A Methodology to Develop Energy Adaptive Software Using Model-Driven Development

Fumiya Tanaka*‡, Kenji Hisazumi*†§, Shigemi Ishida*¶ and Akira Fukuda*†‖
*Graduate School and Faculty of Information Science and Electrical Engineering
Kyushu University, Motooka 774, Nishi-ku, Fukuoka 819-0395, Japan
†System LSI Research Center,
Kyushu University, Motooka 774, Nishi-ku, Fukuoka 819-0395, Japan
‡Email:tanaka@f.ait.kyushu-u.ac.jp
§Email:nel@slrc.kyushu-u.ac.jp
¶Email:ishida@f.ait.kyushu-u.ac.jp
‖Email:fukuda@f.ait.kyushu-u.ac.jp

***Abstract*—In embedded system development, a crucial task is to reduce the maximum power consumption owing to power source limitations while maximizing the quality of service. The tradeoff between power consumption and quality of service needs to be resolved. If software can change its power consumption in accordance with the power consumption of hardware, it can reduce the maximum power consumption while increasing the quality of service. In this paper, we propose a model-based Development methodology for software with self-adaptive power consumption. With the proposed method, software changes its behavior during runtime by linking state-machine diagrams described by Executable UML to a feature model used in Software Product Line development. This method makes it possible to change the power consumption due to software according to the power consumption of the whole target device. The target software can maximize the quality of service under certain power constraints. Therefore, the target software can satisfy the tradeoff between power consumption and quality of service. Evaluation results showed that the average response time was about 0.22 s, and the adaptive rate was about 87.6%.**

***Index Terms*—Embedded System, Self-adaptive Software, Model-Driven Development, Energy Analysis**

I. Introduction

In embedded system development, size limitations and manufacturing costs are common issues. In the case of battery-driven devices, the battery size is restricted by the hardware size. This scales down the battery storage capacity and shortens the operating time. The operating time is also shorted when the embedded system provides multiple functions and is used in many types of scenarios. With this backgrounds, the power consumption needs to be reduced as much as possible to meet the uptime requirements. On the other hand, there is a tradeoff between the power consumption and quality of service. In embedded software development, there is a need to both reduce the power consumption while improving the quality of service. If the software can change its power consumption according to the power consumption of the hardware, it can minimize power consumption while maximizing quality of service.

Self-adaptive software development is one of the solutions to this challenge. The runtime circumstances constantly change. Therefore, the developer cannot describe the best behavior in the development phase. An application software in an embedded system should automatically behave appropriately according to current environment because not all possible patterns can be accounted for in the design phase. Self-adaptive software can change its behavior depending on the runtime circumstances. In this paper, we propose self-adaptive behavior in accordance with the power consumption of other software or the situation of the power supply.

To facilitate self-adaptive software development, we introduce the Model-Driven Development (MDD) [2] and Software Product Line (SPL) [3] development methodologies. MDD uses executable and translatable models to improve the efficiency and quality of software development. We can verify software described with models in the early stage of development to reduce reworking costs and increase the quality. We also can generate final codes from the models. SPL helps with optimizing the development of the entire software family. Despite its name, the technique can also be applied to other kinds of system development, such as hardware and systems. This approach commonly employs a feature model to determine the commonality and variability among software groups. The variability of a feature model represents the difference among software groups, and the production efficiency of the entire product family can be improved by designing a combination of common and variable parts for features during the design phase. In this paper, we propose a model-based software development methodology for self-adaptive power consumption. In the method, the runtime variation is identified with a feature diagram used in SPL. State-machine diagrams described by Executable UML [4] are linked to the feature model. During the design phase, the developer assigns features

This is an accepted version of the paper.

doi: 10.1109/TENCON.2017.8227963

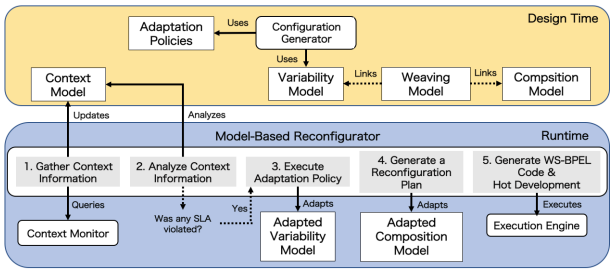

Fig. 1. Overview of Related Works

to software behavior and describes different behaviors for each variation in the state-machine diagram. The behavior can be changed depending on the value estimated with the model-based analysis. During runtime, the target software can change its behavior by dynamically selecting features according to the power consumption status. Software can be written to perform optimally in terms of power consumption and service quality by changing its behavior depending on the situation of the power supply or the power consumption of other software.

The remainder of this paper is structured as follows. Section II describes the application of the SPL approach to the development of self-adaptive software in existing research. Section III presents our proposed model and the SPL-based self-adaptive software development method. Section IV presents an evaluation to demonstrate the feasibility of our proposed method. Section V concludes this paper.

II. Related Works

This section describes related works on self-adaptation using the SPL technique. [5]

A. Dynamic adaptation with variability models

A Web service framework for dynamic adaptation of the service configuration at runtime has been proposed in the literature. This framework uses techniques to handle the operation of Web services as features. Fig. 1 shows the overall framework.

We can use this framework during the design phase and runtime. In the design phase, models are generated to guide the dynamic adaptation during the runtime. During runtime, these models are used for adaptation. The framework of the Model-based Reconfiguration Engine for Web Service (MoRE-WS) is used for adaptation during the runtime. The flow of the adaptation by MoRE-WS is described below.

1) The context model is updated according to changes in the context that are detected by the CONTEXT MONITOR.
2) Information from the context model is used to judge whether or not the service level agreement (SLA) is violated.
3) If the SLA is violated, the framework activates or deactivates features and adapts the variable feature model according to the adaptation rules.

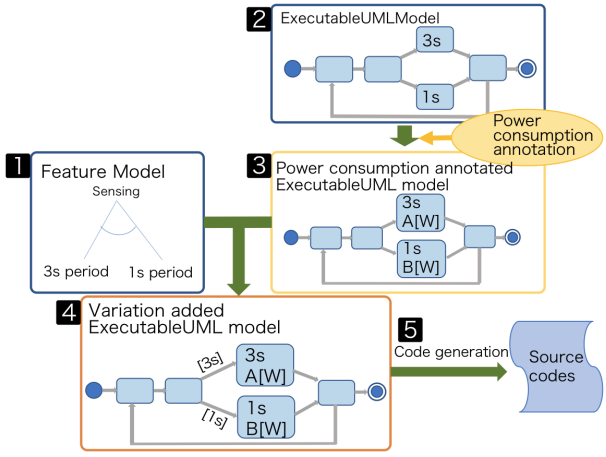

Fig. 2. Overview of proposed method

4) The framework uses the adapted feature model to readjust the service configuration.
5) Fragments of the WS-BPEL code are added or deleted to reflect the adjusted service configuration model.

The framework described below has problems with handling low-level information close to hardware because it adapts functions with a high level of abstraction. For example, the power consumption of the device cannot be handled as context because it is close to implementation. There are methods for dealing with power consumption at the software level, but there is no development methodology with a self-adaptive performance to deal with power consumption.

B. Challenges of Related Works

The framework described below is problematic in handling with low-level information close to hardware because this framework realizes adaptation function with a high level of abstraction.

For example, the power consumption of the device cannot be handled as a context in the research because it is close to implementation.

Further, there is a method dealing with power consumption at the software level, but there is no development methodology with a self-adaptive performance by dealing with power consumption.

III. Proposed Method

In previous research, it was impossible to adjust the power consumption because information close to implementation could not be handled. In the proposed method, the software self-adapts its power consumption by using the Model-Based Energy Analysis Method.

A. Overview

An overview of the development process for our proposed method is described below. Fig. 2 shows the overview of the proposed method.

First, in the design phase, the developer creates a feature model that includes variability regarding power consumption.

Next, a model of the behavior is created by using Executable UML. We employ a state-machine diagram with variable points to model software with various levels of power consumption. Next, a Power Consumption Annotated Model is created by adding a power information to Executable UML model. Next, a Variation-Added model is created by associating two models: a feature model and the Power Consumption Annotated Model. Then, this model is converted into source code by automatic code generation based on the MDD method. During runtime, this software operates below the required power consumption while dynamically changing the variation in response to the power situation.

B. Creation of Feature Model

This section describes the creation of the feature model. The feature model expresses features representing the functionality or non-functionality of the software system and the relationship between them with a directed graph. The feature model allows the commonality and variability of software in a software family to be determined. We employ a feature model to describe the dynamic variable behavior of software during runtime.

Our method models runtime functions as features. Common behaviors during runtime are modeled as common features, and different behaviors depending on context are modeled as variable features. The variable features are classified as alternative or optional. Alternative features are selected from a group of possible features, and optional features are or are not selected according to the situation. The feature model is represented by a tree structure of common and variable components.

The software consists of common components or selected variable components. In the feature model, a tree with no selectivity, which is produced by selecting features at each variable point, has all of the software components. With the proposed method, this no-selectivity tree expresses one variation of behavior.

C. Design of the Executable UML Model

Here, we describe the design of the Executable UML model. The behavior model that realizes functions of the software and model monitoring the power consumption situation are described with Executable UML diagrams. The monitoring model considers periodic behavior to obtain the power consumption. In the behavior model, the states required for self-adaptation are provided for each variable point. Required states are those that transition to the selected state and that are determined by variations in the power consumption. Information on determining the variations are appended to the model presented in the next section. The two models operate in parallel asynchronously at runtime. The operation flow in the Executable UML model is described below.

1) The software behavior is assigned to a state, and the variation is represented as a difference in the state.

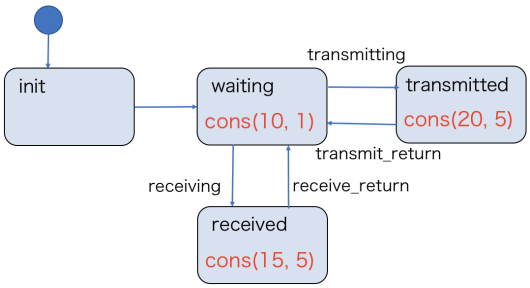

Fig. 3. Overview of Model-Based Energy Analysis Method

2) One state is installed to handle the transition to the selected variation in front of the group representing variations in the state.
3) One state is installed to determine the variation by referring to the power consumption status in front of the state added in Step 2.
4) The state-machine diagram is described as a model to monitor periodic behavior.
5) The behavior model is verified as to whether it is operating correctly.

D. Power Consumption Annotated ExecutableUML Model

Here, we describe the Power Consumption Annotated Executable UML model, which appends the estimated power consumption from the Model-Based Energy Analysis Method [6].

This method can estimate the energy consumption in a particular state by using Executable UML. The energy consumption of each state is estimated from the resource consumption model and power consumption model. In the design phase, this method can estimate the energy consumption at the state level; therefore, it can bottleneck the power consumption. It is possible to estimate the energy consumption with an average estimation error of 9.0% with this method. Fig. 3 shows an overview of the Model-Based Energy Analysis Method. This method is used to estimate the power consumption of each variation in behavior at the modeling stage. The estimated value is appended to the Executable UML model to be used as a reference for changing behavior. This is defined as the Power Consumption Annotated Executable UML model.

E. Association of Features and States

This section describes the association of features and states. In this method, the created feature model is associated with the Power Consumption Annotated Executable UML model to generate a new Executable UML model.

Features and states are associated by a state-transition diagram with variable points. This diagram represents the presence or absence of a state transition as variability in the guard condition. The presence or absence is changed by the activation or deactivation of features. In the feature model, a tree representing one variation can be obtained by selecting a single feature at each variable point. In the state-transition diagram with variable points, only transitions depending on the

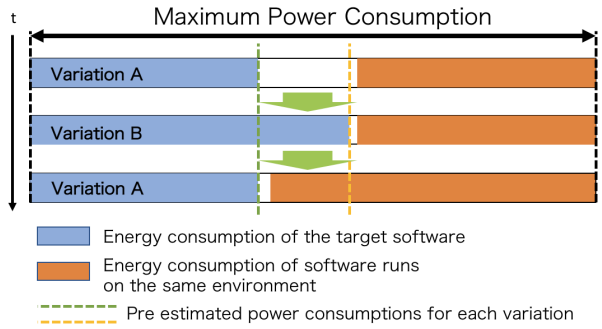

Fig. 4. Runtime behavior

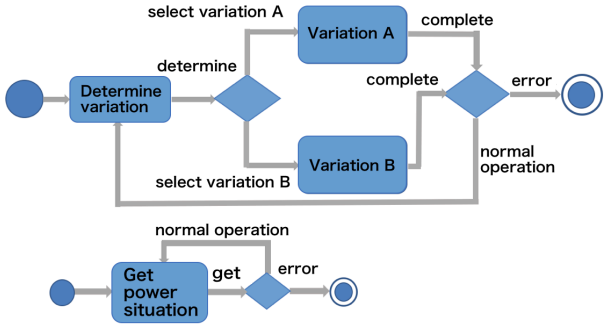

Fig. 5. Power situation acquisition and variation change

selected feature is enabled because other transitions are limited by the guard conditions. Therefore, in the state-transition diagram with variations, only transitions to a certain variation can be carried out. As a result, a state-transition diagram that enables transition depending only on the selected features is generated. Each feature is associated with a state because the feature tree and state-transition diagram have a one-to-one correspondence. The generated model is defined as a Variation-Added Executable UML model.

F. Runtime behavior

This section describes the runtime behavior of the target software developed by proposed method. The software modifies its own behavior while acquiring the power consumption status of the device. The runtime behavior is changed based on information estimated in advance. Variations that operate at the maximum performance below the maximum required power consumption during runtime are selected. Fig. 4 shows the behavior changes during runtime.

Fig. 4 shows that variation B has a higher power consumption and higher performance than variation A. There is a tradeoff between the power consumption and performance for variations A and B. The device running variation A changes its behavior to B if the latter is determined to cause no problem to the reference power status. The software reduces its maximum power consumption by changing to variation A when running variation B is determined to cause problems because of the increased power consumption of other software.

G. Obtaining the Power Consumption during Runtime and Changing the Variation

To change the behavior during runtime, the power consumption during execution needs to be obtained, and the variation needs to be changed. A mechanism for implementing Executable UML models needs to be installed during the design phase. This section describes how to obtain the power consumption during execution and how to change the variation.

The behavior of obtaining the power situation is described by the monitoring state-machine diagram and is periodically performed. The runtime power situation can be obtained by applying the Model-Based Energy Analysis Method to the resource consumption during a one-time execution.

The variation is changed at each variable point. The variation is determined by a function that refers to the latest power consumption situation. The decision function selects a variation that generates the maximum performance without exceeding the maximum required power consumption according to the power consumption status. Fig. 5 shows the state-machine diagram for changing variations and obtaining the power situation.

TABLE I
EVALUATION ENVIRONMENT

Equipment type	Equipment name
Device	Raspberry Pi Model B+
Ammeter	Agilent 34411A 61/2 Digit Multimeter
Power supply	HEWLETT PACKARD 3616A DC POWER SUPPLY
Add-in board	Grove Pi+
Sensor module	Grove - Temperature Sensor v1.2

IV. EVALUATION

In this section, we describe the evaluation of the target software. BridgePoint was used to develop the evaluated software; this is an Executable UML modeling tool. The expression (1) was used as the power consumption model to create the evaluated software. After the data for the changing power consumption were obtained by the stepwise program, each parameter of the power consumption model was determined by the least squares method.

$$Power(A) = 0.279 + 0.0007 \times CPU(\%) + 0.0071 \times Wi\text{-}Fi(MB/sec) \quad (1)$$

A. Evaluation environment

Table I presents the evaluation environment. A Raspberry Pi was connected to the power supply and ammeter. The sensor module was connected to Grove Pi+'s analog device connected to the Raspberry Pi.

In this experiment, BridgePoint [7] was used to model the evaluated software. BridgePoint is a tool that incorporates the methodology of Executable UML. The model can be verified

TABLE II
EVALUATION ITEM

Item	Contents
Response Time	Time until the variation change from the power situation change
Error Rate	The number of times of variation changes that do not meet the power situation per second
Correction Time	Time of changes to the right variation from the occurrence of errors
Adaptive Rate	Time ratio of the correct variation
Estimation Error	The relative error rate of the power estimation
Overhead	Power consumption increase rate by having a self-adaptive

in the modeling stage. This tool can automatically generate code from a model and has been used in MDD.

In BridgePoint, a state-machine diagram is described as class diagrams with conditions by the Executable UML methodology. State-machine diagrams can describe behavior in a state in detail with an Action Language. An Action Language has a high degree of abstraction. It does not depend on the type of generated source code. The model can be connected to an external code by using a Bridge and Function. We adopted BridgePoint as a development tool because it can perform the complete MDD with an Action Language, Bridge, and Function.

B. Evaluation Items

The evaluation items for this evaluation experiment are described here. The target software required higher accuracy and faster adaptation. In addition, the power consumption of the software would be increased by the addition of self-adaptation. Based on these conditions, we evaluated six items, as presented in Table II.

C. Evaluation Procedures

The evaluation procedures for experiments 1 and 2 are described below.

1) Experiment1: The evaluation procedure of experiment 1 is presented below. The power consumption was measured by connecting an ammeter to the Raspberry Pi, and the execution time when the variation was changed was recorded.

1) The evaluated software was run while a test program that changes the CPU utilization was run.
2) The current values at runtime were recorded, and values and variations were estimated.
3) The time when the variation change should occur was calculated and compared with the recorded time in order to evaluate evaluation items 1-4.
4) The estimation error was estimated by comparing the measured and estimated values.

2) Experiment2: The evaluation procedure of experiment 2 is presented below. The software generated in experiment 1 was used.

1) The software was run without the self-adaptive function, and the current values were measured.
2) The software was run with the self-adaptive function, and the current values were measured.
3) The two measured current values were compared to evaluate the rate of increase in the power consumption.

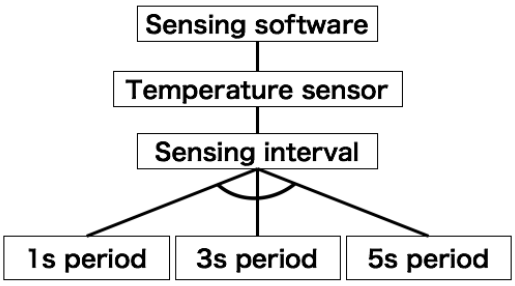

Fig. 6. Feature diagram

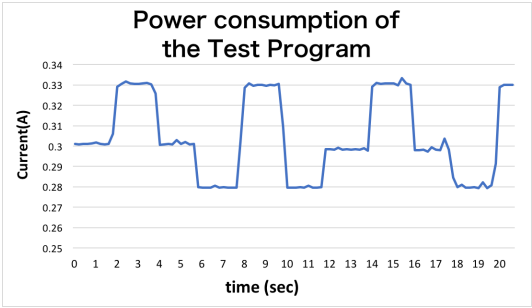

Fig. 7. Power consumption of the test program

D. Software

In experiment 1, two software programs were used: the evaluated software and test program. The specifications of each software are described below.

1) Evaluated Software: The evaluated software used a temperature sensor for sensing. It treated the power consumption of Raspberry Pi as the context and the sensing intervals as variations in the adaptation. This software had three types of sensing intervals as variations. Fig. 6 shows the feature model of the evaluated software.

2) Test Program: The test program changed the CPU utilization to adjust the power situation of the device treated by the evaluated software as context. This software was executed in 2 s and had three stages of power consumption because the evaluated software had three variations. Fig. 7 shows the power consumption when the test program was run alone.

E. Evaluation Result

Table III presents the evaluation result. The response time in the experiments was 0.22 s on average. There were 11 errors when the test program was executed for 80 s, and the correction time for each error was 0.20 s on average. The adaptive rate was 87.6% with respect to the power consumption. The relative error of the power consumption estimation was 1.52% on average. Fig. 8 compares the measured and estimated power

TABLE III
EVALUATION RESULT

Item	Result
Response Time(sec)	0.22
Error Rate(times/sec)	0.14
Correction Time(sec)	0.20
Adaptative Rate(%)	87.6
Estimation Error(%)	1.52
Overhead(%)	2.12

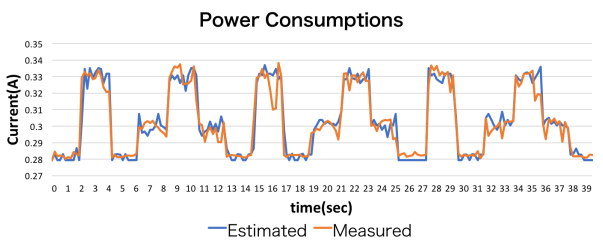

Fig. 8. Comparison between actual value and estimate at runtime

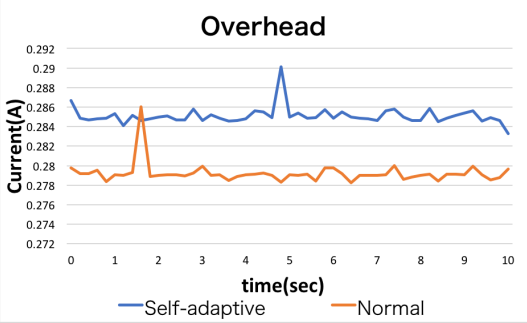

Fig. 9. Evaluation of overhead

consumption. We evaluated the overhead of self-adaptability. The increased rate of power consumption was 2.12% compared to the software without the self-adaptive function. Fig. 9 shows the evaluation results for the overhead. The adaptive rate was 87.6%, which is acceptable. The overhead due to the addition of the self-adaptability was 2.12%, which is in the allowed range.

V. CONCLUSION

An embedded system must reduce its power consumption to meet requirements associated with hardware limitations during the runtime. However, there is a tradeoff between the power consumption and quality of service. In software development, both must be optimized. We used self-adaptive software to meet this tradeoff. Self-adaptive software can change its behavior depending on the behavior of the surroundings. However, existing model-based methods cannot handle the information from self-adaptive software or use it to adjust the power consumption of the device. This is because information about power consumption cannot be handled at upstream stages like modeling. In this study, we developed software to handle the information about power consumption in the modeling stage and proposed a model-based method to develop self-adaptive software for power consumption. This method uses the SPL technique and incorporates the feature model into the UML model. By determining the behavior of the software by activating and deactivating alternative features, it is possible to realize self-adapting power consumption of a device during runtime.

With the proposed method, both the power consumption and quality of service are optimized. We developed software based on the proposed method by using BridgePoint and evaluated it by running it concurrently with a test program. In the results, the average response time was 0.22 s, the adaptive rate was 87.6%, and the estimated error was 1.52%. The overhead due to the addition of self-adaptability was 2.12%.

For future works, the following can be considered:

- Evaluation of software with multiple variable points.
- Optimization of the tradeoff between power consumption and performance.
- Association between optional features and the UML model.

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number 15H05708.

REFERENCES

[1] F. D. Macías-Escrivá, R. Haber, R. del Toro, and V. Hernandez, "Self-adaptive systems: A survey of current approaches, research challenges and applications," Expert Systems with Applications 40. 18, pp.7267-7279, 2013.
[2] J. A. Estefan, "Survey of model-based systems engineering (MBSE) methodologies," Incose MBSE Focus Group 25. 8, 2007.
[3] K. Pohl, G. Böckle, and F. J. van der Linden, "Software Product Line Engineering: Foundations, Principles and Techniques," Springer-Verlag , 2005.
[4] S. J. Mellor, M. Balcer, and I. Jacoboson, "Executable UML: A foundation for model-driven architectures, " Addison-Wesley Longman Publishing Co., Inc., 2002.
[5] G. H. Alfred, V. Pelechano, R. Mazo, C. Salinesi, and D. Diaz, "Dynamic adaption of service compositions with variability models," the Journal of Systems and Software, pp.24-47, 2014.
[6] R. Yoshimoto, T. Kadono, K. Hisazumi, and A. Fukuda, "A Software Energy Analysis Method Using ExecutableUML," IEEE TENCON 2016, pp.218-221, 2016.
[7] BridgePoint HomePage, https://xtuml.org/(Last access 2017/2/9)