High-Level Models for Transformation-Oriented Design of Hardware and Embedded Systems

Robertas DAMASEVICIUS, Vytautas STUIKYS
Kaunas University of Technology, Lithuania
robertas.damasevicius@ktu.lt

***Abstract*—Evolution of design methodologies follows a common trail: technology scaling leads to growing design complexity and rising abstraction level in the domain. Introduction of new (higher) abstraction levels emphasizes the importance of reuse and transformations. The design process can be seen as a sequence of high-level transformations from the higher-level specification to the lower-level one. We analyze high-level modeling and metaprogramming techniques for supporting transformations based on domain variability models. Next, we present a reuse evolution model for domain component design at a higher abstraction level to support the transformation-oriented approaches. Finally, high-level modeling techniques (UML-domain language metamodels for domain code generation, parameterized UML classes for template metaprogramming, feature models for explicit representation of variability) for specification of transformations and metaprograms are analyzed.**

***Index Terms*—design automation, transformation, hardware and embedded system design**

I. INTRODUCTION

In recent years, there have been two significant trends in hardware (HW) and embedded system (ES) design domains. The first trend is the rapid growth of complexity in the design of extremely complex devices, such as System-on-Chip (SoC). The second trend is the adoption of UML to support design of real-time and embedded systems [1]. This progress is partially due to the continuous efforts of the electronics design automation (EDA) community to improve design methodologies based on higher-level abstractions and reuse [2]. Here abstraction is understood as a description of the design problem at some level of generalization that allows concentrating on the key aspects of the problem without getting into details.

Moving towards high-level abstractions is a long-standing engineering tradition in HW and ES design. In the past, a higher-level abstraction was usually introduced every 7-10 years, but now it happens even faster. The introduction of new abstraction levels raises the stake of various design methodologies based on reuse and transformations, such as platform-based design [3], because the ultimate goal of using a higher-level abstraction is to produce a lower-level design specification automatically. Design abstractions are of two categories: either design problem related (e.g., logic gates, platforms, variability models, etc.) or solution domain related (e.g., high-level specification languages, hardware description languages (HDLs), metamodels, etc.).

Abstractly, the design process requires that solution domain abstractions are relevantly and correctly mapped to design domain abstractions. The mapping is to be performed by satisfying some constrains and requirements for a given application. In this context, the design process can be seen as a high-level transformation for transforming the higher-level specification into the lower-level one.

The use of higher-level abstractions means at least two things for a designer: 1) he/she can deal with the design more abstractly, thus eliminating much of unnecessary details and gaining design efficiency; 2) he/she can express design variability, thus enabling the creation of generic solutions for reuse. In a broader sense, the introduction of higher-level abstractions also raises other issues, such as unambiguous understanding of categories and terms used in the Electronic Design Automation (EDA) community [4] and dealing with the levels of abstractions and their dependencies in a unified manner [5].

The paper presents an analysis of high-level models for implementation of transformational approaches based on higher-level abstractions in HW and ES design domains. We motivate the importance of our research by the following reasons: 1) The necessity of considering design at a higher abstraction level caused by ever increasing complexity of systems to be designed [6]; 2) The efforts of introducing high-level languages such as UML and SystemC in the HW and ES design practice [7]; 3) The efforts to exploit the reuse potential in design methodologies as widely as possible in both dimensions (component-based and generative/transformational reuse) [2, 8]. 4) Introduction of novel design paradigms, such as platform-based design [3], ambient intelligence [9], product lines [10]; 5) The increasing role of configurable components in design [8] (configuration is a way for expressing variability at a higher abstraction level).

The paper is organized as follows. Section 2 analyzes the related works. Section 3 presents analyzes domain variability and reuse models for well-understood design domains. Section 4 analyzes the capabilities of metaprogramming techniques within solution domain. Section 5 describes various high level models for specification of transformations and metaprogramming in the domain. Finally, Sections 6 discusses the results and presents conclusions.

II. RELATED WORKS

HW and ES modeling at a higher level of abstraction is a hot topic [11-14], which covers UML-based modeling [15, 16], design patterns [17], meta-models and meta-modeling

[18], model and metamodel taxonomies [19, 20]. A shift towards high-level modeling of HW and ES represents a decisive turn of the EDA community towards a meta-dimension of design [21]. Models are used to define the syntax, semantics and composition of domain components using object-oriented or architecture-based abstractions [19], extend the syntax of HDLs by providing a meta-level above the user modeling level [22], define the relationship between the abstract syntax and semantics domains [17], represent the evolution and co-evolution of design artifacts [18], describe the component reuse process [23], capture and manage the system requirements [24], allow for formal analysis, verification, and ES validation at the design time [25], define platforms for creating executable applications from system-level specifications [15], define complex design steps such as refactorings or introduce complex communication patterns between components [26].

Modeling is required to raise the level of abstraction above the HDL level. As such, modeling can be used for the high-level specification and verification of HW and ES. However, the seamless integration of modeling into domain design flows and the need for increased design productivity requires a higher degree of domain automation, which can not be achieved without defining mappings from user models to other representations (e.g., other models, code, storage formats, etc.). Such mappings are metamodels that connect the problem domains and the solution domains, or product domains and process domains [26, 27]. Metamodels are also used for tool integration [25, 28]; specification and generation of domain-specific modeling languages [29]; and for specifying various aspects of the developed system [30].

The primary motivation for introducing metamodels is to reuse models based on concepts defined in a metamodel and to automate the production of design systems using transformational and generative techniques. Metamodeling defines transformations using the elements of the metamodel, and offers algorithms to apply these transformations using generative techniques. Metamodel-based transformations permit descriptions of mappings between models created using different concepts from possibly overlapping domains. Transformation process facilitates reuse of models specified in one domain-specific modeling language in another context, such as using another modeling language [31]. Various techniques can be used for meta-modeling, but usually meta-modeling is achieved using a subset of UML and Model-Driven Architecture (MDA) methodology as a theoretical background [32].

Program and model transformation is a very broad research area with a long history [33, 34, 35]. Transformations can be classified into two main categories: program source code transformations (not considered in this paper) and model transformations [35]. Transformations commonly used by designers can be defined as transformation patterns [36] or generic transformations [37] that capture intelligent and well-proven design techniques. Generic transformations represent a link between source-level program transformations and model transformations. Model transformations have the direct relation with the UML and MDA concept [32]. It is also recognized and accepted in HW and ES domains [4].

Conceptually, model transformation is similar to program transformation, but is applied to models instead of programs. It provides capability for describing relationships and mappings between model concepts (i.e. metamodel elements) and sets of models, where a mapping defines a correspondence between elements of the source and target model. Model transformations can be classified as endogenous and exogenous [38]. Endogenous transformations are between models expressed in the same language, e.g., code generation, reverse engineering, migration. Exogenous transformations are between models expressed using different languages, e.g., optimization, refactoring, simplification. Also a distinction can be made between horizontal transformations, where the source and target models reside at the same abstraction level, and vertical transformation, where the source and target models reside at different abstraction levels.

The domain of model transformations is closely related to the modeling domain. Transformations itself can be modeled using models and metamodels [39]. Transformations can be treated as captured executable designs and architectural patterns or metamodels that can be applied in many contexts to solve a narrow specific design task [40]. Specifying the transformations as metamodels allows development of tools that support the controlled evolution of models [41]. Furthermore, transformations also can be used to support variability within system product lines at the model level [42].

Summarizing, the role of transformation processes in design of complex electronic systems is increasing due to a common trend to raise the abstraction level in HW/ES design by introducing high-level specification languages (such as UML), which underscores the need of design automation in the domain.

III. VARIABILITY AND REUSE MODELS IN WELL-UNDERSTOOD DESIGN DOMAINS

A. Context of research

We assume that the design is specified at a higher abstraction level using higher-level abstractions such as UML and meta-languages and a lower-level representation of the design using HDLs such as VHDL, Verilog or SystemC [7]. Furthermore, high-level design is not only directed to produce a specific solution (lower-level representation) from a given high-level specification, but rather it is oriented to describe a family of related solutions, i.e., the design is variability-oriented.

We define the design process at a higher abstraction level as a set of interrelated transformation tasks. A transformation task is a transformation from a design problem specification described at the higher abstraction level to the lower abstraction level. In our case, higher-level abstractions are metamodels, UML, metalanguage and metaprogramming techniques [43]. Lower-level abstractions are system/component specifications in a HDL. Transformations usually are performed automatically, but for some kind of higher-level abstractions, such as meta-metamodels, can be performed manually.

We define transformation in well-understood design domains (WUDD). A WUDD is (1) a sub-domain of a larger design domain; (2) it has well-defined and well-proven

models; (3) it is narrow enough but has multiple applications; and (4) it may require an analysis of different characteristics (aspects, models, etc.) for different applications, thus the commonalities, specificity, and variability can be easily captured using ad hoc (the usual design practice) or systematic domain analysis methods such as those overviewed in [44].

The primary motivations for introducing a concept of WUDD are: (1) to achieve a higher extent of reuse; (2) to facilitate the implementation of transformation/generative reuse; (3) to achieve some unification in the provided study and experiments. Note that it is not necessary to analyze the domains only while considering application of UML (model-based) and metaprogramming technologies because these technologies are domain independent. But on the other hand, to implement metaprogramming is much easier for the WUDDs.

HW and ES domains contain a wide variety of WUDDs. Examples are: soft IP computational models [46], communication-based design [46], interface-based design [47], reliable design based on redundancy models [48], and algorithms for embedded software [49]. After we have selected a particular WUDD, the next step is to define a variability model for this domain.

B. Variability model for coarse-grained soft IPs

The variability model is a key point for implementing transformations. A WUDD is described as a set of features or aspects which are to be expressed through parameters. Parameters may be different for some aspects, thus the parameters may have different values. The aspects belong to basic categories, expressing commonalities, specificity, variability, functional, structural or other aspects. As the variability aspects are most important, the dependencies between different variability aspects are taken into account, too. Dependencies within variability are the most crucial part of the model because they may require in-depth analysis or may depend upon the context of an application domain. Furthermore, in some cases dependencies can be very complex, thus requiring the vast amount of modeling efforts.

An example of such variability model is presented in Table I (taken from [50]). The model consists of variability and commonality only. It is assumed that the variability dependencies, i.e. dependencies between different parameter values, are a separate issue. A similar model for the configurable processor is outlined in [8] without explicit representation of parameters for alternatives.

Implementation of the variability model can be carried out differently. For example, one alternative is to develop a set of reusable components, where each component represents some aspects of the model. The other alternative is the integration of all anticipated aspects of the model into a generic specification, thus developing a generic component as a specific kind of the transformation system (or generator). Such a system then can be used for generating an instance for the concrete application when it is actually to be designed. In general, the development and implementation of the generator and the use of the generator can be viewed as a twin life cycle model [51], the higher-level model which combines design for reuse (the development of generators) and design with reuse (the use of generators). Actually this concept is implemented in design repositories already exploited in the EDA community for soft IP (Intellectual Property) exchange [52].

TABLE I. VARIABILITY MODEL FOR DSP MICROPROCESSOR DESIGN (ACCORDING TO [50])

Parameter	Range	Default	Description
Data word	8-64 bits	16 bits	Data word length
Data address	8-23 bits	16 bits	< = data word (dependency)
Program word	32-…	32 bits	Commonality of the model
Program address	8-19 bits	16 bits	<= data word (dependency)
Multiplier width	8-64 bits	Data word	Word length of multiplier operands (dependency)
MUL guard bits	0- 16 bits	8 bits	<= data word – 2 (dependency)
MAC type	0-…	0 (basic unit)	Commonality of the model
Shifter type	0-…	0 (basic unit)	Commonality of the model
ALU type	0-…	0 (basic unit)	Commonality of the model
Index registers	8 or 16	8	Number of index registers
Accumulators	2, 3 or 4	4	Number of accumulators
Enable C & D	0 or 1	0 (not enable)	Use of C and D registers as ALU operands
Modifier only	0 or 1	0 (not enable)	Only odd-numbered index registers can be modifiers
Loop registers	0-8	0 (no loop HW)	Number of HW looping units
Address modes	0-3	0	Supported data-addressing modes

C. Reuse evolution model

At the core of the transformation-oriented approach to system design are two models: reuse evolution model (REM) and variability model. While the variability model is applied only when design entities are modified (e.g., for the product line design), the REM is applied throughout the entire life cycle of the design entity. The REM is seen as a framework for understanding of the soft IP-based system design and evolution processes based on the design for reuse and design for change paradigms.

The primary concept of the REM is simple: the systems (components) must evolve from the already existing ones rather than being developed from scratch. The evolution is achieved by change (customization, adaptation, etc.) of the existing systems, transformation of the existing high-level models and low-level code artefacts, reuse of models, components (soft IPs) and subsystems. Although the introduced model does not restrict the kind of changes, only minor changes (e. g., adding a minor glue code to the coarse-grained soft IP through an external composition [53])

are usually allowed. Therefore, grey-box reuse is the central point in the proposed model.

The idea of the REM is borrowed from the Spiral model [54] well-known in software engineering, which is used to describe a software development process iteratively. The evolution of reuse models, as the REM suggests, is a continuous process that has three phases: search/analysis, understanding and adaptation/ modification (Fig. 1).

1) At the search/analysis phase, the component candidates (instances), which are considered as black-box soft IPs, are searched and selected for usage.

2) At the understanding phase, the components are analyzed as glass-box entities. Their functionality, architecture and potential for reuse and modification is understood.

3) At the adaptation/modification phase, a designer performs adaptation (extension of functionality), as well as modifications (introduction of variations) based on grey-box reuse. The modified component is used in the currently designed system or can be used again as the black-box entity in the other context of an application, and thus the evolution process repeats itself in a spiral-like cycle.

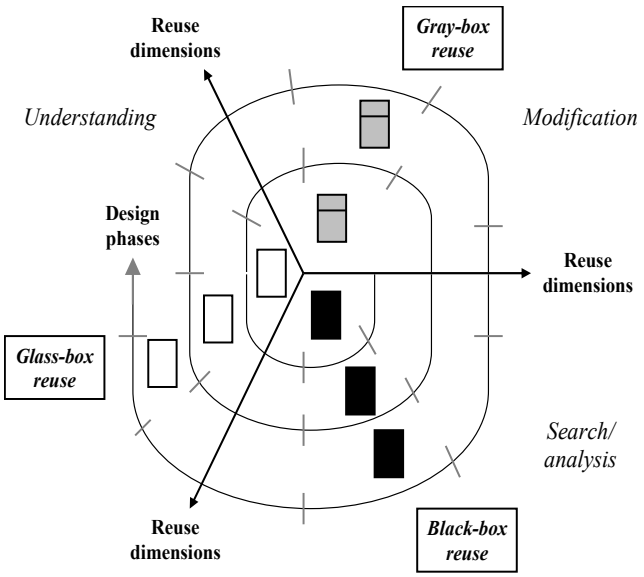

Figure 1. Reuse Evolution Model: Designer's Viewpoint.

However, the REM differs from the Spiral model significantly as follows:

1) REM explicitly focuses on the reuse process in design. It describes three different stages of reuse, reuse models used and the particular actions of the designer.

2) It combines the ideas of design-for-reuse and design-for-change. Design-for-reuse promotes the development of reusable components. Design-for-change promotes the development of components while foreseeing and anticipating future modifications and changes. The combination of two paradigms requires the development of modifiable and changeable components using open transformation environments and catalogues of design transformations in the same way as the design patterns are catalogued.

3) The process of evolution is continuous, uninterrupted and possibly everlasting. No component is being ever finished, finally released or disposed after the end of its life cycle. The usage of a particular design entity may be discontinued in a certain context of an application, but it is preserved for possible future uses and modifications in other contexts of application.

IV. METAPROGRAMMING METHODOLOGY

Metaprogramming is a methodology which allows manipulations with programs as data [55]. The methodology allows representing generic specifications for reuse. The generic specification represents domain variability either implicitly or explicitly. Once the domain is expressed as a metaprogram, the metaprogram and its processing tool is a vehicle for generating the domain program instances.

There are two kinds of metaprogramming: homogeneous and heterogeneous ones. Homogeneous metaprogramming deals with program transformations within the endogeneous environments. Heterogeneous metaprogramming deals with program transformations within the exogeneous environments (usually two).

Homogeneous metaprogramming depends on the capabilities of the given domain language. The key point is that the domain language should provide a support for realization of the metaprogramming concept. More precisely, such a language should allow the decomposition of its constructs into two levels as follows. The first level represents the lower-level constructs for expressing basic domain functionality. The second level represents higher (meta)-level constructs for expressing variability and generalization. Homogeneous metaprogramming is typically used for describing generic domain components. Two standard hardware languages (VHDL and SystemC) support homogeneous metaprogramming: VHDL has Generic statement and SystemC has class template.

Heterogeneous metaprogramming depends on the capabilities of a metalanguage, which has metaconstructs (such as meta-if, meta-for) and provides support for implementing the metaprogramming techniques. The main aim is to create a metaprogram – a program generator for a narrow domain of application. Conceptually, a metaprogram is based on the domain variability model [56]. Structurally, a metaprogram consists of a generic interface, domain code instances and a modification algorithm that describes generation of a particular domain program depending upon values of the generic parameters (Fig. 2).

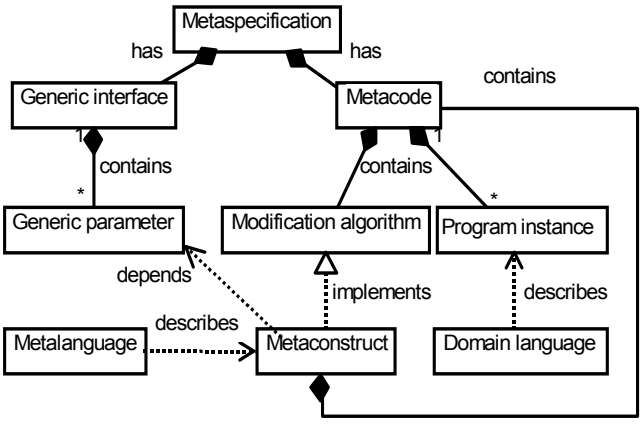

Figure 2. Structure of a Metaprogram.

Heterogeneous metaprogramming uses two different languages in the same metaprogram. The lower-level language (domain language) is used for expressing the basic domain functionality. The higher-level language (metalanguage) is used for expressing generalization and describing domain program modifications. A general-purpose programming language (such as C++, Java) can be used as a metalanguage for implementing component generators. An alternative is to use a dedicated metalanguage (such as Open PROMOL [57]), which is dedicated to generic component parameterization and component instance generation. Though there are no essential differences between both methods, the second has some advantages from the user's perspective. A dedicated metalanguage processor can better ensure the explicit separation of concerns when implementing external parameterization, thus giving some advantages for a user.

The capabilities of homogeneous and heterogeneous metaprogramming are compared in Table II. The particular advantage of heterogeneous metaprogramming is its suitability for developing domain code generators. For more information concerning heterogeneous metaprogramming, see [44, 53, 56, 57, 58].

TABLE II. HOMOGENEOUS METAPROGRAMMING VS. HETEROGENEOUS METAPROGRAMMING

Criteria	**Homogeneous metaprogramming**	**Heterogeneous metaprogramming**
Need of an external metalanguage	No	Yes
Kind of languages	Domain (target) only	Meta and domain (target)
Type of metalanguage	Domain language subset	Metalanguage or programming language as a metalanguage
Dependency upon target language	Dependent	Independent
Separation of instances within generic specification	Implicit within compiler (user inaccessible)	Explicit (user visible& accessible) via external processor
Scope of components	Fine-grained components	From fine- to coarse-grained
Scope of variability expression	Restricted	From grey- to white-box reuse
Adaptation to hardware synthesis limitations	Limited adaptation	No limitations
Verification problem	Moderate	Difficult

V. HIGH-LEVEL MODELS FOR SPECIFICATION OF TRANSFORMATION AND METAPROGRAMMING

A. Concept of model and metamodel

Models are abstract views of a system that describe the structure and behavior of the system and the relationship between the parts of the system. Models underline the features of a system that are important to the designer, while other (unimportant) features are not represented. Models are implemented using a domain language (e.g., a HDL). Such implementations are specific domain systems performing a particular domain task. What lies behind a model is the relationship between modeling language constructs and domain language constructs. Such relationship can be defined in a metamodel that represents an abstract view on system description notations and describes a mapping between particular elements of these notations. Metamodel relates to the model of a system, as the model of the system relates to the system itself. Therefore, the model-based design allows to build a hierarchy of models (metamodels, meta-metamodels, etc.) which allows to model domain systems at multiple levels of abstraction and detail.

The process of model-based design of a system from the abstract representation to the low level implementation involves the usage of two languages: 1) model-based design languages (such as UML) are used explicitly to describe a model of the system, while 2) domain language (such as VHDL, SystemC) is used implicitly when a system (or parts thereof) are generated from the model. Generally, this means that model-based design requires two metamodels: 1) a metamodel of a model-based design language (e.g., MOF [59] for UML) and 2) a metamodel of a relationship between abstractions of a model-based design language and a domain language (e.g., UML-VHDL metamodel).

B. Metamodel as a bridge between models and metaprograms

The practical application of metamodels is the description of transformation from high-level system specification in a modeling language to a specific implementation in a HDL. An example of such metamodel, describing generation of VHDL component from UML state diagrams is given in Fig. 3. UML state diagrams are composed of states and transitions. UML states are always mapped to a case statement within a VHDL process. A special internal signal “state” is used to activate the process. Then depending upon a value of the signal “state”, the particular actions are selected and executed. UML events correspond to VHDL signals. UML actions are represented in VHDL as sequential statements selected using the case statement and are executed within the process.

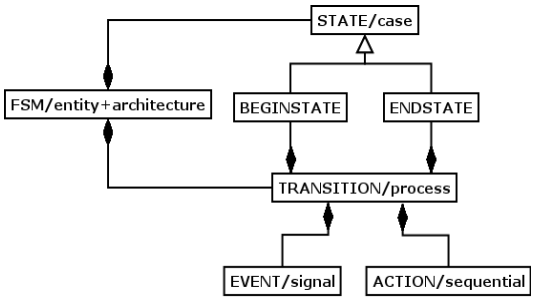

Figure 3. A Relationship Metamodel Between UML State Diagram and VHDL Abstractions.

Such transformation between UML and VHDL van be performed manually or can be automated using the heterogeneous metaprogramming techniques. In the latter case, the metamodel serves as a guideline for developing a metaprogram. For example, a metamodel shown in Fig. 2 was used to develop a VHDL code generator [27] in a PragScript scripting language used by the UMLStudio package [60]. PragScript is a LISP-like language and in this

context can be treated as a metalanguage. A PragScript script encapsulates two levels of abstraction: meta-level provides access to the data stored by UMLStudio projects and describes generation of VHDL code, which is at the domain level of abstraction.

C. Generic model as a high level specification of metaprograms

Another view to metamodeling considers metamodels as generic models that captures aspects specific to the domain at hand. The model-based design provides several mechanisms for describing genericity in the domain. 1) Generalization (inheritance) relationship: abstract class (super-class) is a generic representation of its subclasses. 2) Template classes: parameterized classes in UML class diagrams. 3) Design patterns that can be seen as a generic representation of a class of design solutions or design activities [27, 61]. 4) Feature models that describe mandatory and optional features of a system using Feature Diagrams [62, 63].

Such generic models can be directly or indirectly implemented as metaprograms. The main mechanism of metaprogramming for describing genericity is the metalanguage itself. It allows representing common as well as variable parts of a domain system in a generic fashion. Here we focus on the implementation of template classes and feature models.

A template class defines a family of classes in UML models. Template classes can be implemented in SystemC using the template metaprogramming [64] technique, which is a special case of homogeneous metaprogramming. Template keyword is used to specify a generic skeleton for a class parameterized by generic parameters or types. To complete the declaration, a programmer must supply a concrete value for each of the template's parameters, which must be known at compile time. Specification of parameter values causes the template to be instantiated: replacing all occurrences of the parameter with its value creates an instance of the template. An example of template-based metaprogramming is presented in Fig. 4, where a parameterized gate class in UML and the corresponding SystemC generic gate module is presented. Template parameter T is used to select, whether the AND gate, or the OR gate will be implemented.

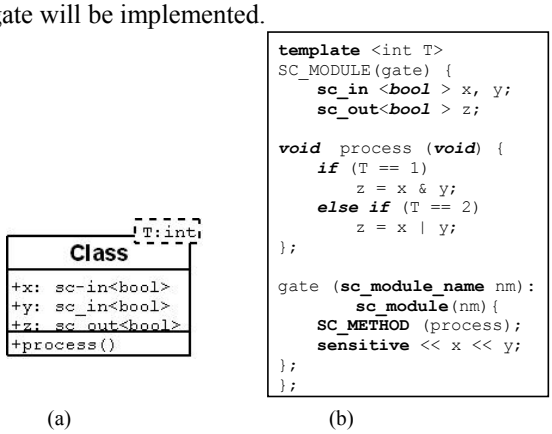

Figure 4. An example of a) parameterized class in UML, and b) template-based metaprogramming in systemc.

The reasons behind the introduction of the generic parameters are as follows: (1) to express the generalization of a model, when new generic parameters are introduced, and (2) to perform the customization of a model, when a specific instance is derived depending on the values of the generic parameters. For example, a parameter T (Fig. 4) generalizes expressions regardless of the implementation, whether it is a variable or a generic parameter. However, in case of the generic implementation, only one specific gate is instantiated (AND gate or OR gate), whereas in a non-generic implementation both gates will be created. Therefore, genericity allows achieving larger reusability, as well as better performance.

D. Feature model as a domain variability model

Feature models specify hierarchies of system features (external characteristics) in terms of commonality and variability, rather than describing all details. As such feature modeling is important for describing domain variability models. Features are primarily used in order to discriminate between system instances. Common features among different systems are modeled as mandatory features, while different features among them may be optional or alternative. Optional features represent selectable features for systems and alternative features indicate that no more than one feature can be selected for a system. The derivation of a system consists of traversing the feature tree in an orderly manner and selecting the optional features. The result is a system description containing all features in the system (a feature configuration).

As there can be many different configurations of a system encapsulated by a feature model, such model can be seen as a high-level specification of a metaprogram that specifies different variants of system implementation. Feature models can be used as domain variability models specifying system commonalty and variability and their relationship, whereas metaprogramming can be used to specify implicitly common parts of a system using domain language and variable parts of a system using metalanguage abstractions.

An example of such application of feature models is presented in Fig. 5 and Fig. 6. Fig. 5 shows feature model of a combinatorial circuit that necessarily has the following design characteristics: chip area, delay and power consumption. When considering circuit implementation, only one of the above-mentioned characteristics can be used as a design criterion. When developing a metaprogram for this circuit, the generic interface of a metaprogram can contain the evaluation of any of these characteristics allowing the user to select and generate the implementation satisfying the design constraints. Fig. 6 presents a metaprogram developed from such feature model. It encapsulated three different implementations of 2-bit comparator in VHDL optimized with three different design criteria in mind and the corresponding metadata for each implementation. However, only one implementation can be selected for generation by the user.

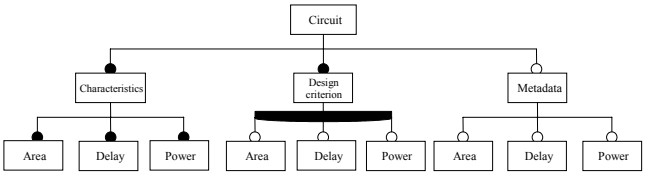

Figure 5. Feature model of a circuit.

A combination of feature modeling and metaprogramming is especially useful for developing product lines of embedded systems [10], where the aims are to achieve greater configurability, variability and adaptability to particular user and application requirements. A configurable product family may include millions of variants from that individual products are configured to meet particular customer needs. Therefore, the designers must use high-level models and abstractions to specify systems and their variants.

```
$
"Select design criterion:
        1 - area (area=1151 um2; delay=0.51 ns; power=1.828 uW/MHz)
        2 - delay (area=1553 um2; delay=0.49 ns; power=2.697 uW/MHz)
        3 - power (area=1370 um2; delay=0.67 ns; power=1.784 uW/MHz)"
        {1,2,3} sel:=1;
$
entity system_@case[sel,{area},{power},{delay}] is
 port(X0,X1,X2,X3: in bit;
        Y0,Y1,Y2: out bit);
end system_@case[sel,{area},{power},{delay}];

architecture behave of system_@case[sel,{area},{power},{delay}] is
begin
@case[sel,{
 Y0 <= (not ((X1 nor (X3 nand X2)) nor
((((X3 nor X2) nor X1) nor X2) nor X0)));
 Y1 <= ((X0 xor X2) nor (X3 xor X1));
 Y2 <= ((((X0 nand X1) nand X2) nand (X3 nand
((not X2) nand X0))) nor (X0 nor X1));
},{
 Y0 <= ((((X0 nand X2) nor X1) nor (X0 nor X0)) nor
((not ((X1 nand X3) nand X3)) nor ((X2 nor X2) nor X0)));
 Y1 <= ((X0 xor X2) nor (X3 xor X1));
 Y2 <= (((((X3 nand X2) nand X0) nand X2) nand (X3 nand (X0 nand X0)))
        nor ((X2 nand (not X1)) nand ((not X0) nand (not X1))));
},{
 Y0 <= (not ((X0 nor (((X3 nor X2) nor X1) nor X2)) xnor
((X2 nor X1) nor (((not X3) nor X1) nand X0))));
 Y1 <= ((X0 xor X2) nor (X1 xor X3));
 Y2 <= ((((X0 nand X1) nand X2) nand (X3 nand X2)) nor
(((X0 nor X1) nor X3) nor X0));
}]
end behave;
```

Figure 6. Open promol metaprogram of 2-bit comparator.

Table III summarizes the capabilities of model-based design and metaprogramming for implementing transformational design processes.

TABLE III. CAPABILITIES OF MODEL-BASED DESIGN AND METAPROGRAMMING FOR IMPLEMENTING TRANSFORMATIONAL DESIGN

Capability	**Model-based design**	**Metaprogramming methodology**
Parameteriza-tion	Implemented using parameterized classes and represented using template parameters	Implemented using generic parameters represented at generic interface
Representation	Real-world entities	Domain program families
Abstraction	Class is an abstract representation of a family of objects	Metaprogram is an abstract representation of a family of programs
Generalization	Two mechanisms: through inheritance and via templates	Via generic parameters represented at generic interface
Specialization	Via sub-classing hierarchies	By instantiating a metaprogram
Modification	May be implemented using parameterized template	Performed automatically using metalanguage abstractions
Generation	Class instances are generated from class templates at compile time	Domains programs are generated from metaprograms at construction time
Reuse	Implemented using inheritance: subclasses reuse attributes and methods of the superclass	Implemented by selecting values of generic parameters and generating domain programs
Separation of concerns	Class header is separated from class implementation; class attributes – from class methods; class – from its objects	Generic interface is separated from metaprogram body; metaprogram – from domain programs
Variability	Implemented using class constructor(s) that modify the values of class attributes	Implemented using metalanguage abstractions that modify domain programs

VI. DISCUSSION AND CONCLUSIONS

The metaprogramming methodology and its elements (variability model, reuse evolution model, modeling language and domain language relationship metamodels, transformational design techniques) have been supported by numerous case studies [21, 27, 43, 44, 49, 53, 56, 57, 58, 61, 63]. Based on these case studies and the research presented in this paper we can conclude that the higher abstraction level is used, the larger is the role of various transformation processes in the design. However, the effective specification, implementation and automation of transformation processes in the domain requires introduction of 1) high level transformation model (metamodel), and 2) generative technology for derivation of lower-level domain programs from higher-level design task specifications.

Heterogeneous metaprogramming is a domain language independent generative technology, which when compared to homogeneous metaprogramming better fits for wide scale domain generators (e.g., for product line oriented designs and huge soft IP repositories). The cost we need to pay for this is the need of a metalanguage and its environment and much more complicated verification. The use of the grey-box-based reuse evolution model within this technology can significantly reduce its deficiencies. The dedicated metalanguage as compared to the general-purpose programming language allows expressing metaspecifications more concisely, but a designer should learn yet another language and be confident in its reliability.

Metaprogramming per se requires the definition of a high-level domain model (metamodel or domain variability model) for its implementation. Such models describe high-level abstractions can be mapped to the lower level of abstraction and can serve both as a guideline for manual development of metaprograms or executable specifications

for producing domain programs or at least parts of metaprograms.

Due to increasing complexity of designed systems in the embedded system and hardware design domains and the complexity of the design process itself, design of modern systems require design automation and management of domain variability. Domain variability at best can be modeled using feature diagrams and other relationships within the domain can be modeled using a subset of UML (though UML allows specifying variability at a certain degree, too). These technologies introduce new layers of abstraction above traditional HDL specifications. Heterogeneous metaprogramming brings the expressive power to bridge the gap between different levels of abstraction and to describe genericity and variability in the domain explicitly.

REFERENCES

[1] L. Lavagno, G. Martin, B. Selic (eds.). UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Publishers, 2003.

[2] M. Keating, P. Bricaud. Reuse Methodology Manual for System-on-a-Chip Designs. Kluwer Academic Publishers, 2001.

[3] A. Sangiovanni-Vincentelli, G. Martin, "Platform-based design and software design methodology for embedded systems", IEEE Design and Test of Computers, 18(6), 2001, pp. 23-33.

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

[5] A. Jantsh, S Kumar, A. Hemani, "Ruby: A metamodel to study concepts in electronic system design", IEEE Design & Test of Computers, 2001.

[6] L.P. Carloni, F. De Bernardinis, A. L. Sangiovanni-Vincentelli, M. Sgroi, "The art and science of integrated systems design", Proceedings of the 28th European Solid-State Circuits Conference ESSCIRC, 2002, pp. 25-36.

[7] W. Műller, W. Rosenstiel, J. Ruf. SystemC: Methodologies and Applications. Kluwer Academic Publishers, 2003.

[8] G. Martin, "IP reuse and integration in MPSoC: Highly configurable processors", MPSoC'04, 8 July 2004.

[9] E. Aarts, R. Roovers, "IC design challenges for ambient intelligence", Proc. of Design, Automation and Test in Europe Conf. (DATE 03), Munchen, Germany, 3-7 March 2003, pp. 2-7.

[10] H.-K. Kim, "Applying Product Line to the Embedded Systems", Proc. of Int. Conf. on Computational Science and Its Applications, ICCSA 2006, Glasgow, UK, May 8-11, 2006. LNCS 3982 Springer 2006, Part 3, pp. 163-171, 2006.

[11] G. Martin, "UML for embedded systems specification and design: motivation and overview", Proc. of Design Automation and Test in Europe (DATE 2002), 4-8 March 2002, Paris, France, pp. 773-775.

[12] M. Edwards, P. Green, "UML for hardware and software object modeling", in L. Lavagno, G. Martin, B. Selic (eds.), UML for Real, Kluwer Academic Publishers, 2003, pp. 127-148.

[13] G. Jong, "A UML-based design methodology for real-time and embedded systems", Proc. of Design Automation and Test in Europe (DATE 2002), 4-8 March 2002, Paris, France, pp. 776-778.

[14] Q. Zhu, A. Matsuda, S. Kuwamura, T. Nakata, M. Shoji, "An object-oriented design process for System-on-Chip using UML", Proc. of the 15th Int. Symp. on System Synthesis (ISSS 2002), October 2-4, 2002, Kyoto, Japan, pp. 249-254.

[15] T. Beierlein, D. Frvhlich, B. Steinbach, "Model-Driven compilation of UML-models for reconfigurable architectures", in 2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES '04), Toronto, Ontario, Canada, May 25-28, 2004.

[16] T. Schattkowsky, W. Müller, "Model-based design of embedded systems", Proc. of 7th IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04), May 12 - 14, 2004, Vienna, Austria, pp. 121-128.

[17] D.H. Akehurst and S.J.H. Kent, "A relational approach to defining transformations in a metamodel", in J.-M. Jezequel, H. Hussmann, S. Cook, eds. UML 2002 – The Unified Modeling Language: 5th International Conference, Springer-Verlag, 2002, pp. 243 – 258.

[18] J.M. Favre, "Metamodels and models co-evolution in the 3D software space", In ELISA 2003, associated with ICSM 2003, Amsterdam, The Netherlands, September, 2003.

[19] K.-K. Lau and Z. Wang, "A Taxonomy of software component models", in 31st Euromicro Conference on Software Engineering and Advanced Application, August 31–September 2, Porto, Portugal, 2005.

[20] R. Gitzel, T. Hildenbrand. A Taxonomy of metamodel hierarchies. Research Report, Department of Information Systems. University of Mannheim, 2005.

[21] R. Damaševičius, "On the Application of Meta-Design Techniques in Hardware Design Domain", International Journal of Computer Science (IJCS), Vol. 1, No. 1, pp. 67-77, 2006.

[22] F. Doucet, S. Shukla, R. Gupta, "Introspection in System-Level Language Frameworks: Meta-level vs. Integrated", Proc. of Design Automation and Test in Europe Conference (DATE 2003), 3-7 March 2003, Munich, Germany, 382-387.

[23] F. Seyler, P. Aniorte, "A Component metamodel for reuse-based system engineering", Workshop in Software Model Engineering, October 1st 2002, Dresden, Germany.

[24] D. Orr, "Model driven software development through the integration of three models", OOPSLA Workshop on Best Practices for Model Driven Software Development, 2005.

[25] G. Karsai, M. Maroti, A. Ledeczi, J. Gray, J. Sztipanovits, "Composition and cloning in modeling and Metamodeling", IEEE Trans. on Control Systems Technology 12 (2004), pp. 263–278.

[26] B. Schätz, A. Pretschner, F. Huber, J. Philipps, "Model-based development of embedded systems", in J.-M. Bruel, Z. Bellahsene (Eds.), Advances in Object-Oriented Information Systems (OOIS'2002) Workshops, Montpellier, France, Springer LNCS, 2002.

[27] R. Damaševičius, V. Štuikys, "Application of UML for Hardware Design Based on Design Process Model", Asia South Pacific Design Automation Conference (ASP-DAC 2004), January 27-30, 2004, Yokohama, Japan, pp. 244-249.

[28] L. Tratt, "Model transformations and tool integration", Journal of Software and Systems Modelling, 4(2), May 2005, pp. 112-122.

[29] A. Lédeczi, G. Nordstrom, G. Karsai, P. Völgyesi, M. Maróti, "On metamodel composition", Proc. of the IEEE Int. Conference on Control Applications, CCA 2001, Mexico City, Mexico, pp. 756-760.

[30] P.-A. Muller, P. Studer, J.-M. Jézéquel, "Model-driven generative approach for concrete syntax composition", OOPSLA & GPCE Workshop on Best Practices for Model Driven Software Development, Vancouver, 2004.

[31] T. Levendovszky, G. Karsai, M. Maroti, A. Ledeczi, H. Charaf, "Model reuse with metamodel-based transformations", in C. Gacek, (ed.), Proc. of 7th Int. Conf. on Software Reuse: Methods, Techniques, and Tools, ICSR, LNCS vol. 2319. Springer, 2002, pp. 166-178.

[32] S. Deelstra, M. Sinnema, J. van Gurp, J. Bosch, "Model driven architecture as approach to manage variability in software product families", Proc. of Workshop on Model Driven Architecture: Foundations and Applications (MDAFA'2003), June 2003, pp. 109-114.

[33] J. van Wijngaarden, E. Visser. Program Transformation Mechanics: A Classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University, May 2003.

[34] L. Kuzniarz, M. Staron, "On model transformations in UML-based software development process", Software Engineering and Applications'03, Marina del Rey, CA, 2003.

[35] K. Czarnecki, S. Helsen, "Classification of model transformation approaches", OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[36] W. Wu, I., Sander, A. Jantsch, "Transformational system design based on a formal computational model and skeletons", Forum on Design Languages (FDL'2000), September 4-8, 2000, Tübingen, Germany.

[37] O. de Moor, G. Sittampalam, "Generic program transformation", Proc. of the 3rd Int. Summer School on Advanced Functional Programming, Braga, Portugal, September 12-19, 1998, Springer LNCS 1608, pp. 116-149, 1999.

[38] T. Mens, K. Czarnecki, P. van Gorp, "A Taxonomy of Model Transformations", in J. Bézivin, R. Heckel (Eds.), Proc. of Language Engineering for Model-Driven Software Development, 29 February - 5 March 2004. Dagstuhl Seminar Proceedings 04101, Schloss Dagstuhl, Germany 2005.

[39] M. Gogolla, A. Lindow, M. Richters, P. Ziemann, "Metamodel Transformation of Data Models", Workshop in Software Model Engineering, October 1st 2002, Dresden, Germany.

[40] S. Nedunuri, W. Cook, "Transforming Declarative Models Using Patterns in MDA", OOPSLA & GPCE Workshop on Best Practices for Model Driven Software Development. Vancouver, 2004.

[41] S.R. Judson, R.B. France, D.L. Carver, "Specifying model transformations at the metamodel level", WiSME@UML'2003 - UML Workshop in Software Model Engineering, October 21, 2003, San Francisco, USA.

[42] J. Kovse, "Generic model-to-model transformations in MDA: Why and How?", in OOPSLA 2002 Workshop on Generative Techniques in the context of Model Driven Architecture. Seattle, November 4-8, 2002.

[43] R. Damaševičius, V. Štuikys, "Soft IP customization models based on high-level abstractions", Information Technology and Control, 2005, Vol. 34, No. 2, pp. 125-134.

[44] 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, 2003, pp. 229-250.

[45] E.A. Lee, A. Sangiovanni-Vincentelli, "A Framework for comparing models of computation", IEEE Transactions on CAD, Vol. 17, No. 12, 1998, pp. 1217-1229.

[46] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, A. Sangiovanni-Vincentelli, "Addressing the system-on-a-chip interconnect woes through communication-based design", Proc. of Design Automation Conference, 18-22 June 2001, pp. 667-672.

[47] J. Rowson, A. Sangiovanni-Vincentelli, "Interface-based design", Proc. of the 34th Design Automation Conference (DAC 97), June 9-13, 1997, Anaheim, CA, USA, pp. 178-83.

[48] L. Entrena, C. Lopez, E. Olias, "Automatic generation of fault tolerant VHDL designs in RTL", Forum on Design Languages FDL'2001, Lyon, France, 2001.

[49] R. Damaševičius, V. Štuikys, E. Toldinas, "Embedded program specialization for multiple criteria trade-offs", Electronics and Electrical Engineering, 8(88), pp. 9-14, 2008.

[50] M. Kuulusa, J. Nurmi, J. Takala, P. Ojala, H. Herranen, "A flexible DSP core for embedded systems", IEEE Design & Test of Computers, October-December, 1997, pp. 60-68.

[51] Domain Engineering and Domain Analysis, Software Technology Roadmap. http//www.sci.cmu.edu/str/descriptions/deda_body.html, 2005.

[52] A. Gerstlauer, D.D. Gajski, "System-level abstraction semantics", Proc. of 15th Int. Symposium on System Synthesis (ISSS'02), October 2-4, 2002, Kyoto, Japan, pp. 231-236.

[53] R. Damaševičius, V. Štuikys, "Wrapping of Soft IPs for Interface-based Design Using Heterogeneous Metaprogramming", INFORMATICA, 2003, Vol. 14, No. 1, pp. 3-18.

[54] B.W. Boehm, "A spiral model of software development and enhancement", IEEE Computer, 1988, 21(5):61-72.

[55] T. Sheard, "Accomplishments and research challenges in meta-programming", in 2nd Int. Workshop on Semantics, Application, and Implementation of Program Generation (SAIG'2001), Florence, Italy. LNCS, vol. 2196, 2001, Springer, pp. 2-44.

[56] V. Štuikys, R. Damaševičius, "Variability-Oriented Embedded Component Design for Ambient Intelligence Systems", Information Technology and Control, 36(1), pp. 16-29, 2007.

[57] V. Štuikys, R. Damaševičius, G. Ziberkas, "Open PROMOL: An experimental language for domain program modification", in A. Mignotte, E. Villar, L. Horobin (Eds.), System on Chip Design Languages, Kluwer Academic Publishers, 2002, pp. 235-246.

[58] V. Štuikys, R. Damaševičius, "Soft IP customization model based on metaprogramming techniques", INFORMATICA Vol. 15, No. 1, 2004, pp. 111-126.

[59] Object Management Group (OMG). MOF: MetaObject Facility. http://www.omg.org/mof/

[60] UMLStudio. http://www.pragsoft.com.

[61] R. Damaševičius, G. Majauskas, V. Štuikys, "Application of Design Patterns for Hardware Design", Proc. of 40th Design Automation Conference DAC 2003, June 2-6, Anaheim, CA, USA, pp. 48-53.

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

[63] R. Damaševičius, V. Štuikys, E. Toldinas, "Domain Ontology-Based Generative Component Design Using Feature Diagrams and Meta-Programming Techniques", in R. Morrison, D. Balasubramaniam, and K. Falkner (Eds.), Proc. of 2nd European Conference on Software Architecture ECSA 2008, September 29 - October 1, Paphos, Cyprus. LNCS 5292, pp. 338-341. Springer-Verlag, 2008.

[64] D. Abrahams, A. Gurtovoy. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, 2004.