ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2008, Vol. 37, No. 4

DESIGN OF ONTOLOGY-BASED GENERATIVE COMPONENTS USING ENRICHED FEATURE DIAGRAMS AND META-PROGRAMMING

Vytautas Štuikys, Robertas Damaševičius

Software Engineering Department, Kaunas University of Technology
Studentų 50, LT-51368, Kaunas, Lithuania

Abstract. A product line (PL) approach is emerging as the most promising design paradigm for embedded software design domain, where a great variability of requirements and products exists. The implementation of the PL approach requires thorough domain analysis and domain modelling. We propose to represent embedded software components using Enriched Feature Diagrams (EFDs). EFDs are an extension of traditional Feature Diagrams (FDs) for explicit representation of domain variability enriched with contextualization and domain ontology. We suggest to transform feature models described using EFDs into generative component specifications encoded using the meta-programming techniques. A case study from the embedded software specialization domain is presented.

Key words: feature diagram, generative component, domain ontology, product line, embedded software.

1. Introduction

Current approaches for architectural design of systems or components (either instances or generative ones) predominantly use the product line (PL) concept. A software PL is a set of software systems that share a common, managed set of features satisfying the specific needs and are developed from a common set of core assets in a prescribed way [1]. The concept of PLs, if applied systematically, allows for the dramatic increase of software design quality, productivity, provides a capability for mass customization and leads to the 'industrial' software design [2].

The key for the PL implementation is the use of domain analysis and domain modelling methods. From the computer science and software engineering perspective, a vast majority of the methods (e.g., FODA [3], FORM [4], FAST [5], etc.) exploit such domain properties as scope, commonality and variability [6]. These concepts enable to express the content in the form of features and to model the domain well through the identification of structural, functional and other characteristics (otherwise features) and their relationships. However, with the further expansion of complexity, which is inspired by ever-growing technology capabilities, market demands, user requirements, new appliances (e.g., ambient intelligence, mobile computing), etc., it is not enough to rely on the domain content-based and feature-centric analysis in the system development only.

What is needed is the extension of the scope of analysis in order to extract along with the content-oriented features the other domain relevant knowledge that may be, for example, related with the context of use. Context awareness is a very important feature in such appliances as ambient intelligence [7], e-learning systems [8], knowledge-based information systems and many others, because this kind of information hides (if it is not yet revealed) or brings more complex relationships of features that can be treated as knowledge (if this information is already revealed). In a broader sense, context analysis is a matter of cognitive science. Software engineering approaches (such as FODA method for domain analysis) do not also neglect the importance of the context; however, these approaches deal with the context in a narrow sense (usually statically as, e.g., FODA that neglects possible changes in the context).

What we suggest in this paper is: 1) to represent embedded software components using Enriched Feature Diagrams (EFDs), an extension of traditional Feature Diagrams (FDs) for the explicit representation of domain variability enriched with contextualization and domain ontology; 2) to transform such feature models described using EFDs into generative component specifications encoded using the meta-programming techniques.

The paper is organized as follows. Section 2 discusses the related work. Section 3 describes the basics of Feature Diagrams and motivates for their extension.

Section 4 describes Enriched Feature Diagrams (EFDs). Section 5 discusses attributes of the ontology-based generative components. Section 6 describes encoding of EFDs using meta-programming. Section 7 presents a case study. Finally, Section 8 presents evaluation, discussion and conclusions.

2. Related work

We categorize basic related works into three research streams: 1) approaches that deal with feature diagrams (FDs) and ontology-based representation of commonality-variability for embedded software; 2) analysis of specific requirements and methods, such as power optimization for embedded software; 3) generative approaches for implementing systems and components.

Stream 1. This stream has a direct link with domain analysis methods (FODA, FORM, FAST) and PL approaches already referred to in the previous section. The origins of FDs can be traced back to the FODA methodology in 1990 [9]. Since then, they have undergone several extensions [3, 4, 10, 11, 12] intended to improve their expressiveness. FDs first were applied in the context of industrial manufacturing product lines, e.g. for modelling car assembly lines. Later, the idea was extended to software product lines (PLs).

Based on the success of feature modelling and PL approach in industrial manufacturing and software engineering domains and intention to introduce product families in System-on-Chip (SoC) design [13], we propose using FDs for specification, representation and structuring of generative embedded software components. Furthermore, FDs are also important for constructing domain ontologies by providing views on ontologies [14, 15] in order to acquire a common understanding of the domain. Ontology is a conceptual specification that describes knowledge about a domain [16]. The construction of such ontologies allows providing shared and common understanding of a specific domain, and facilitates knowledge sharing and reuse.

Domain ontologies, where domain knowledge is represented as ontology trees, have some syntactic and conceptual resemblance with feature hierarchies represented using FDs [17]. However, FDs have weaker capabilities to express various relationships in representing knowledge [14]. On the other hand, when those capabilities are not enough, FDs can be easily combined with more powerful domain knowledge representation methods, such as fuzzy logic [18].

Stream 2. Specifically to embedded software design for mobile computing, energy consumption is a major cost when running some large scale applications [19], though other characteristics (execution time, accuracy, memory) remain as important as ever. Combined together those characteristics are highly influential to domain ontologies and should be included in the requirement statement. We restrict ourselves by providing power analysis methodologies that are relevant to the application level only. The method [20] to the application level only. The method [20] relies on application-level observations of battery dissipation for a representative set of benchmarks by showing of how these benchmark dissipation rates can be combined to form an estimate for an arbitrary program. Another approach [20] enables generation of the energy-efficient code based on an instruction level model that quantifies the energy cost of individual instructions and of various inter-instruction effects.

Profiling-based power optimization methods use profiling tools that are generally applied at several levels of abstraction: user [21], operational [22], algorithmic, data and instruction-level [23]. Application-level profiling can be used for dynamically modifying application's behaviour to conserve energy [24]. Energy profiling, automated data representation conversion, derivation of polynomial representation and symbolic algebra is combined by Peymandoust *et al.* [25]. In this approach energy profiling is necessary to identify critical sections of code that needs to be optimized. For more complex arithmetic functions, the symbolic algebra techniques decompose the polynomial representation of the basic blocks of a program into a set of instructions available on the embedded processor that automates energy and performance optimization of the arithmetic sections of source code.

In [26], power consumption is optimized using two well-known transformation methods: loop unrolling, where it aims at reducing the number of processor cycles by eliminating loop overheads, and loop blocking, where it breaks large arrays into several pieces and reuses each one without self interference. Compiler optimizations such as linear loop transformations; tiling, unrolling, fusion, fission and scalar expansion are also considered in [27]. However, only loop unrolling is shown to decrease the consumed energy. Software pipelining and recursion elimination for energy optimization are also considered in [28]. Various code transformations for software power optimization are discussed in [29]. For the application of trigonometric functions in real-time ES software, typically both the numerical precision and the resource demands are relevant [30]. For the following discussions of different power optimization techniques we concentrate on the cosine function, since other trigonometric functions can be directly derived from it.

Stream 3. There is a broad discussion on generative approaches. Most relevant works to our paper are: generative programming [10], aspect-oriented programming [31], frame-based technology [32] and meta-programming [33]. A common usage of meta-programming is to provide mechanisms for designing generic (generative) components [34], i.e. explicitly implementing generalization in the domain. Domain language implements commonalities in a domain, while a meta-language allows developers to specify variations to be implemented in the domain system. Thus meta-programming provides means for implementing domain commonalties and variability at the generic component implementation level, which are

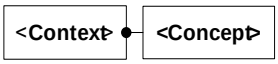
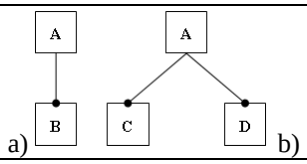
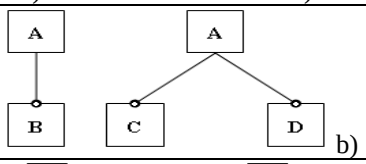
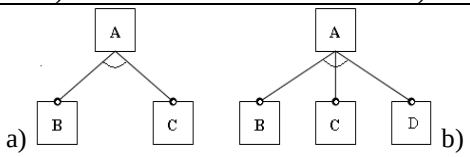
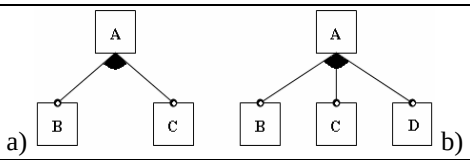
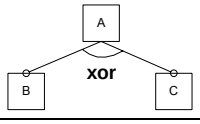
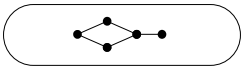
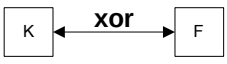
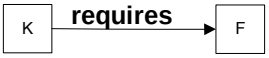
specified graphically by FDs at the generic component design level.

3. Basics of Feature Diagrams and motivation of their extension

Conceptually, when applied in modelling, the notation of FDs represents the domain model that describes the architecture of system or component at a higher abstraction level. A conventional FD [3] is a tree-like directed acyclic graph, in which the root represents the initial concept (also referred to as domain), intermediate nodes represent compound features, and leaves represent non-decomposable atomic features that may have values (aka variants); branches represent the parent-child relationships among compound features or among compound features and atomic features. Furthermore, some additional relationships such as constraints (e.g., <require>, <mutual exclusion>, etc.) between leaves derived from different parents are identified.

FDs are a graphical notation. Features are denoted by boxes. Features differ in types. There are mandatory, optional and alternative feature types. Mandatory feature is the one which always is selected (it is marked by a black circle above its box). Optional feature is the one which may be selected or not. Alternative feature is the one which is selected depending on some alternative (condition). Both are marked by a white circle above its box (see Table 1). If atomic feature has values (variants), it is also treated as a variant point.

Table 1. Feature types, ontology and constraints for feature model representation

Feature type	**Definition, formalism and semantics of relationships**	**Graphical notation (syntax)**
Concept and its context	Concept is represented by the root with the explicitly stated context on the left at the same level; context is seen as the highest mandatory feature with variants	
Mandatory	Feature B (C, D) is included if its parent A is included: a) *if* A *then* B; b) *if* A *then* C&D; (Relationship-and: <**R-and**>)	
Optional	Feature B (C, D) may be included if its parent A is included: a) *if* A *then* B *or* <**no feature**>; b) *if* A *then* C *or* D *or* <**no feature**>	
Alternative 1	Exactly one feature (B or C or D) has to be selected if its parent A is selected: a) *if* A *then* case-of (B, C) ; b) *if* A *then* case-of (B, C, D) ; (Relationship-case: <**R-case**>)	
Alternative 2	At least one feature has to be selected if its parent A is selected: a) *if* A *then* any-of (B, C); b) *if* A *then* any-of (B, C, D); (Relationship-or: <**R-or**>)	
Alternative 3	*if* A *then* (B *but* ¬C) *or* (C *but* ¬ B); (Relationship-xor: <**R-xor**>; differs from R–case by: 1) having two sons only; 2) label "xor" is written at the father's node)	
Ontology	A compound of atomic features and their relationships; ontology expresses the domain knowledge in some way	
Constraint xor	*if* F *then* ¬K *and if* ¬F *then* K (<**R-xor**> between atomic features F and K that are derived from different parents);	
Constraint require	Feature K requires feature F, or shortly: K **requires** F	

FDs are now at the focus of researchers. As a result of continuous efforts to enhance expressiveness, there are some syntactic discrepancies and different interpretations of FDs semantics. All these should be taken into account when dealing with FD-related problems. With respect to the aims of our research, we need to extend FDs, too. Our aim is similar to Batory [35], who is the proponent of moving the FD notation closer towards domain ontologies. The need for extending FDs with contextualization is also motivated in [36].

In this paper, we are seeking to enrich FDs by domain ontologies and, on this basis, to provide some extensions. The motivation is the following observation: the structure and meaning of a concrete FD is dependable on the context and the latter on the goal a FD is pertaining. As we will show later by examples, by changing such attributes as <goal>, <context> we alter the shape and, perhaps, semantics of the FD (e.g., feature types). Having in mind requirements of the PL description, the context is inevitably changing in architectural design. This property further leads to treating attributes <goal> and <context> as generic categories meaning that each have some pre-specified concrete value taken from a prescribed space. In this paper, we use the term <generic context> only. It should be understood as a higher-level attribute (feature) having at least two different values. When representing the same initial concept, the use of generic context results in the construction of a set of the related FDs. The latter corresponds to the PL approach, in which the related groups of features model product families.

Table 1 summarizes the (syntax and semantics) of conventional attributes as well as innovative attributes of enriched FDs (EFDs) that are shown in bold.

4. Enriched Feature Diagrams (EFDs): Motivating research examples

To support the framework introduced in the previous section, we present two motivating research examples in this section. Let us consider the cosine calculation domain. This domain has the exceptional importance for many embedded and real time applications (e.g., FFT in DSP, image processing, etc. [30]). The goal is identified as "Design of Embedded SW to support product Line approach". Example 1 (Figure 1) explains the essence of the EFD use with respect to the introduced innovations. The term <context 1> has the meaning: "high performance computing".

Example 2 (Figure 2) explains what happens when the context is changed. The <context 2> has the meaning (value) now: "high performance mobile computing". As a result, new features appear (e.g., energy, C#) and richer ontology is introduced in the EFD. Furthermore, some features (e.g., C, Java) changed their type (from mandatory to optional because C# is more relevant than C and Java for mobile computing).

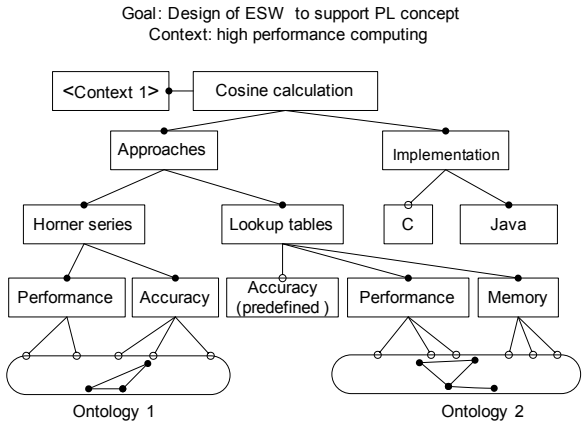


Figure 1. EFD with explicit context and ontology for cosine calculation

Attributes of EFDs are:

– The explicitly stated generic context

– A set of related EFDs

– The extended scope of variability

– Richer domain ontology

– An architectural description of a domain with ontology-based variability in mind.

The essential attribute of EFDs is domain ontology. In general, domain ontology can be expressed in a variety of ways depending on the domain, feature properties and design goal (context). Sometimes it is enough to specify relationships among features-leaves using the simplest constraint relationships such as 'feature A <requires> feature B' or 'A and B are mutual exclusive features' (see Table 1). If the features are of Boolean type more complex relationships based on the propositional logic can be used [35].

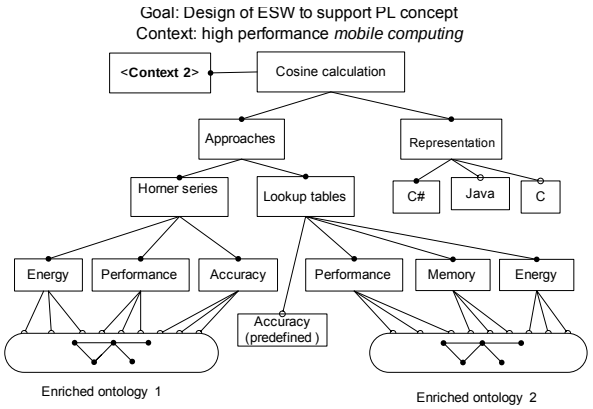


Figure 2. Changes in EFD due to context change

The atomic features (leaves on the EFDs) can be also related with some complex functional dependencies that can be specified using some analytic methods (if any exist), empiric (experimental) methods (if there is no other way to obtain functional dependencies as it is the case for energy consumption and performance)

or by using some prognostic methods based on probabilistic models, such as fuzzy logic [18].

In this paper, we used analytical methods where ontology can be expressed through a functional relationship (e.g., performance and accuracy can be expressed through the series length in the Horner scheme) and empiric methods in energy consumption evaluation (see a case study in Section 7).

5. Attributes of ontology-based generative components (OBGC)

In general, a generative component allows generating component instances on demand specified by meta-parameters values. An ontology-based generative component (OBGC) is the one which is built using the EFD (i.e., it is enriched by domain ontology) and implements the ontology and other features represented in the EFD using some generative technology. We make a distinction between terms 'generic' and 'generative' (we use the latter one when a generative technology is defined explicitly). In this paper, we use heterogeneous meta-programming as a generative technology [34]. Basic attributes of such a component are as follows.

1. We are treating the OBGCs as members of reuse repositories to support large-scale reuse and design knowledge sharing for embedded systems. EFDs are architectural models of OBGCs represented at a higher abstraction level.
2. An OBGC represents a family of generative components. A configuration of the family is specified by the design goal and design context. A member of the family is a generative component (either HW-oriented, e.g. given in a HW description language such as VHDL, SystemC, etc., or a pure SW component given in C, C# or other language).
3. FDs enriched by domain ontology (i.e., EFD) are a part of the specification document that serves as a high-level model of a generative component.
4. The rest part of the specification is a meta-program that encodes domain ontology and other relationships explicitly stated in the EFD.
5. Compliance between the high-level model (i.e., the EFD) and functionality of the meta-program is a very important attribute because of many aspects (e.g., better reuse, capabilities of transformation, and maintenance and evolution).
6. A full compliance may result in generating several domain program instances from the same meta-specification at a time.
7. The model of a generative component (in the case of the use of meta-programming) has two interrelated parts: meta-interface (for expressing communication with the environment and initialization of generative aspects) and meta-body (for expressing functionality and implementing generative aspects).
8. At the core of the meta-interface model is the meta-parameter concept. There are three categories of meta-parameters in the OBGC specification: 1) the highest-level meta-parameter(s) that correspond to the design context (goal); 2) the ontology-related meta-parameters; and 3) ordinary meta-parameters.
9. The highest-level meta-parameter(s) pre-specify the configuration of the family that can be implemented either as a set of separate modules or as a hierarchal branching of set of modules of a single specification.
10. The structure of an ontology-based meta-parameter has a name, abstract value and semantics. The latter one is expressed through explicitly described domain ontology. This requires decomposition, classification and ordering of domain ontology obtained as a result of analysis (e.g., analytic or experimental). The abstract value is a bridge for connecting knowledge represented in the meta-interface with the implementation knowledge that is hidden in the meta-body.

6. Encoding of EFDs using meta-programming

We consider encoding as model transformations that are not yet supported by automatic tools. Abstractly, model transformation is a process that transforms a source model (EFD, in our case) into a target model (meta-program, in our case). As a meta-program specification is a compound of two languages (meta and target in heterogeneous meta-programming), we need to use two-level model transformations. At level one, the given EFD is transformed into a meta-program model. Then, at level two, the latter model is transformed into meta-program itself. The meta-program model was described in Section 5 (although implicitly). For an explicit meta-program example, see Figure 4.

The model consists of meta-interface and meta-body. Meta-interface specifies meta-parameters, their values and constraints between some meta-parameter values (if any). For simplicity reasons, we describe level-one transformations, when a source model is given by a single EFD under the following conditions:

– FD is complete (with context, features, relationships, variant points, constraints and ontologies) and syntactically correct (in terms of introduced syntax).

– At least one variant for each variant point is identified.

– A scenario or scenarios written in the target language are given. How many of such scenarios are to be given is the matter of debates.

Transformation rules at level one are as follows.

A. Firstly, the context as a higher-level feature (variant point) is transformed into the highest-level meta-parameter(s).

B. Secondly, feature constraints are transformed into constraints that are expressed in terms of meta-parameters and their relationships using meta-constructs (e.g., meta-meta-if)

C. Thirdly, variant points that describe ontology-based features in the EFD are transformed into ontology related meta-parameters; then the rest variant points are transformed into meta-parameters.

D. Finally, meta-parameter values, which may express ontology at the meta-program level, are identified; then the rest values are identified.

By transformations mentioned in A, B, C and D we mean the rewriting of a graphical notation of EFD and changing them by a meta-program notation specified by the given meta-language.

Transformation rules at level two are as follows.

– Firstly, the first scenario of the target program instance is embedded into the meta-body; then places (locations) which relate to variants are identified and variability is implemented using meta-language constructs (e.g., meta-for, meta-case, meta-if, etc.)

– Secondly, the process of A is repeated for the rest scenarios.

– Thirdly, within the process the checking for completeness of encoding of the EFD is provided, as well as encoding correctness using tools that support meta-programming.

7. Case study: an extended research example

7.1. Definition, aim and methods used

Program efficiency (in terms of time or power consumption) can be improved by using 1) more efficient algorithms that solve the same computation problem, or 2) approximate computation algorithms that sacrifice accuracy for gain in other characteristics. More specifically, there are two methods for solving this problem: *data specialization* and *program specialization*.

Data specialization [37] aims at encoding results of early computations in data structures. The execution of a program is divided into two stages. First, a part of the algorithm is pre-computed in advance and the results are saved in a data structure such as look-up table (LUT). A LUT usually is an array (cache), which replaces a runtime computation with a simpler memory access operation. Of course, caching a computation is beneficial only if its execution cost exceeds the cost of a cache reference, i.e. it is recommended only for such performance-costly functions as *cosine, logarithm*, etc. The speed gain can be significant, since retrieving a value from the memory is faster than undergoing an expensive computation. We perform specialization of the given algorithm as follows. (1) We analyze the application source code to identify references to the computation costly functions. (2) We generate a LUT for the specialized function using the meta-programming techniques. (3) Then, all references to the function are replaced by the reference to its LUT. A more detailed description of the methodology can be found in [34].

Program specialization [38] aims at improving the efficiency of programs by exploiting known information about the input to a program, i.e. program specialization is the optimization of a program. An example of such specialization can be computation of the Taylor series specialized for its length.

7.2. Case study for cosine function

The most performance-costly part of many DSP algorithms (such as FFT, DCT, JPEG) is the calculation of the *cosine* function [39]. According to the *Amdahl's law*, the most effective way to improve performance of a program is to speed-up the most time-consuming part of it. If we speed-up the calculation of cosine values, we can achieve significant gains in program execution times and power usage. Such a fine-grained customization is very typical to embedded software development [40].

The cosine calculation has been chosen as a representative algorithm of the calculation-intensive application. We analyze three variants of the representative algorithm: Taylor series, cosine LUT and cosine LUT with linear interpolation.

Approximation of a function using simpler operations (e.g. addition and multiplication) as in the Taylor series of a cosine function (see Eq. (1)) can allow achieving higher performance and lower power consumption at a cost of accuracy.

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ... = 1 - \frac{x^2}{2}\left(1 - \frac{x^2}{12}\left(1 - \frac{x^2}{30}(...)\right)\right) \qquad (1)$$

Evaluation using the monomial form of an n-degree polynomial requires at most n additions and $\left(n^2 + n\right)/2$ multiplications, if powers are calculated by repeated multiplication and each monomial is evaluated individually. Using the Horner's scheme representation we need only n additions and n multiplications.

For even better performance, *data specialization* can be applied: known cosine values can be stored in a generated LUT. The trade-off here is that accuracy of the result may depend upon the size of the table. However, in many applications such as JPEG, the results of DCT are rounded-off to the integer values anyway. The complexity of the LUT based method is constant. It requires only 1 multiplication for the calculation of a LUT index and does not depend upon the size of a LUT.

In a simple LUT, the value of a function argument is rounded to the nearest value for which a function value in a LUT exists. Thus the accuracy of this ap-

proach is not fine. A more complex approach includes a LUT with linear interpolation of the function values for these arguments of a function, which are not available in the LUT. The complexity of the LUT with linear interpolation is also constant. It requires 2 multiplications and 4 additions, and does not depend upon the size of a LUT.

7.3. Results of experiments

Our investigation corresponds to that part of the general framework (see Section 3), which is described by obligatory features in feature diagrams (see Figures 1 and 2). The experiments were performed on a Compaq iPAQ H3900 (Pocket PC platform, Intel PXA250 400 MHz CPU, 32 MB RAM, Windows CE 3.0 OS).

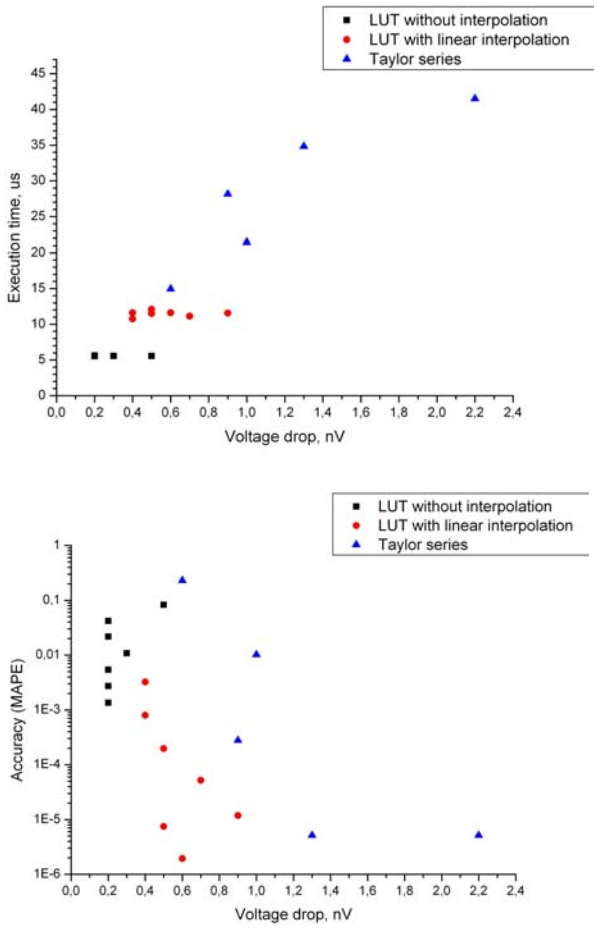


Figure 3. Trade-offs: power/execution time (top), and power/accuracy (bottom)

Our experiments show that the execution time and voltage drop for the Taylor series of *cos* function grows linearly with series length, whereas for the LUT-based approximation with and without interpolation the execution time and voltage drop values are flat. The LUT without interpolation has the lowest power consumption and the best performance. The LUT with linear interpolation has worse power consumption and performance, but higher accuracy. Taylor series have the worst results both in terms of power and performance (except for n=2 case, which however, has worst accuracy).

The trade-offs between power consumption (expressed via battery voltage drop), execution time and calculation accuracy (expressed via Mean Absolute Percentage Error - MAPE) parameters are shown in Figure 3.

7.4. Examples of meta-programs for implementing OBGC

Figure 4(a) describes the implementation of the only small part of EFD (see Figure 1), which is identified as “ontology 1”. The implementation consists of meta-interface (between symbols **$**) and meta-body (the rest part). The meta-interface has the human-oriented information (between symbols “) and the machine executable information (i.e., {2..6} n := 2;), where n is the name of the ontology-based meta-parameter. Its default abstract value (i.e., 2) also informs the user about performance-accuracy relationship in this case. The entire space of values specifies ontology 1.

Meta-body in the specification is implemented using Open PROMOL functions (**@sub**, **@for** and **@rep** given in bold in Figure 4), as meta-language [23]. The result of execution, when n = 3, is the program in C (Figure 4, b).

Figure 5 describes a more complex implementation, in which some results from experiments we have carried out for energy as ontology for transferring the knowledge to the user are included. The only meta-interface of the generative component is given here. This example is also illustrative because some important characteristics (e. g., argument parameter whose values are influential to accuracy; operating system, mode, and type of processors, which are important for energy measurements) are missed in this specification for simplicity reasons. Note that square brackets (see Figure 5) specify feature constraints under which meta-parameter values are assigned.

```
$
"Enter the number of terms in Taylor series of cosine function:
       2 terms:  4 mult/div ops, 2 add/sub ops, 10E-2 accuracy
       3 terms:  6 mult/div ops, 3 add/sub ops, 10E-3 accuracy
       4 terms:  8 mult/div ops, 4 add/sub ops, 10E-4 accuracy
       5 terms: 10 mult/div ops, 5 add/sub ops, 10E-6 accuracy
       6 terms: 12 mult/div ops, 6 add/sub ops, 10E-8 accuracy" {2..6} n:=2;
$

double cos_@sub[n](double x) {
       double x2=x*x;
       return 1-x2/2@for[i,2,n,{*(1-x2/@sub[2*i*[2*i-1]]}]@rep[n-1,{)}];
}
```

(a)

```
double cos_4(double x) {
       double x2=x*x;
       return 1-x2/2*(1-x2/12*(1-x2/30*(1-x2/56)));
}
```

(b)

Figure 4. Generative cosine meta-program (a) and its generated instance (b) when n=4

```
$
"Select the context" {energy, performance, accuracy} context:=energy;
"Select the implementation language" {C++, C#, Java) lang:=C++;
"Select the algorithm type:
       1 - Taylor series
       2 - Look-up table
       3 - Look-up table with linear interpolation"   {1,2,3} type:=3;
[type=1 and [[context eq {accuracy}] or [context eq {performance}]]]
       "Enter the length of the Taylor series
       2 terms:  4 mult/div ops, 2 add/sub ops, 10E-2 accuracy
       3 terms:  6 mult/div ops, 3 add/sub ops, 10E-3 accuracy
       4 terms:  8 mult/div ops, 4 add/sub ops, 10E-4 accuracy
       5 terms: 10 mult/div ops, 5 add/sub ops, 10E-6 accuracy
       6 terms: 12 mult/div ops, 6 add/sub ops, 10E-8 accuracy" {2..6} n:=3;
[type=1 and context eq {energy}]
       "Enter the length of the Taylor series
       2 terms: voltage drop 0.6 nV
       3 terms: voltage drop 1.0 nV
       4 terms: voltage drop 0.9 nV
       5 terms: voltage drop 1.3 nV
       6 terms: voltage drop 2.2 nV" {2..6} n:=3;
[type=1] "Enter the size of the look-up table (time: 5.6 us, voltage drop 0.28 nV,
       accuracy: 8 - 1E-2; 16, 32 - 1E-3; 64, 128 - 1E-4; 256 - 1E-5; 512 - 1E-6)"
       {8,16,32,64,128,256,512} size:=32;
[type=2] "Enter the size of the look-up table (time: 11.5 uS, voltage drop 0.56nV,
       accuracy: 8 - 1E-2; 16, 32 - 1E-3; 64, 128 - 1E-4; 256 - 1E-5; 512 - 1E-6)"
       {8,16,32,64,128,256,512} size:=32;
$
```

Figure 5. Meta-interface containing domain knowledge and context information

8. Discussion, evaluation and conclusions

The higher complexity of architectural design (in terms of features), the greater need to model and implement the domain variability in a systematic way is. However, architectural design may be dependent on the context which is influential to the variability leading to the formation of more complex relationships among features (i.e., domain ontology). Although feature diagrams provide a mechanism enabling to model and manage the complexity, their expressiveness is not enough in this case.

What we suggest in this paper is: 1) to enrich feature diagrams by context changes and repurposing (i.e., by ontology) and then to represent the domain variability model explicitly; 2) to encode enriched feature diagrams using heterogeneous meta-programming techniques, thus resulting in creating of generative components for embedded software domain.

Our research to support the introduced methodology is based on specialization of data, algorithms and programs of various embedded software tasks (e.g., FFT, sparse matrix multiplication, triple redundancy solutions, etc.) with enhanced requirements in mind (e.g., energy consumption estimates). As majority of computation aggressive tasks (e.g., FFT) are based on cosine calculations, this function has been chosen as the most representative one in this paper.

With the energy estimates at the application level in mind, a discovery of domain ontology requires a thorough experimentation followed by analysis of the results in order to obtain various relationships among features (e.g. energy, performance, accuracy, argument values, and memory). As we provide experiments automatically, a huge space of relationships can be

identified. Therefore, it is possible to transfer the only part of knowledge (essential) (obtained during experiments) to the highest level of meta-program (meta-interface) in order the user is being informed to select the solution relevant to his context (during generation). The document of an ontology-based generative component, which is assumed to be a member of external repositories, along with feature diagrams can also be supplemented with more thorough descriptions of experiments (e.g., graphics supplied by conditions of experiments).

References

[1] **P. Clements, L. Northrop.** Software Product Lines: Practices and Patterns. *Boston: Addison-Wesley*, 2002.

[2] **J. MacGregor.** Requirements Engineering in Industrial Product Lines. *Proc. of Int. Workshop on Requirements Engineering for Product Lines REPL'*02, *Essen, Germany*, 2002, 5–11.

[3] **K.C. Kang, K. Lee, J. Lee, S. Kim.** Feature-Oriented Product Line Software Engineering: Principles and Guidelines. *In K. Itoh, S. Kumagai, T. Hirota (Eds.), Domain Oriented Systems Development - Practices and Perspectives, Taylor & Francis*, 2003.

[4] **K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh.** FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering, Vol*.5, 1998, 143–168.

[5] **D.M. Weiss, C.T.R. Lai.** Software Product-Line Engineering: *A Family-Based Software Development Approach. Addison-Wesley, Reading, MA, USA*, 1999.

[6] **J. Coplien, D. Hoffman, D. Weiss.** Commonality and Variability in Software Engineering. *IEEE Software* 15(6), 1998,37–45.

[7] **M. Lindwer, D. Marculescu, T. Basten, R. Zimmennann, R. Marculescu, S. Jung, E. Cantatore.** Ambient intelligence visions and achievements: linking abstract ideas to real-world concepts. *Proc. of Design, Automation and Test in Europe Conference and Exhibition, DATE* 2003, 3–7 *March* 2003, *Munich, Germany,* 10–15.

[8] **R. McGreal (ed.).** Online Education Using Learning Objects. *Routledge*, 2004.

[9] **K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson.** Feature-Oriented Domain Analysis (FODA) Feasibility Study. *TR CMU/SEI*-90-*TR*-21, *Software Engineering Institute, Carnegie Mellon University, November* 1990.

[10] **U.W. Eisenecker, K. Czarnecki.** Generative Programming: Methods, Tools, and Applications. *Addison-Wesley*, 2000.

[11] **K.C. Kang, J. Lee, P. Donohoe.** Feature-Oriented Product Line Engineering. IEEE Software, 19(4), 2002, 58–65.

[12] **P.-Y. Schobbens, P. Heymans, J.-Ch. Trigaux, Y. Bontemps.** Feature Diagrams: A Survey and a Formal Semantics. *Proc. of* 14*th IEEE Int. Requirements Engineering Conference* (*RE*'06), 11–15 *September* 2006, *Minneapolis/St.Paul, Minnesota, USA*, 136–145.

[13] **B. Bailey, G. Martin, T. Anderson (eds.).** Taxonomies for the Development and Verification of Digital Systems. *Springer*, 2005.

[14] **K. Czarnecki, C.H.P. Kim, K.T. Kalleberg.** Feature models are views on ontologies. *Proc. of* 10*th Int. Software Product Line Conference* (*SPLC* 2006), 21–24 *August* 2006, *Baltimore, USA*, 41–51.

[15] **V. Štuikys, R. Damaševičius, I. Brauklytė, V. Limanauskienė.** Exploration of Learning Object Ontologies Using Feature Diagrams. *Proc. of World Conference on Educational Multimedia, Hypermedia & Telecommunications* (*ED-MEDIA* 08), *June* 30–*July* 4, 2008, *Vienna, Austria*, 2144–2154.

[16] **G. Guizzardi.** On Ontology, ontologies, Conceptualizations, Modeling Languages, and (Meta)Models. I*n O. Vasilecas, J. Eder, A. Caplinskas (Eds.), Frontiers in Artificial Intelligence and Applications, Databases and Information Systems* IV. *IOS Press, Amsterdam*, 2007, 18–39.

[17] **S.-B. Lee, J.-W. Kim, C.-Y. Song, D.-K. Baik.** An Approach to Analyzing Commonality and Variability of Features using Ontology in a Software Product Line Engineering. *Proc. of* 5*th ACIS Int. Conf. on Software Engineering Research, Management & Applications*, *SERA* 2007, 20-22 *Aug*. 2007, 727–734.

[18] **S. Robak, A. Pieczynski.** Employing fuzzy logic in feature diagrams to model variability in software product-lines. *IEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems*, 2003, *Huntsville, USA*, 305–311.

[19] **P.Y.H. Wong.** An Investigation in Energy Consumption Analyses and Application-Level Prediction Techniques. *MSc. Theses, Univ. of Warwick, UK*, 2006.

[20] **Ch. Krintz, Y. Wen, R.Wolski.** Application-level prediction of battery dissipation. *Proc. of Int. Symp. on Low Power Electronics and Design ISLPED*'04, *New York, NY, USA*, 2004, 224–229.

[21] **N. Ravi, J. Scott, L. Iftode.** Context-aware Battery Management for Mobile Phones. *Proc. of* 6*th Annual IEEE Int. Conf. on Pervasive Computing and Communications* (*PerCom* 2008), 17–21 *March, Hong Kong*, 2008, 224–233.

[22] **A. Sagahyroon.** Power Consumption in Handheld Computers. I*EEE Asia Pacific Conf. on Circuits and Systems, APCCAS* 2006, *Singapore*, 4–7 *December* 2006, 1721-1724.

[23] **T. Simunic, G. de Micheli, L. Benini, M.Hans.** Source code optimization and profiling of energy consumption in embedded systems. *Proc. of* 13*th Int. Symp. on System Synthesis* (ISSS'00), *Washington, DC, USA*, 2000, 193–198.

[24] **J. Flinn, M. Satyanarayanan.** Energy-aware adaptation for mobile applications. *Proc. of the* 17*th ACM Symposium on Operating Systems Principles (SOSP),* 1999, *Charleston, South Carolina, USA*, 48–63.

[25] **A. Peymandoust, T. Simunic, G. de Micheli.** Low power embedded software optimization using symbolic algebra. *Proc. of Design, Automation and Test in Europe Conference and Exhibition, DATE*'02, 4–8 *March* 2002, *Paris, France*, 1052–1057.

[26] **E.-Y. Chung, L. Benini, G. De Micheli.** Source code transformation based on software cost analysis. *Proc. of* 14*th Int. Symp. on Systems Synthesis ISSS*'01, *New York, NY, USA*, 2001, 153–158.

[27] **M. Kandemir, N. Vijaykrishnan, M. Irwin, W. Ye.** Influence of Compiler Optimizations on System Power. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 9(6), 2001, 801–804.

[28] **H. Mehta, R.M. Owens, M.J. Irvin, R. Chen, D. Ghosh.** Techniques for Low Energy Software. *Proc. of Int. Symp. on Low Power Electronics and Design, August* 18-20, *Monterey, CA, USA*, 1997, 72–75.

[29] **L. Benini, G. de Micheli.** System-level power optimization: techniques and tools. *ACM Trans. Des. Autom. Electron. Syst*., 5(2), 2000, 115–192.

[30] **R. Kirner, M. Grossing, P. Puschner.** Comparing WCET and Resource Demands of Trigonometric Functions Implemented as Iterative Calculations vs. Table-Lookup. *In F. Mueller (Ed.),* 6*th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July* 4, 2006, *Dresden, Germany. Dagstuhl Seminar Proceedings* 06902, *Schloss Dagstuhl, Germany*, 2006.

[31] **G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin.** Aspect-Oriented Programming. *Proc. of* 11*th European Conference on Object-Oriented Programming, ECOOP*'97, *Jyväskylä, Finland, June* 9–13, 1997. *Springer LNCS Vol.*1241, 220–242.

[32] **Y.C. Cheong, S. Jarzabek.** Frame-Based Method for Customizing Generic Software Architectures. *Proc. of the Fifth Symp. on Software Reusability, SSR* 1999, *May* 21-23, 1999, *Los Angeles, CA, USA*, 103–112.

[33] **T. Sheard.** Accomplishments and Research Challenges in Meta-Programming. *Proc. of* 2*nd Int. Workshop on Semantics, Application, and Implementation of Program Generation* (*SAIG*'2001), *Florence, Italy*, 2001. *Springer LNCS, Vol*. 2196, 2–44.

[34] **V. Štuikys, R. Damaševičius.** Metaprogramming Techniques for Designing Embedded Components for Ambient Intelligence. *In T. Basten, M. Geilen, H. de Groot (eds.), Ambient Intelligence: Impact on Embedded System Design. Kluwer Academic Publishers, Boston, November* 2003, 229–250.

[35] **D.S. Batory.** Feature Models, Grammars, and Propositional Formulas. *In J.H. Obbink, K. Pohl (Eds.), Proc. of 9th Int. Conf. on Software Product Lines, SPLC* 2005, *Rennes, France, September* 26–29, 2005. *Springer LNCS, Vol*. 3714, 7–20.

[36] **R. Damaševičius, V. Štuikys, E. Toldinas.** Domain Ontology-Based Generative Component Design Using Feature Diagrams and Meta-Programming Techniques. *In R. Morrison, D. Balasubramaniam, K. Falkner (Eds.), Proc. of* 2*nd European Conf. on Software Architecture ECSA* 2008, *Paphos, Cyprus. LNCS* 5292, *Springer-Verlag*, 2008, 338-341.

[37] **T. Knoblock, E. Ruf.** Data Specialization. *ACM SIGPLAN Notices* 31(5), 1998, 215–225.

[38] **N.D. Jones, C.K. Gomard, P. Sestoft.** Partial Evaluation and Automatic Program Generation. *Prentice Hall International*, 1993.

[39] **L. Nyland, M. Snyder.** Fast Trigonometric Functions Using INTEL's SSE2 Instructions. *Intel Tech. Rep., available online at: http://www.weblearn.hs-bremen.de/risse/RST/docs/Intel/03-041.pdf.*

[40] **D. Beuche, O. Spinczyk, W. Schroeder-Preikschat.** Fine-grain Application Specific Customization for Embedded Software. *Proc. of Design and Analysis of Distributed Embedded Systems, DIPES* 2002, *August* 25–29, 2002, *Montréal, Canada*, 141–151.

Received August 2008.

DOI: 10.5755/j01.itc.37.4.11986