of the code behaviour and, consequently, its energy-related behaviour described in the model. Note that not every line needs to be part of a BET, which means that the developer may choose to leave parts of the code out of the model, such as lines with low cost or of no interest, thus simplifying the construction and analysis.

To illustrate the use of a BET, consider the program bellow:

```
1 int a = 1;
2 a ++;
3 if(a == 2) {
4    print("a␣is␣equal␣to␣2");
5    a = a − 1; }
```

Initially, we could consider the whole program as a unique BET, from line 1 to line 5, which is the more abstract model. Another possible way of dividing this code into BETs would be to define a BET $B1$ considering the statements between lines 1 and 2 and a BET $B2$, including lines 3 to 5, or any other division of the set of lines. Note, however, that we could not define a BET $B1$ involving lines 1 to 3 and another BET $B2$ including lines 4 and 5, as this would mean splitting the parts of the selection statement of lines 3 to 5, thus violating the semantics of the code.

When creating a model, it is desirable that it is the most abstract possible representation of the system behavior, so that analyses on it are easier to carry out and visualization is facilitated. The same applies when defining BETs. When a BET is abstract enough to be subdivided into other smaller BETs, we say it is a *composite BET* (i.e., BETs involving more than one line of code). BETs containing only one independent line of code are denominated *atomic BETs* because they can not be subdivided. For instance, in the previous example, a BET involving line 1 would define an atomic BET, as it contains only a single line of code, whereas a BET involving lines 1 and 2 would be a composite BET, since we could then split it into two separate atomic BETs. Note, that lines 3 to 5 necessarily define a composite BET, as lines 4 and 5 depend on line 3 to execute.

Applying the idea of BETs to our running example of Figure 3.2, we could divide it into the BETs presented in Table 3.1. This table exemplifies the definition of BETs following all the presented rules.

Table 3.1: Table with BETs of the running example.

BET	Lines
B1	1-2
B2	3-11
B3	12

3.2 Step 2: Energy Cost Gathering

Given the BETs selected in the previous step, we now need to collect their respective energy costs. To accomplish it, we need a tool (e.g., jRAPL, pyRAPL, Gem5 or MCPAT) that can measure energy costs according to the level of abstraction of the defined BETs. In this work, we concentrate only on the model construction, hence it is the user's responsibility to choose the adequate tool to obtain energy costs considering the desired level of abstraction.

To associate a given BET to its respective cost, we define a cost function where, given a BET, it returns the energy consumed to execute the part of the code represented by the BET. More formally, this function is defined as:

Definition 3.2.1 (*BET cost function* δ.)**.** Let be a program $Prog$ and a BET $b \in Prog$. The BET cost function is defined as $\delta : b \rightarrow n$, where $n \in \mathbb{R}$ is the cost of executing the portion of the code represented by b.

In practice, the implementation of this function depends on the tool used to collect the software energy consumption. Hence, for each execution, the value of energy cost may vary and the developer should use a more representative value, such as the average cost, to guarantee only one cost n is associated to each BET b in a given program. The unit in which n is measured also depends on the used tool and should be the same for all costs pertaining to the same model.

Table 3.2 shows an example of associating energy costs to BETs considering the code in Figure 3.2. These values are fictitious and only used as part of the example to present a complete description of the model construction.

Table 3.2: Table with BETs of the running example with energy costs.

BET	Lines	Cost
B1	1-2	2
B2	3-11	8
B3	12	1

3.3 Step 3: Model Construction

Having defined the BETs and collected their respective energy costs, then we can build the model. The model can be constructed manually or by using some visual tool, such as LoTuS (BARBOSA; LIMA; MAIA; COSTA, 2017). To construct this model, we use the same basic concepts as the original definition of LTS presented by Keller (1976) and Milner (1999). However, we define a guideline to build LTS models focusing on better representing the energy behaviour of a system. For this reason, we have extended the standard LTS to an *ELTS (Energy Labelled Transition System)*. In this ELTS model, each BET is mapped to a transition and its cost is used to label this transition according to the BET cost function δ. States only represent a boundary between BETs. Hence, the selection of BETs and their costs impact directly the model. Formally, an ELTS can be defined as follows:

Definition 3.3.1 (*Energy Labelled Transition System.*)**.** An *Energy Labelled Transition System (ELTS)* $M = (S, s_i, \Sigma, T, \delta)$ is a model where:

- S is a finite set of states;
- $s_i \in S$ represents the initial state;
- Σ is an alphabet (set of BETs);
- $T \subseteq S \times \Sigma \times S$ is a transition relation;
- δ: is the BET cost function from Definition 3.2.1, associating an energy cost to each element of Σ.

Given two states $s_0, s_1 \in S$, a BET $b \in \Sigma$ and $\delta(b) = x$, then $s_0 \overset{b\{x\}}{\rightarrow} s_1$ describes a transition from state s_0 to state s_1 through the execution of BET b with energy cost x. Thus, when the system is in state s_0 and BET b is executed, the transition to s_1 is triggered with the cost of x units of energy.

Using these definitions, we propose a guideline for modeling software energy behaviour, where we establish a mapping between a source code and an ELTS model. At a first sight, the ELTS could be seen as simply a sequence of BETs. However, this approach cannot accurately represent the richer semantics and structure of a source code regarding software energy behavior, such as points of choice and iterations. For this reason, we define types of relations between BETs to reflect these elements. These definitions are similar to the basic algorithmic structures used in programming languages (MICHAELSON, 1990).

In this work, these structures will be used to model energy behaviour and to indicate how to analyze energy information. Thus, the use of these relation definitions enables the representation of the semantics and basic structures of the source code. We classify relations between BETs into three types: sequence relation, conditional relation and iteration relation. Next, we detail each one of them.

3.3.1 Sequence Relation

In a *sequence relation*, a given BET $B2$ can only occur after a previous BET $B1$ has been executed. However, they execute independently, which means the result of executing $B1$ does not prevent $B2$ from occurring. The total energy cost of a sequence of BETs is represented by Equation 3.1, where $\delta(Bi)$ is the cost of executing BET Bi. Therefore, if there is a sequence of BETs $\langle B1\ B2\ B3\rangle$, then the total cost of this sequence would be $\delta seq(B1, B2, B3) = \delta(B1) + \delta(B2) + \delta(B3)$.

$$\delta seq(B1, ..., Bn) = \sum_{i=1}^{n} \delta(Bi) \tag{3.1}$$

In some cases, a sequence of BETs $B1$ and $B2$ can be seen as a single, more abstract BET B, where $\delta(B) = \delta seq(B1, B2)$. This may be useful when a developer can abstract a part of the model which is not relevant for some analysis, thus, reducing the cost of running such analysis.

Figure 3.3a presents a code example with method calls a, b, c and d in a sequence. In this example, we consider each method call as a separate BET and transform each one of them into a transition with its respective energy cost (step 2 of the diagram presented in Figure 3.1). Line 1 maps to a transition representing method call *a()*, identified as BET $B1$ in the model with its respective energy cost. Similarly, lines 2, 3 and 4 are mapped to BETs $B2$, $B3$ and $B4$, respectively. With all the BETs mapped, we build the ELTS model presented in Figure 3.3b. Each BET is represented by a transition in the model and their respective energy costs are indicated between braces (e.g., the transition between states 0 and 1 corresponds to BET $B1$, which has an associated cost of x). The total energy cost for this sequence is $\delta seq(B1, B2, B3, B4) = x + y + t + z$.

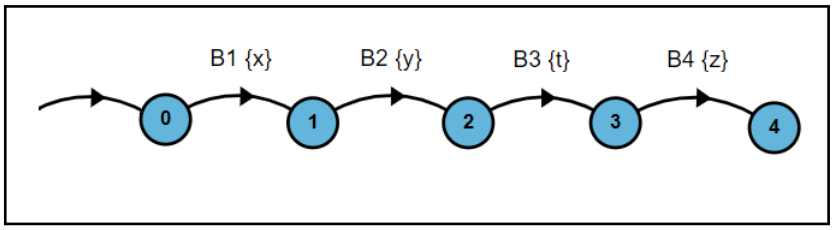
As mentioned before, BETs $B1, B2, B3$ and $B4$ could be modelled as a unique, composite BET, as presented in Figure 3.3c. This model contains only BET $B1$, whose cost is $w = x + y + t + z$.

Figure 3.3: Sequence Relation between methods.

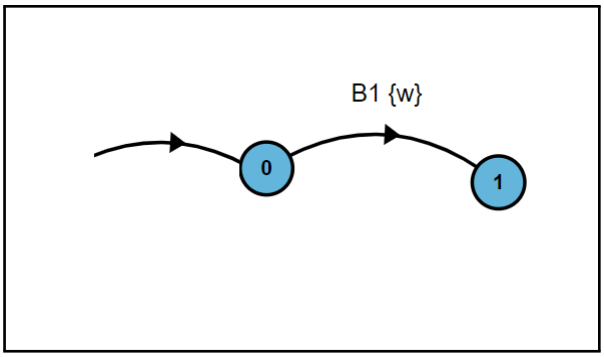
(a)

```
1 a();  // B1
2 b();  // B2
3 c();  // B3
4 d();  // B4
```

(b)



(c)



Source: Author.

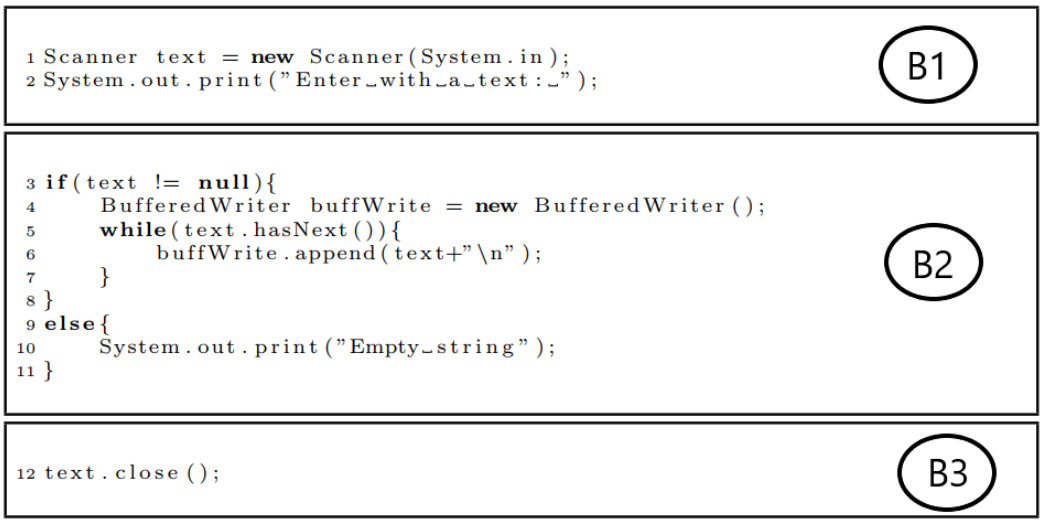
We apply the idea of sequence relations to our running example, as presented in Figure 3.4. In this example, one can note the separation of BETs, following the definition presented in Chapter 3.1. In principle, we could have considered each line as a separate BET. However, the corresponding ELTS would be larger and would not follow the requirements of a BET, for example, breaking apart the selection statement between lines 3 and 11. Hence, we chose these BETs such that we do not break the selection statement. To simplify the model, we defined a composite BET containing both lines 1 and 2 (if necessary, later we could break it into two atomic BETs). The respective ELTS model is shown in Figure 3.5. BET $B1$ represents lines 1 and 2, whilst $B2$ is used to represent lines 3 to 11, and finally, $B3$ maps line 12.

Using this ELTS, we can build a model such that traceability between source code and model can be maintained. This can be done through BETs, which are parts of code mapped to model transitions. Therefore, identifying the a relevant transition of the model

and its respective BETs, it is possible to determine which part of the code is represented by that BET, thus indicating where modifications should take place, if necessary.

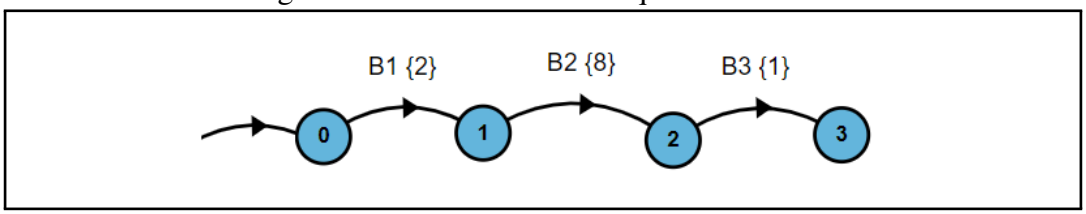
Note that the code contains other types of relations, but here we focus only on sequence relations as a first step. Next, we will detail the other relations and how to use them.

Figure 3.4: Running Example with Sequence Relation.



Source: Author.

Figure 3.5: ELTS model of sequence relation.



Source: Author.

3.3.2 Conditional Relation

A *conditional relation* is defined when a set of BETs may occur as the next step of an execution depending on the evaluation of a given expression. The evaluation of this expression enables the execution of only one of the BETs in the set. For example, suppose a conditional relation between two BETs $B1$ and $B2$: in case the expression evaluation enables the execution of $B1$, then $B2$ is not executed; if $B2$ is enabled, then $B1$ is not executed.

The energy cost of a conditional relation of BETs is represented by the equation

3.2, where $\delta(Bi)$ is the cost of executing BET Bi, e is the expression evaluated and x and z are possible valuations of e. Therefore, the total cost of executing this part of the code would be given by a function defining an association between each possible value of an expression e and the cost of executing the corresponding BET, as presented in Equation 3.2.

$$\delta cond(B1, .., Bn) = \begin{cases} \delta(B1) & \text{if } e = x \\ ... & \text{if ...} \\ \delta(Bn) & \text{if } e = z \end{cases} \tag{3.2}$$

$$\delta cond(B1, .., Bn, e) = \delta(e) + \begin{cases} \delta(B1) & \text{if } e = x \\ ... & \text{if ...} \\ \delta(Bn) & \text{if } e = z \end{cases} \tag{3.3}$$

In case the software developer needs to consider the energy costs of the conditional test, equation 3.3 should be used, which is the same equation as 3.2 but with the addition of cost of the conditional test.

An example of conditional relation is a selection statement $if - else$, where only one of the alternative blocks of code will be executed depending on the condition tested at the beginning of the statement. In the model, a conditional relation is represented by multiple outgoing transitions from the a source state leading to the same target state.

Figure 3.6a shows an example of this relation, where BET $B1$ is represented by lines 1-3 and BET $B2$ refers to lines 4-6. Hence, depending on the evaluation of $condition$, either $B1$ or $B2$ will executed and, thus, produce the corresponding energy cost. In this example, no matter which BET executes, the code in line 7 will be executed next.

If there is a condition with only one BET to be executed (e.g, when exists an if statement without an $else$ statement), we introduce an empty transition (a transition labelled with a symbol ε and with cost 0) to show the possibility of not executing the associated BET. Hence, this representation describes a choice of executing a certain BET or not depending on the evaluated condition.

Figure 3.6b presents the ELTS model built from the code of Figure 3.6a, where, from state 0, only one of the outgoing transitions (representing BETs $B1$ and $B2$) is executed, leading to state 1. The transition labelled with $B3$ represents the BET involving a set of lines starting in line 7 of Figure 3.6a. Figure 3.6c represents the situation when there exists only one block of commands that can be executed depending on a condition.

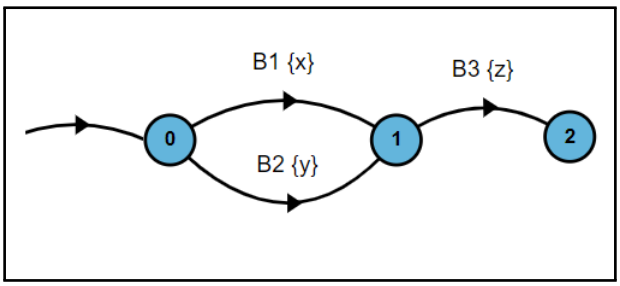
If the block is not executed, it is used the label ε with costs zero to represent this situation, where nothing is executed.

Figure 3.6: Conditional Relation Examples

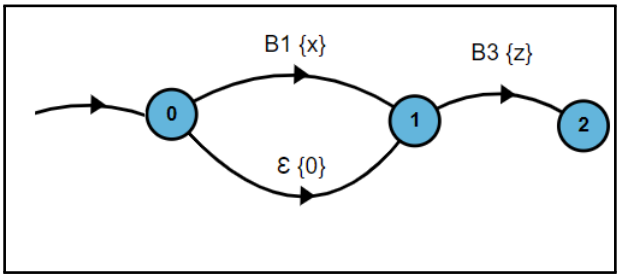
(a)

```
1 if(condition){
2       ...
3 }
4 else {
5       ...
6 }
7 ...
```

(b)



(c)



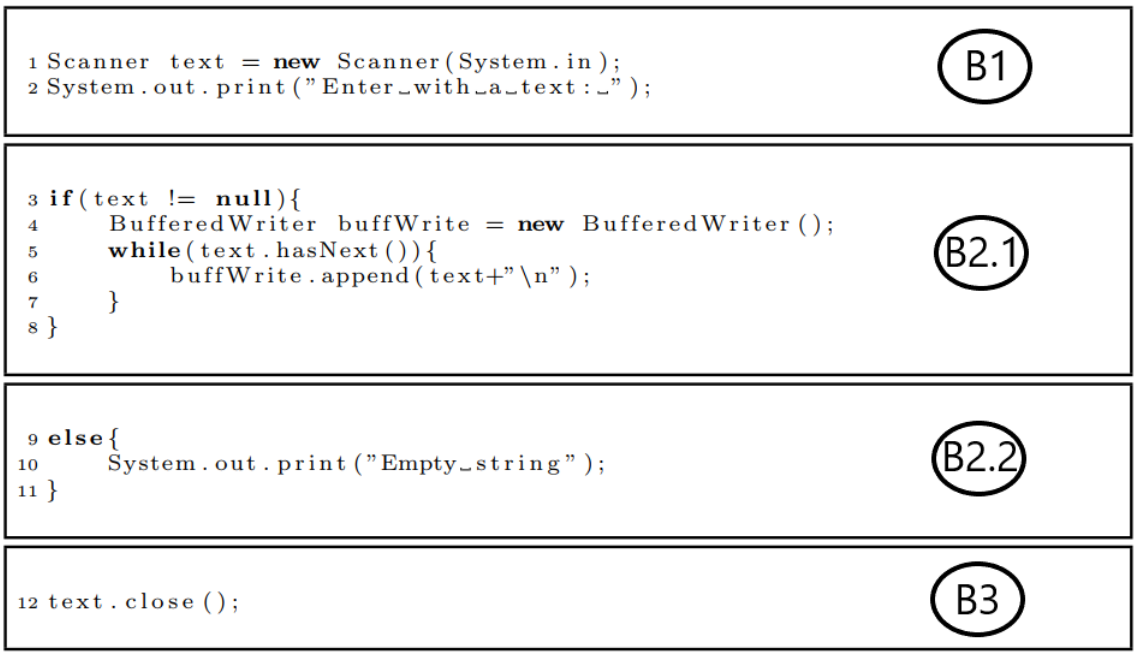
Source: Author.

An instance of a conditional relation using our running example is presented in Figure 3.7 between lines 3 to 8 and 9 to 11. In this example, the remaining BETs do not change, therefore, the only modification is the newly identified relation. Thus, the updated ELTS model representing our running example is expanded only in the transition referring to $B2$, as shown in Figure 3.8. In this case, BET $B2.1$ will only be executed if the condition is true; otherwise, $B2.2$ will be executed.

In the model represented by Figure 3.5, we had considered only the execution of BET $B2.1$, assuming it was the most probable option. Possibly a better way to represent this situation would be to identify the probabilities of execution and to consider only the

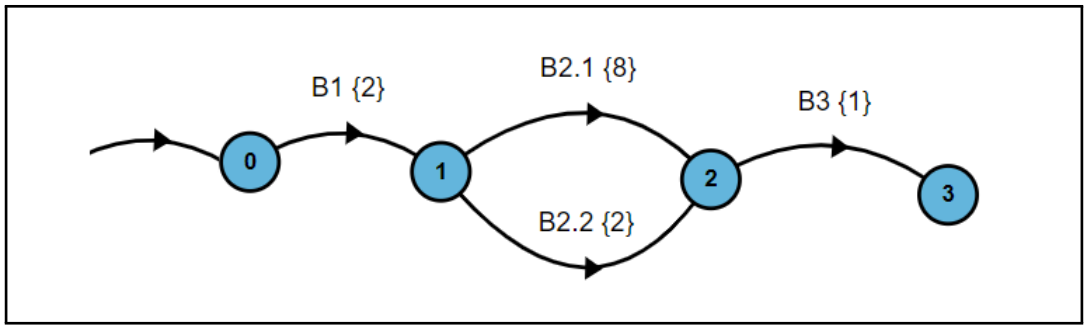
costs of the BET with the highest probability of being executed.

Figure 3.7: Running Example with Conditional Relation.



Source: Author.

Figure 3.8: ELTS model of conditional relation.



Source: Author.

As in a conditional relation only one of the BETs involved will be actually executed, the cost of an execution involving a conditional relation will take into account only one possible path in the model. In the case of the running example, we could choose to consider an execution of $B2.1$ (with cost 8) or $B2.2$ (with cost 2), depending on the type of analysis we were interested in.

3.3.3 Iteration Relation

An *iteration relation* is characterized by a set of BETs that can occur iteratively until a condition is met, as in a loop. For example, if a BET $B1$ is a part of an iteration relation, it may be repeatedly executed until the required condition occurs. This repetition of execution can be seen as $B1$ occurring after another occurrence of $B1$ itself, such as in a sequence relation. The main difference is that the BETs in this sequence are all the same. Moreover, the number of repetitions is determined by the associated condition, as in a conditional relation, but the condition is repeatedly tested.

The evaluation of the energy consumption in an iteration relation is given by equation 3.4. In the equation, $\delta(B)$ represents the energy of one execution of the BET B involved in the iteration relation. To obtain the total energy cost (i.e., the sum of the costs of each iteration), we only multiple the energy cost of one iteration by the number of iterations.

The value of $\delta(B)$ could be based on one single execution of B, an average cost of executing B, the highest/lowest cost of executing B, or even it could represent the total energy cost to execute all the necessary iterations. This choice is up to the developer and the analysis of the model should take this into account. As the number of times the BET will be executed depends on the evaluation of a condition, the user has to define a value for N according to their need or evaluate multiple possibilities.

$$\delta_{iter}(B) = \delta(B) * N \tag{3.4}$$

$$\delta_{iter}(B) = (\delta(B) + \delta(e)) * N \tag{3.5}$$

Similar to the costs of conditional tests, when we handle iteration tests the energy costs can be obtained with the equation 3.5, where we consider the costs to realize the conditional loop plus the costs of the internal BET.

An iteration relation is characterized by a condition that is tested to check whether the BET will continue to execute or not. To represent this condition test, we use an empty transition (denoted by symbol ε) to reach the state that enables the BET execution, which leads back to the state where the condition test is represented, thus simulating the execution of a new iteration. We neglect the energy cost of the condition test, focusing only on the execution of the BET, assigning a cost value of 0 to the empty transition.

Hence, when the condition is false, then the BET does not execute and the flow of the program continues to the next BET. This means that, if the condition is initially false, the BET will not be executed at all. However, in some cases, the BET may occur at least once before the condition test (such as in a *do-while* structure), as shown in the code in Figure 3.9b. In these cases, we model the initial execution of the BET as a transition occurring before the condition test and then leading to same idea as discussed before.

As an example of iteration relation, we present the code in Figure 3.9a, which can be represented by the model in Figure 3.9c, where BET $B1$ is executed with cost x while the condition continues to be true. Note that the ELTS model in Figure 3.9c contains a shaded transition with BET $B1$ (as represented by the code shown in Figure 3.9b) to represented the situation mentioned before, where the BET involved in the iteration relation occurs at least once.

Figure 3.9: Iteration relation examples

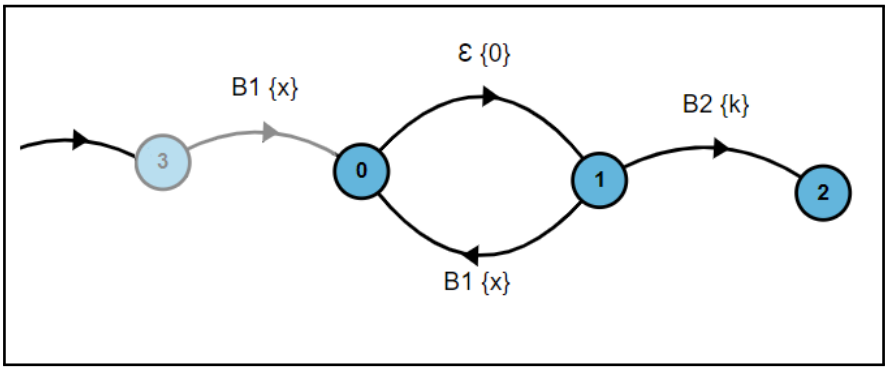
(a)

```
1 while (i < 5){
2       ...
3       i++;
4 }
```

(b)

```
1 do {
2       ...
3       i++;
4 } while (i < 4)
```

(c)

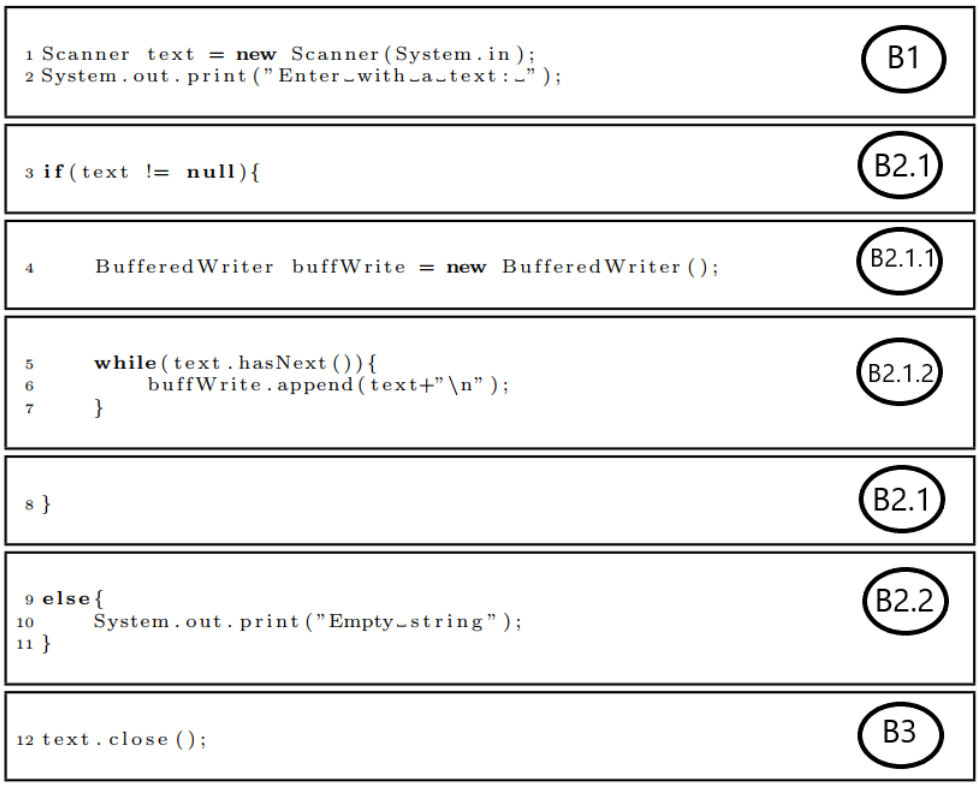


Source: Author.

An instance of an iteration relation using our running example is presented in Figure 3.10 between lines 5 and 7. We expand only BET of $B2.1$ from the ELTS presented

in Figure 3.8, where the internal relations are detailed in BETs $B2.1.1$ (sequence relation) and $B2.1.2$ (contains an iteration relation), as shown in Figure 3.11. Note that line 3 contains the condition associated to $B2$ and we do not consider its cost. However, if necessary, its cost could be obtained by calculating $\delta(B2.1) - (\delta(B2.1.1) + \delta(B2.1.2))$. This information can be necessary when the software developer is analyzing the energy consumption of their system and notice that the internal BETs do not contain an energy hotspot, indicating that, in this case, the tested condition may have a great influence on the total cost.

Figure 3.10: Running Example with Iteration Relation.

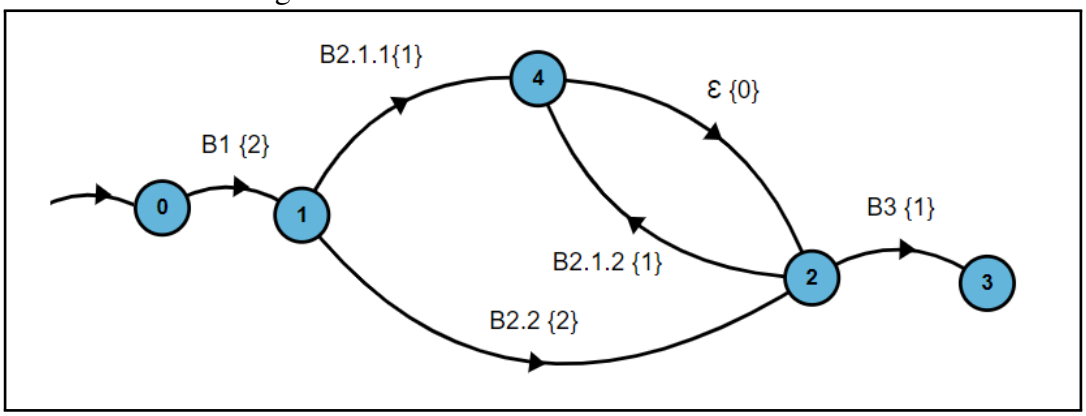


Source: Author.

3.4 Step 4: Model Analysis

This step consists of an analysis of the model built in the previous step to check whether there is anything that could be modified to improve energy efficiency. The developer can analyze whether energy requirements are satisfied by the system, which could not be done directly on the actual system. For instance, we may apply the energy-based property verification proposed in (DUARTE; ALVES; MAIA; SILVA, 2019) or discover energy bottlenecks of our application simply inspecting the model. In addition, the ELTS

Figure 3.11: ELTS model of iteration relation.



Source: Author.

may possibly allow a better visualization of the system energy behavior, providing documentation of this aspect, also supporting evolution of the system in terms of energy consumption.

Based on the analysis step, the software developer may decide to end the process ("Model Evaluation" in Figure 3.1). This may occur because they did not find any energy problem (e.g., no hotsposts), there was not any part of interest or all energy requirements were satisfied ("No" path from the decision point "Any further analysis required" in Figure 3.1, which leads to the "End" node).

On the other hand, the developer may want to analyze further some part of the energy behavior ("Yes" path from the decision point "Any further analysis required" in Figure 3.1). In this case, they should select what part of the model needs to have its level of abstraction changed. This change in the abstraction level is not mandatory, hence the developer has to define when the level should be changed. The criteria to change the level depend on the properties of interest to be analyzed. In this work, for the experiments, we considered points of high consumption as a criterion to modify the level of abstraction. When the software developer chooses to change the abstraction level of a part of the model, it is necessary to check whether it is possible to do it. This check depends on the type of the involved BET: if the BET is a composed BET, it is possible to change the abstraction level, looking into internal BETs; in case the BET is atomic, there is nothing else to be done using the model, leaving to the developer to try to change the system at the code level for a more energy-efficient alternative.

If the BET is a composed BET, the developer has to check whether the tool used to collect energy costs supports that level of granularity. If it does not, the process stops and the developer should try to use a different tool. However, if the tool supports the required

level of abstraction, then they can proceed with the changing of abstraction level. Hence, taking the "Yes" path from the decision point "Any further analysis required" in Figure 3.1 depends on being possible to expand the necessary BET and having a tool to collect the necessary energy costs.

In this process, a composite BET will be decomposed into its internal BETs, and their costs should represent their part in the total cost of the previous level represented ("Change of abstraction level" in Figure 3.1), which leads to the "End" node). Therefore, the change of abstraction level enables the expansion of parts of the model for better comprehension, identifying more accurately energy hotspots, and so, improving the analysis step. Note that only the part of interest is changed, keeping the rest of the model as it is. That part is refined (i.e., the BET is expanded) and the developer can analyse it using the previous steps (BET definition, energy cost collection, model construction and model analysis). This process occurs iteratively until either the developer determines that it is not necessary to apply any more modifications or they have arrived at a situation where it is not possible to proceed.

To represent the different abstraction levels in our approach, we have adopted a notation to model a composite BET that has been broken into smaller BETs: internal BETs are identified by the same numeric identification of their high-level BET followed by a "." and another number identifying that BET as part of a composite BET. For instance, an internal BET of a BET $B1$ would be $B1.1$. Following this idea, a second BET inside $B1$ would be identified as $B1.2$. An internal BET of $B1.1$ would be $B1.1.1$, an internal BET of $B1.1.1$ would be $B1.1.1.1$, and so on. Hence, the longer the identifier, the more levels of abstraction are involved. These notations were chosen to define a pattern of representation in the model and make it easier to identify different abstraction levels. However, the software developer can use their own notations according to their projects.

An example of the change of abstraction level is BET $B2$ represented in Figure 3.4, where we identified this BET as a conditional relation and subdivided it into BETs $B2.1$ and $B2.2$ as illustrated in Figure 3.7. BET $B2.2$ is an atomic BET, not being possible to alter its abstraction level. In this case, the developer should try to change the print command to improve energy efficiency. BET $B2.1$ contains other relations that may be expanded (in this case, a sequence and iteration relation, labeled $B2.1.1$ and $B2.1.2$ as represented in Figure 3.10), which means that it could be further analyzed.

A BET that occurs iteratively, such as $B2.1.2$, could be a probable energy bottleneck. In this case, we could look into its internal BETs or analyze whether the number of

iterations could be a problem. If there is a need to understand better the energy behavior of BET $B1$, we could split it into two other BETs, at which point we would arrive at a level where it is not possible to carry out any more changes of abstraction level. This analysis can be done due to the traceability established between the BETs and their respective parts of the code. In this case, the highest consumption identified in the model in Figure 3.8 was in transition/BET $B2.1$, which represents lines 3 to 8. When we expanded it, we found the energy interest point represented by BET $B2.1.2$, which is an iteration relation.

Therefore, we decrease the level of abstraction, detailing the internal BETs, to provide to the software developer a better visualization of possible energy bottlenecks and points of interest. These evaluations help the developer improve energy efficiency due to the traceability between model and code. It is possible to go back to the source code and identify if the BET is composed or atomic, and so, change the abstraction level and also define which part of the code has to be modified.

4 RESULTS

In this chapter, we present some experiments using the approach proposed in the previous chapter. In these experiments, we show how to construct a model to represent the energy behavior of the system, keeping the traceability between source code and model. Furthermore, we can use different levels of abstraction, based on the relations defined in Chapter 3.3, to better understand the energy behaviour of a program and identify which specific parts significantly affect energy consumption. We used the LoTuS tool (BARBOSA; LIMA; MAIA; COSTA, 2017) to manually create the models and collected energy information using the jRAPL library (LIU; PINTO; LIU, 2015).

We performed the evaluation of possible scenarios in practice, executing all steps that we have proposed to construct ELTS models and carried out the analysis of these models. These experiments were carried out using a PC with a processor Core i5 3 GHz, with 8 GB of RAM. The energy cost of each action (represented between curly brackets in the transition label) consists of an average value based on 10 executions and its value is presented in Watts (W).

The experiments are used to show how to construct an ELTS model from a source code, keeping the traceability between code and model. To demonstrate this, the selected experiments are used to identify the basic energy units of the programs and their relationships, also how to change the abstraction level to a better understanding regarding energy behavior. We have chosen programs that implement algorithmic tasks that demand a significant energy consumption to be analyzed. These programs are available online with open access, making it possible to reproduce our results. These are real applications, being used in a server environment, and, the program was developed in Java language due to the jRAPL library enabling to collect of energy costs in different granularities. For these experiments, we use the equations 3.2 and 3.5, where we ignore the conditional and iterations test.

4.1 FTP Program

The File Transfer Protocol (FTP) (POSTEL; REYNOLDS, 1985) involves two entities (client and server), where the user wants to transfer files to a remote host. To establish communication, the user provides their identification and password. With the user authorized, it is possible to transfer files between both sides (KUROSE; ROSS, 2007).

We used the Apache Commons Library[1] and we have adapted the code of the FTP implementation from a repository of Java codes and tutorials[2] that uses this library. The FTP program implements the transmission of a file using the FTP protocol. In our experiment, we upload a file of 15 MB from a client to a server, given the IP address, port number, username, and password, and evaluate the software energy consumption of this operation. The original code includes two approaches to submit a file, which are executed in sequence. However, to better analyze the differences between the two approaches, we have adapted the code where the user can choose which approach will be used to send the file.

The purpose of this experiment was to model the energy consumption from the client side using the defined modelling ideas to generate the ELTS model and compare the two approaches to identify which one has lower consumption. As the code was implemented in Java, we used the jRAPL library to annotate and collect the energy costs according to the chosen BETs.

4.1.1 Model Construction

The source code of the adapted FTP protocol program is presented in Figure 4.1. This program initially defines the IP address, port number, username, password, and instantiates an FTP client. Line 3 refers to this part of the code and we leave it out because it does not affect our analysis. After that, the program creates a connection using the provided information and the user chooses which approach will be adopted to submit the file. The software developer has two alternatives to submit the file: one using storeFile (the input file is sent as a whole using the command described in line 12) or through of OutputStream (the input file is sent byte by byte, as described in lines 23 to 25). If an error occurs, the *catch* block in lines 33 to 34 is triggered. Regardless of occurred before (complete execution of the *try* block or error) the *finally* block in lines 35 to 44 executes and ends the FTP connection.

In general, the goals when building a model are: 1) to get the smallest number of BETs possible to facilitate the analysis step and enable us to expand only parts of interest, if necessary; 2) to obtain an overview of the system behavior good enough to conduct an initial analysis, respecting the semantics of the code. Therefore, we should seek a balance

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

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

Table 4.1: BET table of FTP program.

BET	Lines	Energy Consumption
B1	3-4	0.0613
B2	5-32	1.4181
B3	33-34	0.0000
B4	36-43	0.0013

between 1) and 2), i.e, construct a model that covers the general behavior of the system with a minimum number of BETs. Then, we should expand only the parts of the model that we have to in order to do further analyses.

Considering this, we selected the main basic energy units and their relationships from the source code presented in Figure 4.1 and created a table to represent the mapping from code to BETs. This table delimits the set of statements of each BET, which should have their energy costs collected to build the corresponding ELTS model. Using the table 4.1, we mapped parts of interest and associated a label and an energy cost to them.

To construct the table, we analyzed the basic energy units contained in the source code. Lines 3 and 4 of the code are only used to set up some parameters and to instantiate the FTP Client. Therefore, we consider them as a single BET, named $B1$. Next, we identify that lines 5 to 32 represent a *try* block and lines 33 to 34 represent a *catch* block. Hence, the semantics (in this case, the Java semantics) represented by these blocks indicates a possibility of an execution of either the *try* block or the *catch* block). Therefore, we consider each block as a separate BET ($B2$ and $B3$, respectively) and define a conditional relation between them. Because the *finally* block executes regardless of which block occurs before, we define a BET $B4$ to represent lines 35 to 44 and consider it as having a sequence relation with the BETs $B2$ and $B3$.

In the context of the FTP program, we could model the whole program as a unique BET or consider a BET $B1$ between 3 and 4 and another BET involving lines 5 to 44. However, this approach does not provide a view of the system where it is possible to identify the energy consumption for each relevant part. For this reason, we chose to group the BETs for tasks/blocks that bring with them some meaning involved This, we select the task executed in *try* block, where will be representing by BET $B2$, next, the *cath* block case occur some error in *try* block, being representing by BET $B3$, and so, the *finally* block that disconnect the connection establish, used the BET $B4$.

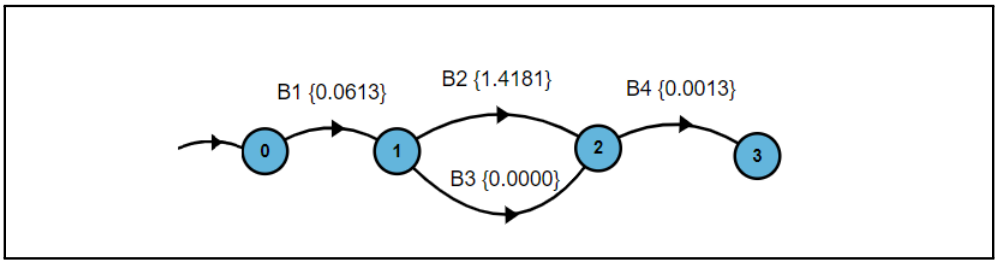
Starting from Table 4.1, created with each BET defined before, we built the ELTS model presented in Figure 4.2. In this model, we describe a view of the system in a high level of abstraction, thus having a more compact model, which facilitates an initial

Figure 4.1: FTP program source code.

```
1 public 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 }
```

Adapted from: https://www.codejava.net/java-se/ftp/java-ftp-file-upload-tutorial-and-example

Figure 4.2: Initial model of the FTP program.



Source: Author.

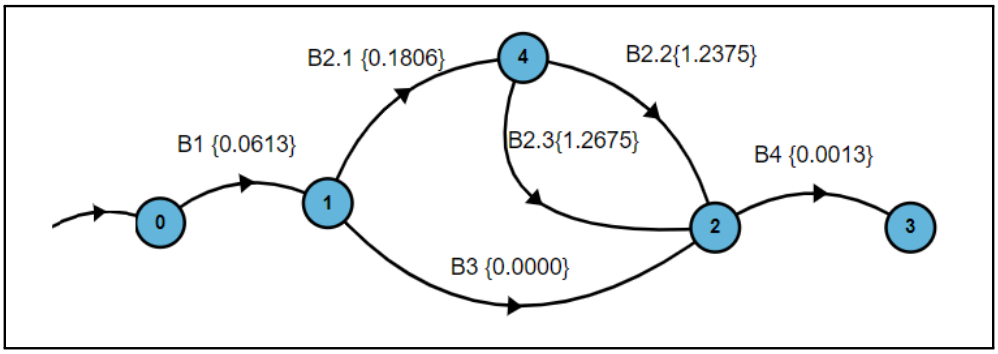
model analysis. With this model, it is possible to determine that the BET consuming most energy is $B2$. Using our approach, we must check whether is required/possible to change the abstraction level, for a better understanding regarding the energy behavior. We can see that $B2$ is a composite BET, therefore, it is possible to expand it and the energy costs of $B2$ can be broken into smaller units. To do that, we keep the other BETs at their current level of abstraction and expand only the unit of interest. Hence, we change the abstraction level, mapping the internals BETs of $B2$. These new BETs are shown in Table 4.2, where we identify the connection and login with the server represented by BET $B2.1$ (lines 6-9), and the BETs to represent the *if* and *else* blocks, defined as $B2.2$ (lines 10-17) and $B2.3$ (lines 18-32), respectively.

Table 4.2: Second BET table of FTP program.

BET	**Lines**	**Energy Consumption**
B1	3-4	0.0613
B2.1	6-9	0.1806
B2.2	10-17	1.2375
B2.3	18-32	1.2674
B3	33-34	0.0000
B4	36-40	0.0013

Using the table, we constructed the respective ELTS model, presented in Figure 4.3. To do that, we kept the other BETs at their current level of abstraction and expanded only the unit of interest. Considering that $B2.2$ and $B2.3$ corresponded the to *if* and *else* blocks, they have a conditional relation. As $B2.1$ occurs before this conditional choice, it has a sequence relation with both $B2.2$ and $B2.3$.

Figure 4.3: First expansion of the FTP model.



Source: Author.

Note that the model in Figure 4.3 shows that the two possibilities (represented by BETs $B2.2$ and $B2.3$) to submit a file have almost the same energy cost. Therefore, we could conclude that these two approaches do not differ in terms of energy consumption and end our analysis. However, other criteria could be analyzed, such as time and space

costs, probabilities of execution, etc., to determine the best option. The choice to continue or to stop depends on the software developer, knowing that the larger the model the harder to analyze it.

Although we could have stopped our analysis, we would like to understand the code better and identify what could be changed in the source code. We started changing the abstraction level of BET $B2.2$ to analyze the first approach for sending a file. We subdivided $B2.2$ into $B2.2.1$ (used for submitting the file using FTP protocol, lines 11-13) and $B2.2.2$ (shows a message if the process was successful, lines 14-16), as shown in Table 4.3.

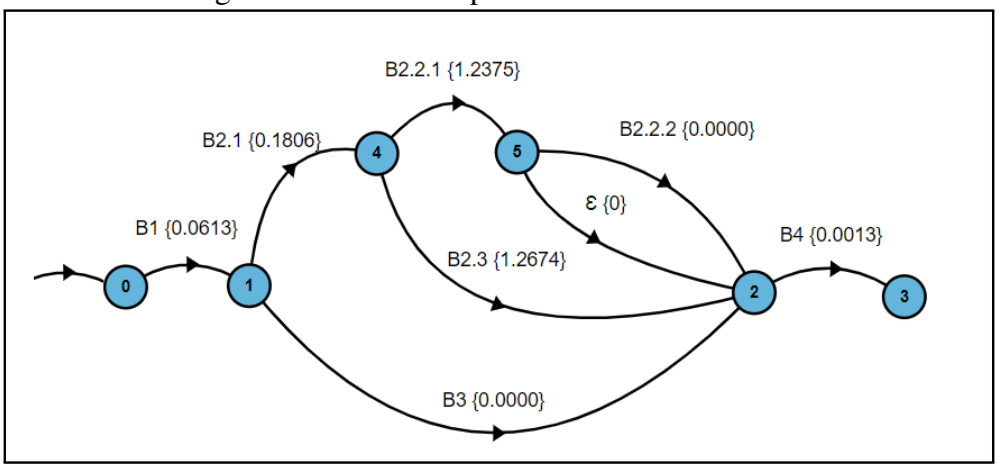
Table 4.3: Third BET table of FTP program.

BET	**Lines**	**Energy Consumption**
B1	3-4	0.0613
B2.1	6-9	0.1806
B2.2.1	11-13	1.2375
B2.2.2	14-16	0.0000
B2.3	18-32	1.2674
B3	33-34	0.0000
B4	36-40	0.0013

A new model represented by Figure 4.4 is constructed updating the abstraction level of BET $B2$ presented in previous level. The transition $B2.2.1$ is the BET that consumed more energy, where the representation in the source code is used to submit the file in line 12, and the other lines that composed this BET (line 11 and 13) are used only to print a message and to close the *inputStream* connection. After this, there is a test in line 14. The empty transition in the model indicates a failure in sending the file and the transition $B2.2.2$ represents the success to submit the file. The cost of $B2.2.2$ is 0 because the energy cost is so small that the measure tool considered it as zero. Hence, the main point of energy consumption is line 12. BET $B2.2.1$ is the call of method *storeFile* of library Apache Commons. This is one of our termination criteria, since it is an external method, being not possible to continue to expand the abstraction levels nor modifying something in our code.

Now, we investigate the other method to send a file, represented by BET $B2.3$. In this BET, we can see a group of configurations between lines 19 to 22, which is used to define the size of the buffer. We labeled this new BET as $B2.3.1$. With these configurations done, lines 23 to 25 contain the code to send the file byte by byte to the server, defining BET $B2.3.2$. We have the closure of the FTP connection, represented by BET $B2.3.3$, and in case of success in submitting the file, a message is printed for the user

Figure 4.4: Second expansion of the FTP model.



Source: Author.

between lines 29 and 31 (BET $B2.3.4$). All these mappings are shown in table 4.4.

Table 4.4: Fourth BET table of FTP program.

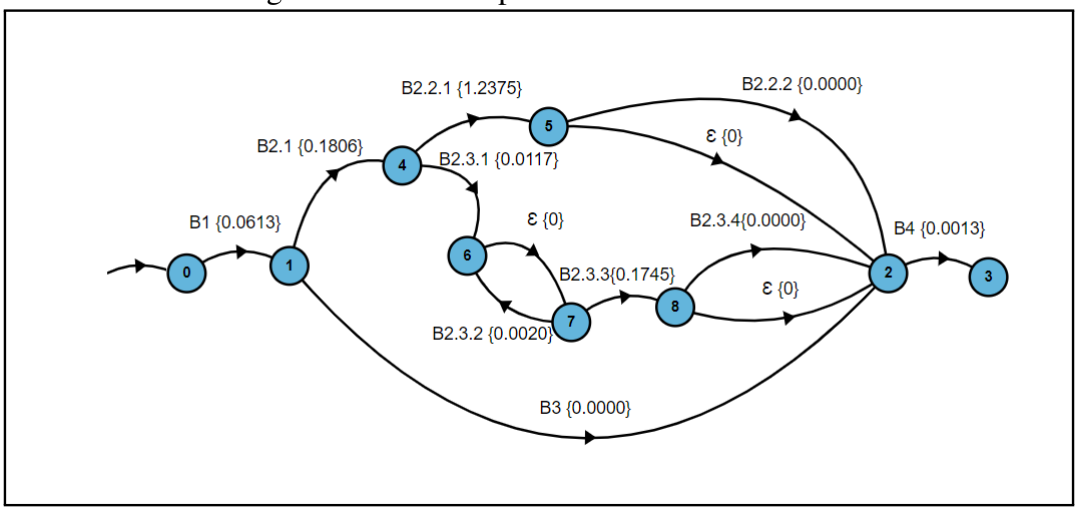
BET	**Lines**	**Energy Consumption**
B1	3-4	0.0613
B2.1	6-9	0.1806
B2.2.1	11-13	1.2375
B2.2.2	14-16	0.0000
B2.3.1	19-22	0.0117
B2.3.2	23-25	0.0020
B2.3.3	26-28	0.1745
B2.3.4	29-31	0.0000
B3	33-34	0.0000
B4	36-40	0.0013

The ELTS of this other abstraction level is presented in Figure 4.5. This model contains the expansion of BET $B2.3$, where we can see that BET $B2.3.1$ has a sequence relation with BET $B2.3.2$, which represents an iteration relation, i.e., the energy costs will occur iteratively and the costs can vary for each iteration, being the total consumed in one iteration is equal to 0.0020 W. Finally, we have a sequence relation with BETs $B2.3.3$ and $B2.3.4$, where the latter contains a condition relation.

At this point, we achieve an ELTS that does not allow us to change the abstraction level any more. Following our approach proposed in chapter 3, we could run some analysis on the model, searching for possible energy problems, the path of execution with highest cost, etc. to define which of the two methods to submitting a file using FTP protocol would have the lower cost.

The points that represent the possible energy bottlenecks in our program would be BETs $B2.2.1$ and $B2.3.2$, which are the ones with the highest costs. Using the traceability,

Figure 4.5: Third expansion of the FTP model.



Source: Author.

we can identify that these BETs correspond to lines 11 to 13 and 23 to 25, respectively. These BETs in the source code are exactly the two different methods to submit the file using the FTP protocol. Here, the software developer could carry out a deeper analysis of these BETs.

Using the model, we could study the two paths of execution with the highest energy costs: 1) $B1 - B2.1 - B2.2.1 - B2.2.2/\varepsilon - B4$, 2) $B1 - B2.1 - B2.3.1 - \varepsilon - (B2.3.2 - \varepsilon)^* - B2.3.3 - B2.3.4/\varepsilon - B4$. We identify that the first path has a cost of $1.4807W$, representing the first alternative to submit a file through FTP. Evaluating the cost of a single iteration in path 2, with a cost of $0.0020W$, we would have a considerably lower cost value for this path. However, to submit the whole file is necessary to execute multiple iterations. In this case, the total cost to send the file of our experiment is $1.0812W$. Thus, the total energy consumed in this path would be $1.5106W$, being the second possibility of sending a file using FTP.

Considering the two possibilities, we could indicate that difference is too small to determine which method is better, as the two methods consume practically the same amount of to be executed. Nevertheless, we could compare further the two methods of sending a file to determine which one is more energy-efficient. We could, for instance, identify the possibilities of reducing the costs of some methods. Thus, a software developer could determine where to change to reduce the total consumption, if possible.

Based on the model of the third expansion of the FTP model (Figure 4.5), we could compare different scenarios of consumption, depending on the size of the file, because the semantics of the code does not change; i.e., the model remains the same, changing only the energy costs according to the file size. Therefore, the larger the file, the more energy

consumption will be needed to send it. This change could affect the energy efficiency of the methods.

4.2 File Compression

We propose a modification of the FTP program, where before submitting a file, the file is compressed. The idea of this evolution is to analyze the impact on the energy costs of sending a compressed file instead of the original file. For this experiment, we used the same file of 15MB from the previous example, which is compressed before it is sent. The compression method produces a file with 9 MB. To do that, we use the library Zip from the package java.util.zip, which enables us to compress a file into a .zip. The original FTP code does not change: we first compress the file and then send the compressed file using the code presented in Figure 4.6. Therefore, previously, we had only lines 3, 7 and 8. Now, we have included lines 4 and 5 to perform the compressor code, before sending the file using the FTP protocol.

Figure 4.6: Main class of program.

```
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 }
```

Source: Author.

In this context, note that we do not change the behavior of the FTP code. However, we add a new component to our system, where this modification can affect the energy consumption. Hence, to evaluate this situation, it is necessary to execute the FTP code again. Although, the behavior keeps that same, i.e, the model will continue with the same BETs of the model in Figure 4.5, due to the file being compacted, the costs in of executing the FTP code can be affected.

Through this modification, we propose to identify, for example, which of the op-

tions to submit a file using the FTP program has a gain in their energy consumption with a compressed file. This can help decide, with this change, which method is a better option to use. We could also analyze the size of the impact in the total energy consumption of the system to check if the inclusion of the new component is worth it according to the gain in the energy consumption.

The source code of the file compression program is presented in Figure 4.7, where we create the output file in line 4. Next, the library ZipOutputStream uses the output stream to create a zip output file and, in line 6, input stream is used to obtain the data from the input file. Lines 9 to 11 write data to the new zip file.

Firstly, we built the ELTS model representing the source code of the compression algorithm (represented by Figure 4.7), identifying the main BETs, i.e, following the concept where we can model the general behavior of the system with the minimum number of BETs possible. Next, we check if is necessary or possible to change the abstraction level of part of the model to highlight a probable energy bottleneck. Lastly, we created a composed model with the ELTS model that represents the FTP code. We analyzed the total energy consumption of the path traveled to compress and submit the file, i.e, understand how much energy consumption consumed more or less through this modification.

Figure 4.7: Code of the file compression program.

```
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}
```

Source: Author.

The BET table that represents our first modeling is presented in Table 4.5. Lines 2 to 16 of the code are used to compress the file and lines 17 to 19 treat the event of an error.

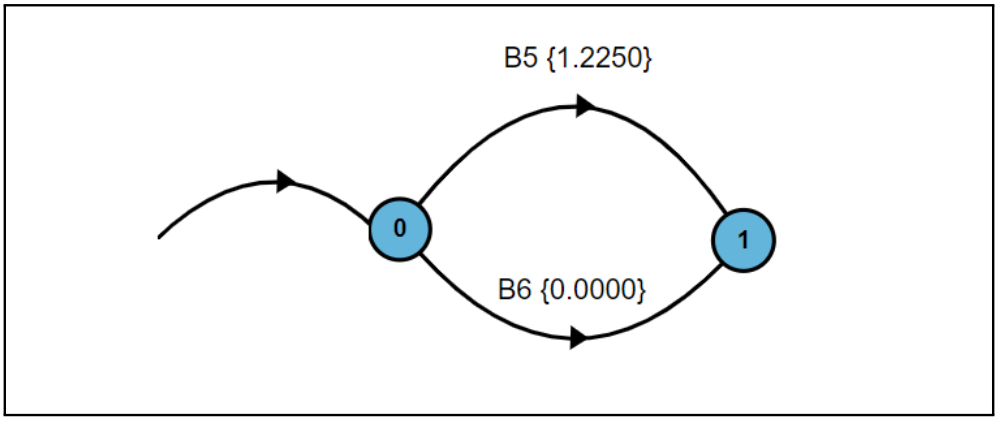
The ELTS that represents the initial modeling is in Figure 4.8. For a better understanding, we initiate the labels from $B5$, following the sequence of labels started in the previous experiment, so that there is no confusion when putting the models together. Analyzing the ELTS, we can see that there exist two paths, $B5$ and $B6$, where only $B5$ has a cost different from zero.

Using the traceability between model and source code, it is possible to change the abstraction level of BET $B5$. As this initial model can be very abstract and does not demonstrate the real software energy bottleneck, we expanded the model to identify if there is an energy bottleneck.

Table 4.5: BET table of compressor

BET	Lines	Energy Consumption
B5	2-16	1.2250
B6	17-19	0.0000

Figure 4.8: Initial model of the compress code.



Source: Author.

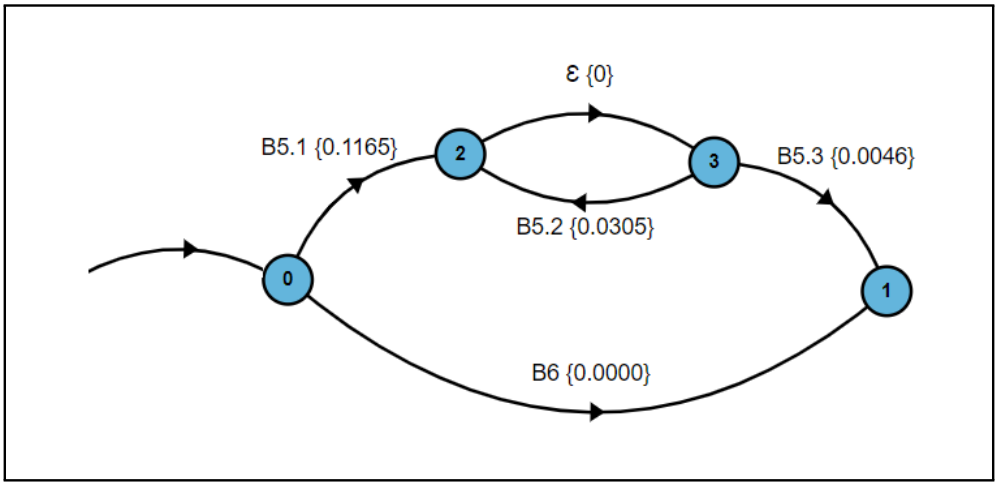
Table 4.6 and Figure 4.9 shows the first expansion of compress code, where we expand the BET $B5$ from previous level, transforming into BETs $B5.1$, $B5.2$ and $B5.3$.

In this experiment, we reach the model 4.9, where in this point, the BET $B5.2$ is the hotspot of energy, being the respective lines represent the part of code that carries out the compression and does not possible change the abstraction level. For better analysis, we could evaluate the impact in the energy with different sizes and types of files, and so, to indicate more precisely according to each file the energy impacted.

Table 4.6: BET table of compressor code in the first expansion in compress code.

BET	Lines	Energy Consumption
B5.1	3-8	0.1165
B5.2	9-11	1.1039
B5.3	12-16	0.0046
B6	17-19	0.0000

Figure 4.9: ELTS compress with First Expansion in compress code.



Source: Author.

4.3 File Compression with FTP

In this experiment, we combine the models created in the two previous experiments. We use this combination to evaluate the composition of models in terms of energy consumption.

We compose the model in Figure 4.9 with the model in Figure 4.5 and obtain the behavior model presented in Figure 4.10. In this model, we can visualize our evolution and compare the energy values from the file with 15MB and the compressed file with 9MB. The model was constructed following the abstraction level presented in Figure 4.5 because this level allows for a more detailed view of the FTP program behavior. Note that, according to the framework, firstly, a model must be constructed and then energy costs are collected and annotated on the model. Therefore, it is recommended to compose the models first, and after that, associate the energy costs for each BET of the composed model.

Intuitively, adding the new feature to compress the file before submitting the file should increase the costs. Indeed, we noted that the costs of the whole system have increased. However, because the file size was smaller after the compression, the costs to send it using the FTP program were lower than before, which led to a lower total cost than

expected, considering the previous costs for each individual component. Analyzing the main BETs ($B2.2.1$ and $B2.3.2$, which are the points corresponding to the two possibilities of sending a file), we can note that in the previous level the costs were $1.2375W$ for BET $B2.2.1$, and, with total costs to execute the loop of $1.0812W$ for BET $B2.3.2$. Now, with this new component added, we have the following costs: $1.0633W$ for $B2.3.2$ and a total of $0.9515W$ for BET of the loop. The reduction in the first method was 14%, while the second method to send the file had a reduction of 12%.

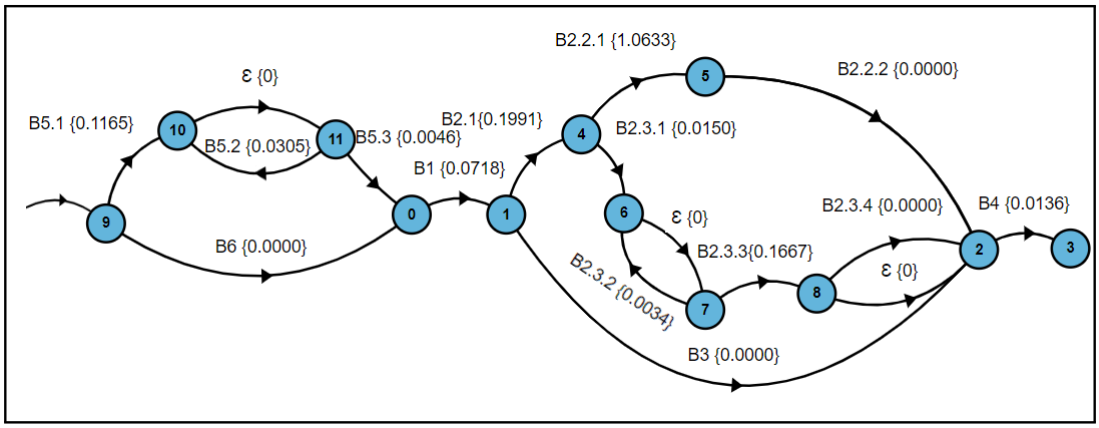
As well as the FTP program, in this experiment of file compression with FTP arrived at a point where it is not possible to expand the model any more. Therefore, we can perform on this new model the same analysis we did on the previous model.

With the addition of the compression program, we have a new energy bottleneck, represented by BET $B5.2$, which is the BET that executes the file compression. In addition, we still continue with BETs $B2.2.1$ and $B2.3.2$ to send the file. However, these BETs now have a smaller file to send.

The model represented in Figure 4.10 continues with 3 paths of executions from state 0, however, with a compression program, before can occur the path $B5.1 - B5.2 - \varepsilon - B5.3$ or $B6$. In this case, we do not consider the execution of BET $B6$ and the path $B1 - B3 - B4$, where interests us to analyze the energy consumed for the success of the operation of compress and sent the file.

Hence, we have two execution paths considering a success situation: $B5.1 - B5.2 - \varepsilon - B5.3 - B1 - B2.1 - B2.2.1 - B2.2.2/\varepsilon - B4$ and $B5.1 - B5.2 - \varepsilon - B5.3 - B1 - B2.1 - B2.3.1 - B2.3.1 - \varepsilon - B2.3.3 - B2.3.4/\varepsilon - B4$. The total energy consumed by the first path of execution is $2.5728W$, while the second path consumed $2.6427W$.

Figure 4.10: ELTS of compress code composite with FTP code.



Source: Author.

The BETs presented in the model of Figure 4.5 remain the same; what changes is the costs associated with them, where the costs will vary according to the scenario represented. As the focus of this work is to show how to model the energy behavior, we do not provide more detailed analyses, but only raise some points to be observed.

Analyzing only the submodel that depicts the FTP, from state 0 the energy costs would be $1.3478W$ taking the first execution path, which has a lower than previously. However, the execution of compress costs $1.2250W$, which added with the FTP cost results in a total of $2.5728W$. Therefore, the compression code brings an impact of $1.2250W$ to the program energy consumption; i.e, the cost to compress the file is higher than that of simply sending the original file. 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.

4.4 Discussion

In this section, we will discuss some aspects regarding the application of our approach in the experiments, discussing advantages and limitations found.

We used the proposed mapping of energy behavior from source code to model, where it was possible to begin at a higher-level abstraction and decrease this level to identify more precisely energy hotspots. In addition, we demonstrated the usability of the model conducting some analyses to indicate possible energy bottlenecks.

To guarantee the traceability between model and code, we used the concept of BETs to associate lines of code to one transition in the model. Therefore, possible modifications will be done at the correct point in the code, due to this mapping.

In the FTP experiment, we were able to construct the ELTS model, expanding this model to better understand the energy behavior until we reached a point where it was not possible or necessary to change abstraction levels. And so, with the model represented in one level that allows some analyses we identified the main energy hotspots (BETS $B2.2.1$ and $B2.3.2$, that correspond to lines 11 to 13 and 23 to 25, respectively) and represent the two methods to send a file using the FTP protocol. We determined the total cost of each path and identified that the difference between them is very small.

A new experiment was done with the same file used by the previous experiment, where this file of 15MB was compressed. We built the model following our approach, where the model showed the energy point of interest, which was the BET that does the

compression byte by byte.

Through these experiments, we composed the resulting models, where the compression code was executed before the FTP code. We evaluated the impact of this evolution. There was a reduction in the first method of the FTP of 14%, while the second method to send the file got a reduction of 12%. However, the costs of the whole system increased, where we conclude that it would be necessary more experiments with different file sizes and types to check if this combination is energy efficient for a certain size or type of file.

In the case of the compression code, represented by the model in Figure 4.10, BET $B6$ represents that the file is not compressed. Hence, the file sent via FTP will be of 15MB, i.e, the costs would be those presented in Figure 4.5. Therefore, we do not change the original control and data flow of the program, but, depending on what occurred previously, the costs of the next BETs represented in the model can be affected. Still, with this evolution, the difference of energy costs between the two methods to send the file continues to be very small.

Analyzing specifically for this evolution proposed, we noted that the cost to compress a file is higher than not to compress. To be more precise, the software developer could use the same model presented in Figure 4.10 and change only the file size. With this, they could better identify from which size it would be more rewarding to compress before sending a file using FTP.

As we add some component to our system, the expected result is that the total costs will increase. In the model represented by Figure 4.10 we can note that costs of the system as a whole follow this pattern. However, if we analyze the part of the FTP model in isolation, the costs are lower, because the input file is smaller than the original file. Hence, specifically, the structure of the FTP model in Figure 4.10 does not change, but due to adding the compress model the costs of FTP were affected. Based on this, we identify the difference between operational behavior and energy behavior, where in this experiment, the operational behavior (behaviour of the code in terms of its operations) represented in the model was not affected, but the energy behavior changed.

Some limitations of our approach were identified during the experiments. For example, we ignore the costs of testing a condition in a conditional or iteration structure, considering only the costs of the internal code, as demonstrated in Figure 4.5 for BETs $B2.3.4$ and $B2.2.2$. The costs are ignored because we assume the analysis is simple and, thus, the costs are minimum. However, depending on the condition tested, this cost should

not be neglected.

For each level of abstraction, the developer needs to collect the necessary energy costs, which can be too much work and even not possible, due to tool limitations. Hence, the change of level of abstraction is also limited by the possibility of obtaining the costs at the desired level.

The case as the try-catch block may not be so well modeled, as shown in Figure 4.8 and Figure 4.2. The problem occurs when the *try* block is not completely executed. In this case, the code executes until a point and an exception occurs, which means that the code executed up to that point consumed energy, but this cost is not present in the model.

The label $B6$ of Figure 4.10 should be connected to the original model, where the FTP deals with a file which is not compressed. In this case, there should be two paths of possibilities: one where the file is compressed before sending, represented by $B5$ (state 11 connected to state 0) and another one where the file is sent without compression, represented by $B6$ (state 9 would be connected to the initial state of the original FTP model). However, we do not represent this second situation due to the model becoming very large and the possibility of the file not being compressed has already been analyzed when we discussed the FTP program alone.

Based on all the results presented in this chapter, we can highlight the following advantages of our approach: 1) Direct traceability between model and source code, which indicates to a software developer more accurately the BET that represents an energy bottleneck or point of interest to be analyzed; 2) Easy local change of abstraction level, which enables us to expand the model according to the necessary, i.e, the model initially is more compact and we can expand it to understand parts of interest; 3) A simple model compositionality, which makes it easy to compose models. Hence, the developer can explore different combinations of components and evaluate the impact of theses combinations. For example, it could be used to analyze how a evolution of a component could affect the system energy consumption.

This work proposed to provide a simple way to change the abstraction level of parts of the model. Hence, the software developer must pay attention to which abstraction level it is working on because the parts of the model contain different levels represented and care must be taken to not conduct incorrect analyses.

In this work, we sought to answer to the research question presented in Chapter 1: “How to allow even developers with little or no previous experience with models to build a model from an existing source code which is as close as possible to the software’s

energy behavior?". Due to the focus on software developers that might have little or no knowledge about behavior models, we presented a simple way to construct these models from an existing code using the concept of BETs (e.g., as presented in the running example and in the experiments).

We believe that our approach is simple for all types of developers considering the following facts:

1. BETs are based on well-known ideas for developers (blocks of code, semantics of programming languages and composition of commands), which makes the construction process simple;
2. The ELTS has a graph structure and semantics based on state machines, which are two familiar ideas for developers;
3. The change of abstraction level is local, such that each abstraction can be seen as a single program, which means that the same model construction ideas apply to each local modification;
4. The form of how the costs are inserted into the ELTS is simple, keeping the semantics of LTS, enabling analyses of the model;
5. Traceability between model and source code is direct and it is easy to determine which part of the model refers to the code, enabling the developer to identify points of interest and resolve possible problems directly in the source code.

For this reason, the answer to the research question is that our approach can allow the software developer with no experience to construct the behavior model with energy costs with an adequate representation, which can lead to analyses with more precise and reliable results.

5 RELATED WORK

In this chapter, we present some of the related work. We focus essentially on the aspects related to energy model representation and model construction.

Regarding energy model representation, the work presented in (DUBSLAFF; KLüPPELHOLZ; BAIER, 2014) describes a compositional modelling framework for dynamic Software Product Lines (SPL) (CLEMENTS; NORTHROP, 2001). The authors used Markov Decision Processes (MDP) (SEGALA, 2007) to model and analyze SPL behaviors, where the transition of the model is labeled with a decision (similar to an action for an LTS) and a cost value, which can represent an energy cost. The authors present the possibility of using a parallel operator to combine the operational behavior with all features of modules constructed. The framework can deal with probabilistic and non-deterministic choices and the produced model can be translated to the PRISM input language (KWIATKOWSKA; NORMAN; PARKER, 2002), where analyses can occur.

The work proposed in (BAIER; DUBSLAFF; KLEIN; KLüPPELHOLZ; WUNDERLICH, 2014) used Markov chains to model energy behavior, where the PRISM tool was used to conduct quantitative analyses based on probabilistic information. In PRISM, the energy costs were modeled as costs/rewards associated to model elements. They use Probabilistic Model Checking (PMC) (VARDI, 1985) in the analysis of low-level properties, where PMC is used to evaluate adaptive systems and their energy usage. The authors created the concept of energy-utility ratio, which consists in identifying interference of energy on restrictions of use. The value is represented as weight functions in the model.

The research in (ALSSAIARI; GINING; THOMAS, 2017) demonstrated how to use model policies in scenario of servers, which use a Markovian process algebra. In PEPA (Performance Evaluation Process Algebra) (HILLSTON, 1996) the modeled system is characterized by activities and components. They establish policies to control the power mode of parallel servers. Heuristics are applied using the policies with the goal of power-saving and performance. Therefore, the focus of this work is on policies established using PEPA, where the authors do not go into details about how the used model is constructed. The authors assume that the energy cost when a server is on will be the same for all servers, since their goal is to evaluate policies to manage energy costs with dynamic servers. Because the energy costs are fixed, they suggest the creation of a more accurate model to represent the behaviour of the system, as their models may be too abstract.

The work proposed in (DAMASO; ROSA; MACIEL, 2014) uses Colored Petri

Nets (CPN)(JENSEN; KRISTENSEN, 2009) to evaluate energy consumption in wireless sensor networks. The focus of this research is specifically to analyze the sensor's lifetime. To do that, they use a process represent an element and simulate its energy consumption. To collect the energy consumption, they execute the whole code and get the entire cost of this execution. This is converted into a transition label PwrConsumption in the model. A tool called IDEA4WSN is used as an instrument for energy-awareness. To represent the energy consumption, they use a node model (represented in the application layer) with costs consumed by the entire application and connected with the network layer. The energy consumed by nodes in application layer allows to evaluate the sensor lifetime.

In all studies cited above, they do not detail how to collect and construct the model with the respective energy consumption, where their main focus is to use the model with the resources only to analyze the energy impact. Hence, they only use the models as means to accomplish specific analyses in their contexts. Therefore, there is no guidance for a software developer to follow, as well as in (DUARTE; ALVES; MAIA; SILVA, 2019). Furthermore, none of them constructs the model from the source code, which may make it difficult to maintain traceability. They also do not specify how to change the abstraction level for enhancing the model representation.

About model construction, in (JUNIOR et al., 2006) it is proposed the use of Colored Petri Nets (CPN) to analyze the software energy consumption in embedded systems. The model is generated according to the occurrence of events and their probabilities. These probabilities are chosen based on assumptions, where these assumptions are provided by the modeler. They mention the possibility of different abstraction levels in different micro-controllers as a contribution. However, they do not show evidence of how to change the abstraction levels and how to deal with this. They collect energy information using a test-board coupled in a PC computer, where it is possible to check the code present in the micro-controller and, through a digital oscilloscope, connect with the test-board to collect the consumption, which is represented as oscilloscope waveforms. The code to be analyzed is implemented in Assembly, while to extract the annotations a C code is used. Finally, a CPN model is obtained, which allows simulations to discover some information of interest. The model can be visualized in the EZPetri (ALVES; ARCOVERDE; LIMA; MACIEL, 2004) tool, where they extended the tool to include the energy information, denominated EZPetri-PCAF (Power Cost Analysis Framework), allowing a set of analyses on the model. However, they do not make it clear how to establish the traceability between model and code and indicate a real hotspot of energy. In addition, as the process is

made automatically, the software developer does not have the possibility to choose which parts of the model should be analyzed.

The research in (DUARTE; MAIA; SILVA, 2018) proposes to extract an LTS model with probabilistic information based on execution traces produced by Java code. The tool uses the idea of context (DUARTE; KRAMER; UCHITEL, 2017), which is a combination of control-flow and values of the program to represent abstract states. The traces are used to calculate the probabilities and construct the behavior model. The model can be analyzed in PRISM to check properties about the represented behavior. As in our work, they present an approach to construct a model from a Java source code. However, the quantitative information used is probabilities, whereas we model energy costs, which require a different approach. For instance, they do not present any information about abstraction level in their approach, which means it is not possible to apply it to energy information as ours can.

As discussed in chapter 2, there are several other models that could be used to model software energy consumption, as well as all research mentioned above. However, none of these studies provides a step-by-step process to help model a system. We defined the ELTS to use a model simple to understand and analyze. In addition, the approaches that perform automatic model construction consider only probabilistic information. Our work, like that presented in (JUNIOR et al., 2006), deals with energy information. Nevertheless, the authors use consumption from the code as a whole, preventing local analyses, which is a valuable feature to detect specific spots of high energy consumption.

In Alves, Ferreira, Duarte, Silva e Maia (2020) they use LTS models to carry out experiments using Java code implementing the algorithms Bubble Sort, Insertion Sort, and Selection Sort. The purpose of this work is to use the jRAPL tool to collect energy information and to use model analysis to discover interest properties. In their analysis, they evaluate the total energy consumed by each data structure applied for the sorting algorithms and check which are the better options. However, the parameters to construct the models are not specified. In addition, they do not deal with multi-component systems and do not mention change of abstraction levels.

Alves, Ferreira, Duarte e Maia (2020) present an approach to support the software developer in the analysis of software energy consumption and the probability of occurrence of executions. They show how to use this information to discover possible aspects of the system, enabling to check a trade-off between the probability of an execution and its respective cost. The energy used is collected using the jRAPL tool and inserted into

an LTS. However, they assume that the model to be analysed exists and it has been constructed somehow. In this context, our models could be used as input to their approach if probability information is added.

The approach proposed by Eder e Gallagher (2017) consists in building a transition system through basic program constructs and associating with them the respective energy costs. These basic program constructs are instructions or statements, such as blocks of code. However, the authors do not specify clearly how to identify these basic software constructs, which means that, when compared to BETs, their abstraction does not have a clear definition and it is less tailored for describing energy units. Hence, traceability between the source code and model can be difficult, not being mentioned by the authors. The energy modeling is made using the instruction set architecture (ISA), which is an interface of hardware and software. This interface enables the developer to collect the energy consumed by operations in terms of hardware associated with basic program constructs. They apply static analysis to energy consumption, where the model represents the program semantics. The energy costs are collected according to the programming language and the hardware platform and associated with a basic software construct. They do not present any result or information about model composition. About abstraction level, the ISA interface is used to assign energy consumption at the hardware level to software operations (instructions, statements, procedures, and functions). However, they do not enter in details about how to change the abstraction level, which is important for pinpointing energy hotspots.

Compared to related work, our approach provides a model defined to work with energy information, allows different abstraction levels, maintains traceability between code and model, and describes a step-by-step way of building the model. None of the other approaches combine these features. Nonetheless, our approach does not deal with probabilities, as other studies do. With this information added to the model, a new type of analysis would be enabled to evaluate the aspect of probabilities, which would allow identifying whether it is worthy to perform some change in the code to reach energy efficiency, such as presented by Alves, Ferreira, Duarte e Maia (2020).

6 CONCLUSION AND FUTURE WORK

This work proposed an approach to support the construction of energy behavior models from code. To do that, we proposed a new model called ELTS (Energy Labelled Transitions System), which is an LTS (Labelled Transition System) augmented with an energy information label on its transitions. To build this model from Java code, we created the concept of basic energy unit (BET), which associates a part of the source code to a transition of the ELTS. This provides the necessary traceability for the identification of possible hotspots of energy in the code after an analysis on the model. This allows a developer to change parts of the code that have relevant consumption. We represent the semantics of the code in the model through relations between BETs, defined as sequence, conditional, and iteration. These relations enable a better understanding of the behavior when analyzing the model and represent the connections of its composing BETs. In addition, we propose how to change the abstraction level of parts of the model respecting the code and the programming language semantics. This change is applied by expanding composed BETS; i.e., BETs representing more than one line of code. A BET is only expanded if necessary, keeping the rest of the model as it is. Hence, the change applies only locally, to focus on the part of interest, thus keeping the model as abstract as possible. Thus, based on our approach, a new step has been taken towards building accurate energy behavior models for energy consumption analysis.

We demonstrate how to build a model with energy costs step-by-step based on the definition of BETs and their relations, consequently, generating an accurate model, i.e, a model that describes as closest as possible the real energy behavior of the system. With this, we search to guarantee that unnecessary details are not included, improving the precision of possible analyses. In addition, the mapping of the source code to the model is intended to respect and represent the semantics of the code and the programming language. The definition of BETs guarantees traceability, making it possible to easily identify which part of the software needs to be modified to improve energy efficiency. Through our approach, it is possible to change the level of abstraction of parts of interest, i.e, we can expand the model only when and where it is necessary. Furthermore, the inverse process can be done, which means the possibility of going back to a more abstract level in case a determined part of the code is unnecessary for a subsequent analysis.

This work presents some limitations and points that could be improved, such as:

- **Automated Model Construction.** We intend to develop a tool to automatically

build models following our approach, thus, avoiding that errors be inserted during manual construction. This could be done, for example, using annotations by the software developers in their code (similar to the idea proposed in (DUARTE; KRAMER; UCHITEL, 2017)), where these annotations would define BETs, which would then be used to construct the ELTS. However, the choice of BETs still continues to be a responsibility of the software developers, who knows their system and main points to be considered as a BET and what should be modeled for the analysis step.

- **Our modelling does not include recursion representation.** We do not consider examples that use recursion, and so, it will be interesting to how to model recursion. Initially, the purpose of BETs was to consider the main semantics used by programming languages, and so, we model the most common compositions of code components (sequence, conditional, and iteration) in a way that facilitates the model analysis step. As an initial idea, recursion could be modeled as an iteration, due to the similarity with this relation. However, the energy can be affected differently and a different representation may be necessary to facilitate understanding.
- **Loss of the energy cost when does not execute the *try* block.** As mentioned for the experiments, we consider the *try* block as a conditional statement, however, in case the try block is running and occurs an exception, the costs consumed until this point will be lost, therefore, will be interesting to find a way to represent this possible loss of energy, i.e, the code executes the costs to execute until this point and the costs consumed by the *catch* block.
- **Our model composition approach needs more investigation.** Model composition was presented in a simple way, and we do not indicate a general procedure for representing the real impact when models are composed.
- **We have not evaluated the approach for other types of systems** We do not explore some aspects that could contribute to energy analysis, such as parallel and distributed systems, and to show how to occur communication and the energy consumption between different components.

As future work, we intend to improve some aspects. For example, the addition of other quantitative information, to check whether it is possible to follow the same approach for other quantitative values, such as time and memory consumed. Also, we intend to find a way to describe the number of iterations of a loop as part of the model, so that the user

can represent this information.

Inclusion of probabilities in the model (as presented in the related work in (ALVES; FERREIRA; DUARTE; MAIA, 2020), (BAIER; DUBSLAFF; KLEIN; KLüPPELHOLZ; WUNDERLICH, 2014) and (JUNIOR et al., 2006)), given the system paths, if there is a low probability that a certain path will occur, a developer could evaluate whether it is worth performing changes to enhance energy efficiency. Therefore, would be interesting to indicate how to include probabilistic information in the ELTS, and so, how to interpret this combination (energy and probabilistic information), i.e, a combination ELTS and PLTS (Probabilistic Labelled Transition System) (JOU; SMOLKA, 1990).

The results of this work could be combined with other studies, such as Alves, Ferreira, Duarte, Silva e Maia (2020) and (ALVES; FERREIRA; DUARTE; MAIA, 2020), where the model constructed using our approach will be used as input to accomplish the different types of analysis. Some of these analyses could be the possibility of a path execution, choice of better options in terms of energy between algorithms, apply heuristics that would not be trivial to perform.

In addition, it would be interesting to allow developers to use our approach and give some feedback. This way, we could identify possible difficulties not detect during this work.

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>.

ALSSAIARI, A.; GINING, R. A. J.; THOMAS, N. Modelling energy efficient server management policies in PEPA. In: **ICPE 2017 - Companion of the 2017 ACM/SPEC International Conference on Performance Engineering**. L'Aquila, Italy: Association for Computing Machinery, Inc, 2017. p. 43–48. ISBN 9781450348997.

ALVES, D. et al. Experiments on model-based software energy consumption analysis involving sorting algorithms. **Revista de Informática Teórica e Aplicada**, v. 27, n. 3, p. 72–83, 2020. ISSN 21752745. Disponível em: <https://seer.ufrgs.br/rita/article/view/Vol27_nr3_72>.

ALVES, D. et al. Probabilistic model-based analysis to improve software energy efficiency. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. Natal, Rio Grande do Norte, Brasil.: [s.n.], 2020. Disponível em: <https://dl.acm.org/doi/10.1145/3422392.3422422>.

ALVES, G. et al. Ezpetri: A petri net interchange framework for eclipse based on pnml. In: . [S.l.: s.n.], 2004. p. 143–149.

BAIER, C. et al. Probabilistic model checking for energy-utility analysis. In: . [S.l.: s.n.], 2014. v. 8464.

BARBOSA, D. M. et al. Lotus@runtime: A tool for runtime monitoring and verification of self-adaptive systems. In: **2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**. [S.l.: s.n.], 2017. p. 24–30.

BINKERT, N.; BECKMANN, B.; AL. et. The gem5 simulator. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 39, n. 2, p. 1–7, ago. 2011. ISSN 0163-5964.

BöHM, C.; JACOPINI, G. Flow diagrams, turing machines and languages with only two formation rules. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 9, n. 5, p. 366–371, maio 1966. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/355592.365646>.

CLARKE, E.; WING, J. Formal methods: State of the art and future directions. **ACM Computing Surveys**, v. 28, 12 1996.

CLEMENTS, P. C.; NORTHROP, L. **Software Product Lines: Practices and Patterns**. [S.l.]: Addison-Wesley, 2001. (SEI Series in Software Engineering).

CORBETT, J. C. et al. Bandera: extracting finite-state models from java source code. In: **Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium**. [S.l.: s.n.], 2000. p. 439–448.

DAMASO, A.; ROSA, N.; MACIEL, P. Using coloured petri nets for evaluating the power consumption of wireless sensor networks. **International Journal of Distributed Sensor Networks**, v. 10, n. 6, p. 423537, 2014.

DAYARATHNA, M.; WEN, Y.; FAN, R. Data center energy consumption modeling: A survey. **IEEE Communications Surveys Tutorials**, v. 18, n. 1, p. 732–794, 2016.

DUARTE, L.; KRAMER, J.; UCHITEL, S. Using contexts to extract models from code. **Software & Systems Modeling**, v. 16, p. 523–557, 05 2017.

DUARTE, L. M. Behaviour Model Extraction using Context Information. n. November, 2007.

DUARTE, L. M. et al. A Model-based Framework for the Analysis of Software Energy Consumption. **SBES - Simpósio Brasileiro de Engenharia de Software**, 2019.

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**. New York, NY, USA: Association for Computing Machinery, 2018. (MiSE '18), p. 25–32. ISBN 9781450357357. Disponível em: <https://doi.org/10.1145/3193954.3193963>.

DUBSLAFF, C.; KLüPPELHOLZ, S.; BAIER, C. Probabilistic model checking for energy analysis in software product lines. In: **MODULARITY '14**. 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>.

EDER, K.; GALLAGHER, J. P. Energy-aware software engineering. In: FAGAS, G. et al. (Ed.). **ICT - Energy Concepts for Energy Efficiency and Sustainability**. Rijeka: IntechOpen, 2017. cap. 5. Disponível em: <https://doi.org/10.5772/65985>.

HANSEN, H.; VIRTANEN, H.; VALMARI, A. Merging state-based and action-based verification. In: . [S.l.: s.n.], 2003. p. 150– 156. ISBN 0-7695-1887-7.

HAREL, D. Statecharts: a visual formalism for complex systems. **Science of Computer Programming**, v. 8, n. 3, p. 231 – 274, 1987. ISSN 0167-6423. Disponível em: <http://www.sciencedirect.com/science/article/pii/0167642387900359>.

HILLSTON, J. **A Compositional Approach to Performance Modelling**. USA: Cambridge University Press, 1996. ISBN 0521571898.

JACKSON, D.; RINARD, M. Software analysis: A roadmap. In: . [S.l.: s.n.], 2000. p. 133–145.

JENSEN, K.; KRISTENSEN, L. M. **Coloured Petri Nets: Modelling and Validation of Concurrent Systems**. Dordrecht: Springer, 2009. ISBN 978-3-642-00283-0.

JOU, C.-C.; SMOLKA, S. A. Equivalences, congruences, and complete axiomatizations for probabilistic processes. In: BAETEN, J. C. M.; KLOP, J. W. (Ed.). **CONCUR '90 Theories of Concurrency: Unification and Extension**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990. p. 367–383. ISBN 978-3-540-46395-5.

JUNIOR, M. N. O. et al. Analyzing software performance and energy consumption of embedded systems by probabilistic modeling: An approach based on coloured petri nets. In: DONATELLI, S.; THIAGARAJAN, P. S. (Ed.). **Petri Nets and Other Models of Concurrency - ICATPN 2006**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 261–281. ISBN 978-3-540-34700-2.

KELLER, R. M. Formal verification of parallel programs. **Commun. ACM**, ACM, New York, NY, USA, v. 19, n. 7, p. 371–384, jul. 1976. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/360248.360251>.

KHALID, H. et al. What do mobile app users complain about? **IEEE Software**, IEEE, v. 32, n. 3, p. 70–77, 2015. ISSN 07407459.

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.). **International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS 2012)**. [S.l.: s.n.], 2002. p. 200–204.

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, D.; HALFOND, W. An investigation into energy-saving programming practices for android smartphone app development. **3rd International Workshop on Green and Sustainable Software, GREENS 2014 - Proceedings**, 06 2014.

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.

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.

LUDEWIG, J. Models in software engineering - an introduction. **Inform., Forsch. Entwickl.**, v. 18, p. 105–112, 04 2004.

LUNT, S. J. **FTP Security Extensions**. RFC Editor, 1997. RFC 2228. (Request for Comments, 2228). Disponível em: <https://www.rfc-editor.org/info/rfc2228>.

MARIANI, L.; PEZZè, M.; SANTORO, M. Gk-tail+ an efficient approach to learn software models. **IEEE Transactions on Software Engineering**, v. 43, n. 8, p. 715–738, 2017.

MICHAELSON, G. **Concepts of programming languages**. [S.l.: s.n.], 1990. v. 32. 230 p. ISSN 09505849. ISBN 9780131395312.

MILNER, R. **Communication and Concurrency**. USA: Prentice-Hall, Inc., 1989. ISBN 0131149849.

MILNER, R. **Communicating and Mobile Systems: The pi-Calculus**. USA: Cambridge University Press, 1999. ISBN 0521658691.

PEREIRA, R. et al. The Influence of the Java Collection Framework on Overall. 2016.

PINTO, G.; CASTOR, F. Energy efficiency: A new concern for application software developers. **Commun. ACM**, ACM, New York, NY, USA, v. 60, n. 12, p. 68–75, nov. 2017. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/3154384>.

POSTEL, J. **RFC0765: File Transfer Protocol Specification**. USA: RFC Editor, 1980.

POSTEL, J.; REYNOLDS, J. K. **RFC 959: File Transfer Protocol**. 1985. Obsoletes RFC0765 (POSTEL, 1980). Updated by RFC2228 (LUNT, 1997). 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>.

SEGALA, R. Modeling and verification of randomized distributed real -time systems. 03 2007.

SEIDL, M. **UML@Classroom: An introduction to object-oriented modeling**. [S.l.: s.n.], 2015. v. 1555. 4–5 p. ISSN 16130073. ISBN 9783898647762.

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, 12 2015.

UCHITEL, S.; KRAMER, J.; MAGEE, J. Behaviour model elaboration using partial labelled transition systems. **ACM SIGSOFT Software Engineering Notes**, v. 28, p. 19, 09 2003.

VARDI, M. Y. Automatic verification of probabilistic concurrent finite-state programs. In: **FOCS**. IEEE Computer Society, 1985. p. 327–338. ISBN 0-8186-0644-4. Disponível em: <http://dblp.uni-trier.de/db/conf/focs/focs85.html#Vardi85>.

WALKINSHAW, N.; TAYLOR, R.; DERRICK, J. Inferring extended finite state machine models from software executions. In: **2013 20th Working Conference on Reverse Engineering (WCRE)**. [S.l.: s.n.], 2013. p. 301–310.

APPENDIX A — RESUMO EXPANDIDO

O consumo de energia de software está se tornando uma questão essencial durante o desenvolvimento e evolução de software, em particular, devido às várias restrições impostas pelas plataformas e requisitos de aplicativos (KHALID; SHIHAB; NAGAPPAN; HASSAN, 2015). Por esta razão, consumo energia de software está se tornando um importante aspecto para ser analisado durante o desenvolvimento e a manutenção do software. Portanto, desenvolvedores necessitam mais conhecimento sobre o consumo de energia para melhorarem a eficiência energética de seu software. No entanto, ainda existe pouco suporte para ajudar os desenvolvedores a entender como alguns pequenos fatores podem afetar a eficiência energética. Isso acontece, principalmente, devido à ausência de abstrações apropriadas para modelar e analisar o comportamento relacionado ao consumo de energia.

Uma maneira de abstrair e analisar melhor os custos energéticos é utilizando modelos de comportamentos (UCHITEL; KRAMER; MAGEE, 2003). Com esses modelos, é possível analisar diferentes componentes em diferentes níveis de abstração e verificar propriedades do sistema que dificilmente poderiam ser verificadas diretamente no código. Além disso, os modelos podem ser usados como documentação do software, em caso de mundanças, possibilitando comparações entre versões.

No trabalho de Duarte, Alves, Maia e Silva (2019), um framework baseado em modelos para análise do consumo energético de software é proposto. O framework consiste em coletar as informações de energia, adicioná-las em um Labelled Transition System (LTS) (KELLER, 1976) e analisar propriedades de interesse, tais como: custo de uma execução, lista de execuções que atedem a um limite de energia, valor médio de todas execuções com desvio padrão e variação, entre outras. Contudo, os autores não descrevem como construir o modelo com custos energéticos a partir do código fonte tal que represente o comportamento real do sistema. Portanto, o modelo é construído de acordo com o próprio conhecimento do desenvolvedor, definindo o que é relevante para ser modelado. Porém, dependendo das escolhas do desenvolvedor, os custos de energia modelados podem ter inconsitências, não garantindo resultados confiáveis, pois qualquer análise sobre um modelo que não represente o comportamento real de um sistema pode ser enganosa (JACKSON; RINARD, 2000). Por isso, é necessário a construção de um modelo acurado, ou seja, de um modelo que represente tão fielmente quanto possível o comportamento do código real, de forma que os resultados sejam confiáveis e indiquem verdadeiros pontos

de consumo de energia que impactam no sistema. Nesse sentido, é necessário direcionar os desenvolvedores em como construir esses modelos e usá-los para auxiliar em melhores decisões a respeito dos custos energéticos de seu sistema.

Este trabalho propõe uma abordagem para apoiar a construção de modelos de comportamento energético a partir de código. Para isso, desenvolvemos um modelo chamado ELTS (*Energy Labeled Transition System*), que é um LTS com a adição de informações de custos de energia. Para construir este modelo a partir de código Java, criamos o conceito de unidade básica de energia (*basic energy unit* - BET, em inglês), que permite associar partes do código a elementos do ELTS. A proposta da BET é auxiliar a análise de possíveis pontos de energia de interesse mantendo rastreabilidade, de forma a simplificar a identificação de qual parte do código corresponde à parte do modelo que requer atenção, de acordo com análises no modelo. Representamos a semântica do código no modelo por meio de relações entre BETs, definidas como sequência, condicional e iteração. Essas relações possibilitam um melhor entendimento do comportamento ao analisar o modelo e representam as conexões das BETs que o compõem, facilitando também a construção do modelo de maneira composicional. Além disso, propomos como alterar localmente o nível de abstração do modelo. Assim, podemos expandir somente uma parte do modelo, mantendo o restante como está, tornando simples mudar localmente o nível de abstração.

Para mostrar a aplicabilidade da abordagem foram realizados 3 experimentos. Os dois primeiros, modelam componentes isolados, um para enviar um arquivo utilizando o protocolo FTP e outro focado na compressão de arquivos. No último experimento, os modelos dos dois primeiros experimentos são combinados e é avaliado o impacto nos custos energéticos. Por meio desses experimentos, foi aplicado o passo a passo da abordagem proposta, que possibilitou construir modelos a partir do código fonte. Nos 3 experimentos foi possível encontrar os maiores pontos de consumo energético e suas respectivas linhas, ou seja, o usuário saberia exatamente quais linhas estariam provocando o maior consumo e poderia tentar realizar alguma modificação. No primeiro experimento, o gargalo era nas funções de envio do arquivo, enquanto que, no segundo, a iteração que comprimia o arquivo causava o maior consumo. No exemplo utilizando composição, percebe-se que o consumo energético como um todo aumentou, porém, se analisar isoladamente somente o modelo que representa o FTP o consumo diminiu, portanto, em um cenário em que o arquivo seria comprimido somente uma única vez, seria uma boa opção. Contudo, se a compressão for realizada para cada envio, torna-se energeticamente muito custoso.

A mudança do nível de abstração de parte do modelo é realizada conforme o pos-

sível ou necessidade do desenvolvedor de descobrir o real ponto energético de maior consumo. O uso da mudança de níveis de abstração permite identificar pontos específicos de consumo e, em um cenário real, serviria para guiar possíveis alterações no software. Além disso, é possível combinar dois modelos construídos independentemente e usar este modelo composto para obter informações relevantes sobre o sistema resultante da combinação, conforme demonstrado nos experimentos.

Por meio dos resultados apresentados neste trabalho, obteve-se uma melhor demonstração de como construir um modelo com custos de energia passo a passo com base na definição de BETs e suas relações, consequentemente gerando um modelo que descreva o mais próximo possível o real comportamento energético do sistema. Além disso, este trabalho forneceu a possibilidade de alteração do nível de abstração de uma parte do modelo para entender melhor o consumo de energia da parte correspondente do código, identificando possíveis pontos de interesse e onde modificar o código para melhorar a eficiência energética. Como trabalho futuro, seria interessante investigar como combinar e construir informações probabilísticas com energia, e realizar análises mais profundas, com intuito de descobrir informações que não seriam triviais. Temos de investigar melhor a questão de composição de modelos e suas implicações. Também seria interessante automatizar o processo da construção do modelo, facilitando para os desenvolvedores nesta etapa e, evitando possíveis erros que poderiam ocorrer.