International Journal of Electrical, Electronic and Communication Sciences
ISSN: 2517-9438
Vol:2, No:2, 2008

On the Application of Meta-Design Techniques in Hardware Design Domain

R. Damaševičius

***Abstract*—**System-level design based on high-level abstractions is becoming increasingly important in hardware and embedded system design. This paper analyzes meta-design techniques oriented at developing meta-programs and meta-models for well-understood domains. Meta-design techniques include meta-programming and meta-modeling. At the programming level of design process, meta-design means developing generic components that are usable in a wider context of application than original domain components. At the modeling level, meta-design means developing design patterns that describe general solutions to the common recurring design problems, and meta-models that describe the relationship between different types of design models and abstractions. The paper describes and evaluates the implementation of meta-design in hardware design domain using object-oriented and meta-programming techniques. The presented ideas are illustrated with a case study.

***Keywords*—**Design patterns, meta-design, meta-modeling, meta-programming.

I. Introduction

HARDWARE (HW) and software (SW) components are essential parts of any embedded system. The complexity of such systems is growing continuously. For example, complexity of System-on-Chip (SoC) in terms of logic transistors that can be integrated on a chip is increasing at the rate of 58% per year (Moore's law). However, the design productivity is increasing at the rate of 21% per year only. This fact is known in the Electronic Design Automation (EDA) community as *design productivity gap* [62].

Due to the ever-increasing complexity of such systems, their development must inevitably rely on the usage of higher-level models and abstractions. Most of current research efforts in the domain are aimed at bridging design productivity gap as well as raising the level of abstraction, increasing IP (*Intellectual Property*) reuse and unifying HW and embedded SW design methodologies [67]. The researchers have to analyze and evaluate the existing HW modeling and design techniques as well as to develop or adopt the new ones that can provide higher productivity and shorten time-to-market. In general, reuse-oriented system engineering can be categorized into as *Domain Engineering* or Application *Engineering* [36].

Manuscript received March 10, 2006.
R. Damaševičius is with the Software Engineering Department, Kaunas University of Technology, Studentu 50, 415, 51368 Kaunas, Lithuania (phone: 370-37-300399; fax: 370-37-300352; e-mail: robertas.damasevicius@ktu.lt).

Domain Engineering is a design-for-reuse methodology for creating families of domain assets. It aims at creating new solutions or/and inventing new design technologies, such as product lines [21]. Domain Engineering usually requires careful technical and economic analysis and goes through all system development phases.

Application Engineering is a design-with-reuse methodology of producing specific systems by using the pre-designed assets used to solve recurring design problems in the domain. The original design effort is applied only once. The successful implementation of this approach leads to systematic design reuse for a specific set of domain products and systems.

The main difference between these two categories of system engineering is in the level of abstraction and generalization. Design-for-reuse requires the development of generic domain models and components that can be applicable in many contexts, whereas design-with-reuse deals with specialization of the already developed generic components to the given context of application. This separation of concerns in design allows raising the abstraction level in the domain to a meta-level and increasing design productivity.

Here we consider meta-design – design of systems at a meta-level of abstraction. The aim of this paper is (1) to analyze the basic concepts of meta-design, including high-level abstractions and models, (2) to describe the main techniques of meta-design used for meta-modeling and meta-programming, and (3) to demonstrate how meta-design can be applied for designing HW systems using object-oriented and meta-programming techniques.

Our previous research has been focused on generic component models and meta-programming for describing variability and generalization in a domain [17, 18, 19, 64, 65]. The novelty of this paper is (1) a unified view at meta-modeling and meta-programming as separate stages of the same process, the meta-design, and (2) formulation of the main principles of meta-design in the context of HW design.

The remainder of this paper is structured as follows. Section 2 analyses the basic concepts of meta-design, its stages, main techniques and levels of abstraction. Section 3 presents a case study for the application of meta-modeling and meta-programming in HW design domain. Section 4 presents discussion on meta-design. Finally, Section 5 presents the conclusions.

II. Basic concepts of meta-design

Meta-design [29, 30] is an emerging system engineering methodology that extends the traditional system design beyond the development of a specific system to include design for change, modification and reuse. A particular emphasis is given to (1) increasing participation of users in system design process, and (2) evolutionary development of systems during their use time when dealing with future uses and problems unanticipated at domain analysis and system design stages. Such systems must be flexible and evolve, because they cannot be completely designed prior to their use [30].

Meta-design focuses on designing "design processes" [19] or "product lines" [70] rather than designing the specific content or system. It focuses on general (generic) structures and processes, rather than on fixed objects and contents. To support evolvability and adaptability of designed systems to changing user requirements and the context of usage, meta-design focuses on the development of unified system design frameworks and meta-environments [37], development of generic component models [64], definition of languages and language meta-models [65], abstract definition of architectural and behavioral models ("well-proven" models, patterns) commonly used in the domain [18], development of mechanisms that permit users to implement complex design transformations and customizations [17], and application of automatic domain analysis and design space exploration tools.

As traditional system development consists of three main stages: analysis (identification of user/product/market requirements), modeling (development and simulation of a system model), and programming (implementation of a final SW product), the process of meta-design can be separated into three stages: (1) meta-analysis, (2) meta-modeling, and (3) meta-programming. We explain these below in detail.

A. Meta-analysis

Meta-analysis [20] is analysis and knowledge mining of the multidimensional design space using mathematical and statistical methods (such as multidimensional scaling, hierarchical clustering, etc.), automatic design space exploration [74, 75, 76] and optimal solution search methods such as optimization using genetic algorithms [77] or Simulated Annealing [78], extraction and evaluation of multiple design alternatives, automatic feature extraction and analysis, search, retrieval and evaluation of Intellectual Property (IP) components, design context analysis using automatic analysis tools (such as parsers), analysis and prediction of possible changes for anticipation.

As complexity of design systems is ever-growing, meta-analysis of multi-dimensional design space can not be achieved using a single domain analysis method, but rather by using a combination thereof. Below, we present a brief survey of methods that can be used in meta-analysis stage.

Multi-Dimensional Separation of Concerns [54] understands design concerns in terms of a n-dimensional design space, called a hyperspace. Each dimension is associated with a set of similar concerns, such as a set of component instances; different values along a dimension are different instances. A hyperslice is a set of instances that pertain to a specific concern. A hypermodule is a set of hyperslices and integration relationships that dictate how the units of hyperslices are integrated. The method is especially useful in domains where a great variety of requirements exist at different layers of abstraction such as in embedded system design.

Multidimensional Scaling (MDS) [27] is a set of mathematical techniques that allow uncovering hidden structure in complex data. MDS can be used to identify similar objects in multi-dimensional design space such as design space of a component family. Suppose, we have a set of domain objects characterized by a number of features and that a measure of similarity between objects is known. MDS maps each object of a high-dimensional space to a lower dimensional (usually 2D or 3D) space in which each object is represented by a point, and the distances between points resemble the original similarity information. This geometrical configuration of points reflects the hidden structure of the domain data and may help to make it easier to understand.

Parsing [66] is a domain-specific analysis method for automatic analysis of the abstract representation of the domain - source code. It allows better understanding of the domain and extracting the application-specific information for IP customization. If used with other methods such as substring amplification [79], it can help uncover hidden patterns or templates in source code of designed systems.

The challenges for the meta-analysis are as follows: (1) Analysis of multidimensional domain data to uncover its particular structure (patterns, repeating commonalties, templates) or peculiarities (anomalies, optimal solutions). (2) Representation and interpretation of meta-data.

B. Meta-modeling

The basic motivation for meta-modeling [4, 6, 26, 32, 41, 42, and 46] is to improve productivity in SW and HW development. It allows to achieve this by raising the level of abstraction at which the primary SW artifacts are described and developed. The result of meta-modeling is a meta-model – a higher-level model that describes conceptual relationships between lower-level models and elements thereof, design methods, abstractions and tools.

Meta-modeling includes the following activities: (1) definition of concepts for creating and using domain models, (2) description of domain-specific modeling languages and their notation, (3) description of relationship between real world elements and system models, (4) description of reuse, customization and transformation mechanisms applied to models and their underlying meta-models, (5) definition of design processes how to apply the meta-models and the corresponding mechanisms, (6) definition of concepts and standards to facilitate the interchange of meta-models and models between different design teams and tools, (7) definitions of concepts to facilitate user-defined mappings from models to other SW artifacts (including domain code).

The first order task for a meta-designer is to recognize the well-understood domains, to extract the well-proven models from them and to apply the models and abstractions in the design of a system. Well-understood models are frequently used high-level design abstractions, e.g., Finite State Machines (FSMs) for describing behavior of complex systems in the domain. Other examples are Triple-Redundancy Model (TRM) in fault tolerant design [25] and communication protocols in interface synthesis [56, 58].

Meta-modeling or model-driven [5, 11, 24, 52, 53] design is deeply rooted in the domain of HW and embedded system design. The designers use a variety of models, for instance, models of computation (MoC) are formal and abstract definitions of a component [44]. MoC allow analyzing the intrinsic properties of a component such as execution time or memory space of an algorithm while ignoring many implementation issues. The design is iterative - a design is transformed from an informal description into a detailed specification usable for manufacturing. Examples of MoC are Boolean circuits, Petri nets, discrete events, data flows, etc.

Component models such as Virtual Component [1], MetaCore [50] or MetaRTL [72] usually deal with the problems of representation, retrieval and reuse of HW/SW components for IP libraries, IP providers and IP users. These models either allow customization of components with respect to user requirements for successful soft IP reuse, or enable convenient soft IP retrieval and sharing. The design focuses on design space exploration, parameterization, and generation of soft IPs. The proposed solutions are usually language-centric (pre-processing, extensions of languages, etc.).

Architectural models such as platforms [13, 39, 51, 60] focus on embedded system design based on IP reuse. Platforms are common architectures based on principal components that remain fixed within a certain degree of parameterization. Such platforms support a variety of applications in a given domain thus achieving some generalization. Platforms focus on the communication-based design that is independent of the behavior of particular components rather than on the design of functionality. A specific application is derived from the platform using refinement (specialization).

Abstract high-level models such as the ones described using UML diagrams [12] are also beginning to be widely used for HW and embedded system design [34, 49, 73]. The UML models allow a high level specification of a system, provide support for better soft IP reusability and adaptability, as well as they improve the documentation for further reuse and maintenance of a system.

Design patterns [31] are the abstraction for representing common design solutions in an implementation-independent way using UML class diagrams. Design patterns are widely used in SW domain for creating SW systems using previous successful design experience. Recently, they were also adopted for HW and embedded system design [22, 23, 55, 61, 71].

The challenges for the meta-modeling are as follows: (1) To discover and describe well-proven models and architectural patterns that are generally used by designers in the domain. (2) To describe the implementation of well-proven models in terms of high-level abstractions and design techniques using a well-known notation, and (3) to seek for the design methodologies and tools that allow for implementing the well-proven models (semi-)automatically.

C. *Meta-programming*

Meta-programming was known and used for a long time in the past, especially in program synthesis [63]. Now the the application of meta-programming is much wider and covers domain language implementation, including compiler generation [66], application and SW generators [9], product lines [8], generic component design [10], program transformations [48], program evaluation and specialization [68], SW maintenance, evolution and configuration [16], middleware applications [14], XML-based web applications [47], etc. Furthermore, the meta-programming techniques closely relate to the novel technologies, such as generative [15] and aspect-oriented programming [40].

From the perspective of abstraction, meta-programming means programming at a higher level of abstraction. Ryman [59], for example, gives the following definition. Meta-programming is “the technique of specifying generic SW source templates from which classes of SW components, or parts thereof, can be automatically instantiated to produce new SW components”. A meta-language, which is a mechanism for introducing a higher-level of abstraction, does not appear in this definition. It is assumed that source templates are higher-level generic abstractions of the source language itself.

A program written in a meta-language is a meta-program. A meta-program is a program that treats another program as data. According to Batory [7], a meta-program is “a program that generates the source of the application ... by composing pre-written code fragments”. Examples of meta-programs are application generators, and building application generators such as parser generators is an example of meta-programming.

Commonly meta-programming is used to provide mechanisms for writing generic code, i.e. explicitly implementing generalization in the domain. Domain language implements commonalties in a domain, while a meta-language allows developers to specify variations to be implemented in the domain system, and to synthesize customized implementations by composing domain code fragments. The generalization is achieved by the parameterization of differences in different domain program representations, which allows representing domain components with many commonalties in a compact way. We use meta-programming for implementing the generative technology in HW/SW domain [64]. It provides capabilities for expressing domain variability. The product of meta-programming is a meta-program (or meta-program), which describes a family of the related functionality in a narrow well-defined domain.

A meta-program consists of a generic interface and a family of related domain program instances encapsulated with their

modification algorithm. A generic interface describes the generic parameters of a meta-program. The modification algorithm describes generation of a particular instance depending upon values of the generic parameters. At a lower layer of abstraction there is domain language code that describes common parts of component family. At a higher layer of abstraction there is meta-language code that describes variable parts of component family. As a meta-program is a concise representation of its instances, it can be treated as a generic component, too. Together with its environment, a meta-program is a domain program generator.

Flexibility of generalization and domain code generation can be enhanced significantly either through extensions of the domain languages or through the usage of the external meta-language. Meta-language describes the syntax and semantics of generalization, and meta-programming paradigm defines the rules and methods for implementing generalization.

Summarizing, meta-programming can be defined as a programming technique that achieves generalization via manipulation with other program structures. Meta-programs can be represented using the same programming principles and constructs (if, case, for loop) as domain programs, however they manipulate on program representations, not data. In other words, meta-programming is a higher-order programming technique for generalization.

To achieve the prescribed aims, meta-programming uses separation of concerns, parameterization, and parameter dependency knowledge. Separation of concerns separates each domain problem into a distinct generic component or sets of components used to generate target program. Parameterization increases reusability by providing parameterized components, which can be instantiated for different choices of parameters. Parameter dependency knowledge allows capturing specific information about the parameter dependencies, default settings and illegal combinations.

The challenges for meta-programming are as follows: (1) Identification similar (“look-alike”) components in a domain. (2) Selection of a suitable meta-language for meta-programming. (3) Selection of optimal size and number of designed meta-programs. (4) Identification and separation of dependable and undependable meta-parameters. (5) Overgeneralization problem.

D. Basic techniques of meta-design

Here the analyze only the basic techniques of meta-design as follows: separation of concerns for meta-analysis, generalization for meta-modeling, and generation for meta-programming. We explain these below in detail.

1) Separation of concerns

Separation of concerns is primarily focused on (1) the separation of domain commonalities and variabilities. Commonalities are fixed concepts that are common to all parts of (sub-)domain. Variabilities are variable concepts that are unique to every domain object (component, system, etc.), and depend upon certain design concerns or aspects. Separation of concerns leads to decomposition of SW into manageable and comprehensible parts. The second task of separation of concerns is (2) to identify different design concerns that are meaningful to the anticipated user requirements.

Separation of concerns for meta-design must involve the following activities:

(1) Formulation of anticipated design problems from the client's perspective (i.e., functionality of the system as the client expects it).

(2) Identification and separation of user- and application-specific concerns along multiple and arbitrary dimensions of design space.

(3) Partitioning and structuring of dependable and orthogonal concerns into groups or dimensions. Dependable concerns are closely related and depend upon each other (e.g., increasing speed would most certainly increase the consumption of power in a chip, thus speed and power concerns are dependable). Orthogonal concerns are independent upon each other.

(4) The ability to handle new concerns or their dimensions dynamically as they arise at use time.

(5) Concern-based composition of domain systems.

2) Generalization

Generalization is a design technique for expressing and representing domain models and components at higher levels of abstraction. Introduction of generalization usually means transition to the higher level of abstraction where domain knowledge can be represented and explained more comprehensibly and effectively.

Generalization identifies commonalties among a set of domain entities and widens an object (component, system) in order to encompass a larger domain of objects (systems, applications) of the same or different type. Commonality may refer to essential features of a design entity such as attributes or behavior, or may concern only the similarity in description. The result of generalization is a generic component (model), which compactly represents a set of similar components.

Therefore, we can describe generalization as a design technique that is oriented at unifying similar domain objects into a single generic component (model), which encapsulates their similarities and differences. Thus, generalization allows (1) to encapsulate related domain concepts thus simplifying, both quantitatively and qualitatively, the design space; and (2) to reduce the size and improve the structure of the domain component libraries. Thus, generalization allows introducing more simplicity into the domain.

Usually there are many levels of abstraction in the domain of interest. Therefore, we can distinguish, for example, generalization of models at a higher level of abstraction, and generalization of domain components at a lower level of abstraction. The latter is usually achieved by applying some form of generative technologies such as pre-processing or

International Journal of Electrical, Electronic and Communication Sciences
ISSN: 2517-9438
Vol:2, No:2, 2008

meta-programming, whereas the former is achieved using the meta-modeling techniques. This results in a modeling of domain information at different levels of abstraction.

3) Generation

Generation is a process of transformation between the higher-level representation of a domain system (model) and the lower-level implementation in domain language. To implement generation, the designer must built a meta-model that describes (1) a mapping between the modeling language and domain language abstractions, and (2) a set of translation rules that implement a mapping.

Here, we analyze generation in HW design domain from HW models described using UML into VHDL code. A mapping is described semi-formally using UML meta-model, i.e., the model that describes the syntax of UML diagrams using a subset of UML. A meta-model consists of a class diagram, where classes describe the syntactic components of the used UML diagram. A meta-model for mapping UML to VHDL was initially described in [19] and is extended now. Below, we present a mapping between UML class diagrams and a structural subset of VHDL (see Figure 1). VHDL abstractions are shown in parentheses.

Elements of UML class diagrams are classifiers, relationships and features. Classifiers are interfaces and classes that describe basic design blocks. Relationships (Figure 1, a) describe different types of connections and associations between classifiers. Features (Figure 1, b) describe parameters, attributes and methods of classifiers. We map an abstract class (interface) to a VHDL entity. A class that realizes an abstract class is mapped to VHDL architecture. Class parameters are mapped to a VHDL generic statement, class attributes - to the VHDL ports (public) and signals (private), and class methods – to the VHDL processes (procedures). The composition relationship describes composition of a system from the components and is mapped to a VHDL port map statement. The inheritance relationship means that a VHDL entity inherits the I/O ports from a base entity.

Once the mapping between UML and VHDL has been defined, rules that describe the translation process between UML and VHDL can be formulated. The aim of the translation rules is to describe how an instance of a UML meta-model (i.e., any UML model described using a subset of UML defined in a meta-model) can be transformed into an instance of a target model (i.e., a concrete VHDL specification that describes the implementation of a HW model specified using UML). These rules can be implemented manually by a HW designer, or automatically using a dedicated translation tool or code generator using a wide range of code generation strategies [61].

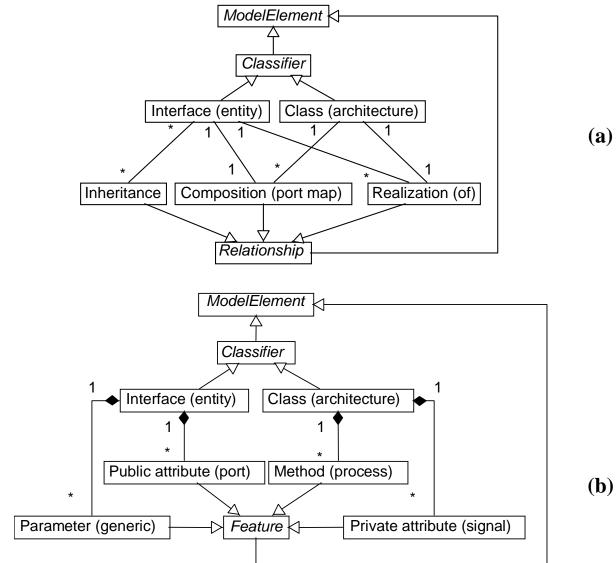

Fig. 1 A mapping between UML class diagrams and VHDL structural abstractions: (a) relationships and (b) features

E. Levels of abstraction and generalization in domain

There are many levels of abstraction used to express the domain content, therefore, generalization can also be represented in many different forms [35].

(1) Hierarchy organizes the domain commonalties into a tree-like structure. A hierarchical organization of components based on a relationship of generalization/specialization (or "is-a") is an important principle in object-oriented programming.

(2) Polymorphism captures commonality in different object types in object-oriented (OO) design. The generality is achieved by allowing the program to uniformly manipulate objects of different classes provided that these share the common properties.

(3) Genericity expresses the commonality using parameters. Genericity captures some common component properties that are expressed in terms of other unspecified abstractions that are denoted by parameters defined externally. These abstractions can be described using another higher-level language, i.e., a meta-language.

(4) Pattern presents an abstract and general solution (the key components and relationships between them) to a commonly occurring design problem. A concept of pattern is widely known in system modeling in general [2], and OO SW development in particular [3, 31, 57].

Therefore, generalization can be introduced into the domain as a multi-staged model having four different levels of abstraction as follows:

(1) Domain abstraction level – the organization of domain data (components) into the tree-like hierarchies, where a root is a generalization of the descendants.

(2) Meta-program level – the development of the generic components (programs) using the internal mechanisms of the domain language (polymorphism) or an external language

(meta-language).

(3) Model level – the introduction of models that describe a specific domain problem in general.

(4) Meta-model level – the representation of the model semantics using more abstract meta-models. A composition of several meta-models implements a platform.

While the first two levels are usually introduced using textual languages (either object-oriented or the meta-programming ones), the last two levels are can be introduced using standard modeling language UML. Recently, UML also began to be used more widely in HW and embedded system domains [28, 34, 43, and 49].

Finally, in Figure 2, we demonstrate how different levels of abstractions relate between themselves in the context of platform-based design of embedded systems. At the highest level of abstraction is a platform – a composition of several different meta-models (design patterns), i.e., general, well-proven and well-defined solutions for a narrow and well-understood domain problem.

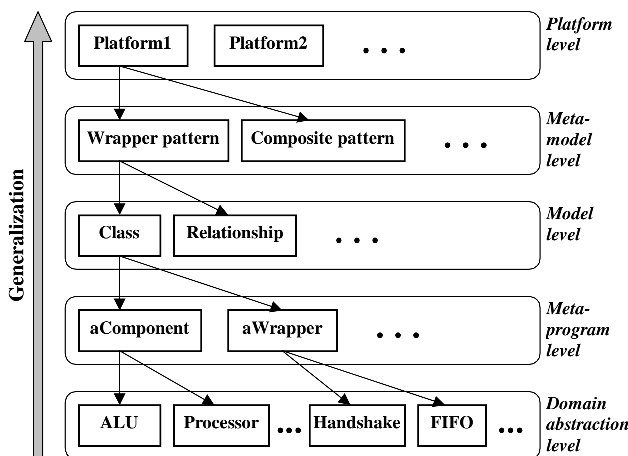

Fig. 2 Relationship between different levels of abstraction in Wrapper design pattern

In our example, a meta-model under consideration is a *Wrapper design pattern* [18], which is a generalization of several UML class diagrams that describe the implementation of communication control for different applications at the meta-model level. The elements of class diagrams, such as classes and relationships, are a generalization of the several different generic domain components and relationships between them at the model level. Finally, generic components, such as aComponent and aWrapper, are a generalization of specific domain components (e.g., ALU, Processor, Handshake FSM, FIFO FSM) at the meta-program level. The arrows show how the refinement is applied from the highest level of abstraction until the final implementation on the domain level is obtained.

III. Case Study

A. Meta-modeling

Our aim is to obtain the invariant, variant, and specific parts of the system and describe the relationship model between these parts and between the high-level model of a system and lower-level implementation. To do this we need to have some knowledge about the domain and perform meta-modeling, i.e. modeling of the domain at a higher abstraction level using a general domain vocabulary.

Figure 3 demonstrates the generalization of similar UML models into a design pattern for communication sub-domain. Figure 3 a) and b) show an extension of two common components: ALU and Processor with two different implementations of communication protocols, Handshake and FIFO, respectively. Figure 3 c) demonstrates a generalization of communication models using a Wrapper design pattern [17, 18] for any HW component cComponent and any wrappers cWrapper1, cWrapper2 that implement a particular communication protocol.

In HW design domain, we interpret the Wrapper design pattern as follows. The abstract class (entity in VHDL) aWrapper inherits the I/O ports of the aComponent, and declares new I/O ports for wrapper functionality. The class (architecture in VHDL) cComponent implements the functionality of entity aComponent. The architectures cWrapper1 and cWrapper2 implement the functionality of aWrapper and contains the aComponent. Essentially, this description means that cWrapper1 (or cWrapper2) wraps cComponent with a new functionality.

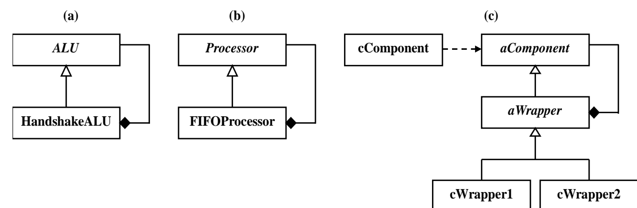

Fig. 3 Generalization of UML models into a design pattern

The obtained meta-model describes the results of domain analysis and does not provide details and concrete values. For example, there is nothing stated about the degree of some HW constraints such as timing and their design characteristics. Those details require a more in deep expert knowledge, because they a related with physical implementation of the schematics at the technological library level. The result of meta-modeling is only a domain-independent pattern of a system, which must be further refined by introducing the domain-specific details and constraints such as gate libraries and timing in HW design domain.

The relationship between this design pattern and its implementation must be described using another meta-model, an example thereof is given in Figure 1.

B. Modeling

To illustrate the application of generalization for modeling domain entities, we present the UML class diagram for representing different kinds of gates in a single gate hierarchy (see Figure 4). Parameterized superclass Gate abstractly represents all kinds of gates in the hierarchy, whereas other parameterized classes (AndGate, OrGate, XorGate) represent the particular implementations of the gate schematics. The role of the generalization relationship is to denote a taxonomic relationship between a more general element and elements that are more specific. The generic parameters used in the model represent the delay of a particular HW element (DELAY) and the width of the I/O signals (WIDTH). The role of generic parameters is to simplify the component hierarchy by hiding as all HW elements of a particular type with different timing and wiring characteristics behind a single parameterized class.

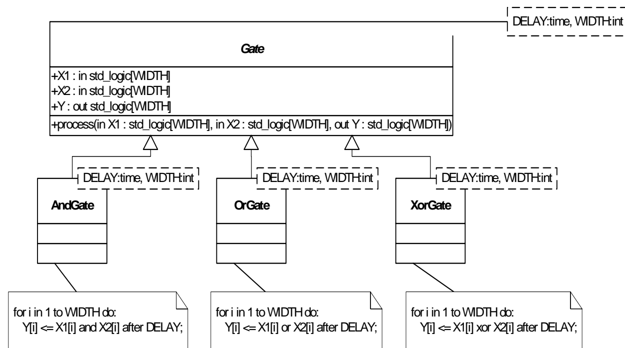

Fig. 4 UML model of the gate schematics

Of course, the UML model of a real-world HW system would be much larger than the one presented in Figure 4, and would include all types of library-based HW components such as gates, registers, adders, counters, etc. Its purpose is to allow modeling of complex HW and HW/SW systems at a level of abstraction that is higher than common programming language code. Generalization here allows hiding the details that are unnecessary at this level of the development of a system, or are continuously repeating from component to component.

Returning back to our design problem, the object-oriented model of a designed system is presented in Figure 5 and explained below (only the top classes are shown). IP is an abstract entity that describes an input interface of soft IP. IP_protocol is an abstract entity that inherits the ports of IP, and declares additional ports for I/O control. IP_handshake is an abstract entity that represents a soft IP communicating using a handshake protocol. Class Model1 is an implementation of IP_handshake that contains HandshakeFSM and an instance of IP. IP_fifo is an abstract entity that represents a soft IP communicating using a FIFO protocol. Class Model1 is an implementation of IP_fifo that contains two FIFO components for storing input and output signals and an instance of IP. FIFO is an abstract entity that describes an interface of a FIFO buffer. FIFO_in is a refinement of FIFO for storing the values of IP's input signals. FIFO_out is a refinement of FIFO for storing the values of IP's output signals. The user can select a communication model to implement by selecting either Model 1 or Model 2 for generation.

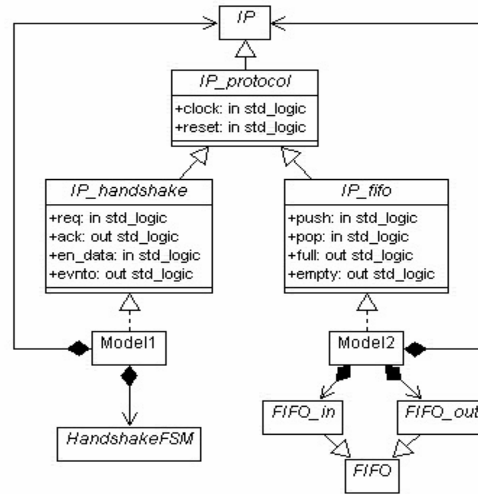

Fig. 5 Simplified UML model of a target system

C. Meta-programming

To illustrate we usage of generalization in the context meta-programming, we deliver a meta-program (Figure 6). Meta-programming is used to implement component hierarchies, especially the parameterized ones, obtained during the modeling stage of system design.

Meta-program (see Figure 6, a) was developed using Open PROMOL [65] as a meta-language and VHDL as a domain language. Open PROMOL is a functional domain-independent meta-language that allows performing text-based modifications of a target program using a set of parameterized functions. The role of the PROMOL functions **@gen** and **@sub** in the meta-program can be easily understood from the context.

```
@- Generic Interface
$
"Select a function:"             {AND,OR,XOR} func:=OR;
"Enter the width of inputs:"     {1..8}       width:=8;
"Enter the delay (in ns):"       {1..10}      delay:=5;
$
@- Gate Interface
ENTITY GATE IS
    PORT (X1, X2: IN STD_LOGIC
       @if[width>1,{_VECTOR(@sub[width-1] DOWNTO 0)}];
          Y: OUT STD_LOGIC
       @if[width>1,{__VECTOR(@sub[width-1] DOWNTO 0)}]);
END GATE;

@- Gate Functionality
ARCHITECTURE MODEL OF GATE IS
    BEGIN
       Y <= X1 @sub[func] X2 AFTER @sub[delay] ns;
END MODEL;                                              (a)
```

```
ENTITY GATE IS
    PORT (X1, X2: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
          Y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END GATE;

ARCHITECTURE MODEL OF GATE IS
    BEGIN
       Y <= X1 OR X2 AFTER 5 ns;
END MODEL;                                              (b)
```

Fig. 6 a) Gate meta-programs, domain language is VHDL and meta-language is Open PROMOL, b) its instance in VHDL

Figure 6, b shows the domain program instance (one of 9 that can be generated from the specifications 1-3) for the type (function) of gate equal to OR, I/O width equal to 8, and delay equal to 5 ns.

The role of generalization here is to simplify the development of soft IP libraries by encapsulating all components with similar functionality under a single generic component, which can be further used for design space exploration and quick generation of a desired component instance using a meta-language processor.

D. Implementation

To validate the described meta-modeling and meta-programming techniques, we have designed the wrapper generator to automatically generate two different wrappers for communication control of third-party soft IP cores using Handshake and FIFO protocols.

The wrapper generator implements a Wrapper design pattern [18] for a specific class of soft IP. The structure of the system is defined using system modeling techniques in UML class diagrams. We use UMLStudio as a front-end tool to draw UML diagrams. The designer develops an UML meta-model and a script for translation from UML to VHDL using a scripting language PragScript that provides straightforward access to the data stored by UMLStudio projects. A PragScript script provides a generic interface to UMLStudio. PragScript interpreter uses UML model (class diagram) and a translation script to generate a structural VHDL model.

Since the structural VHDL model is not enough for a wrapper (class diagrams describe only a structure of the system), and UML class diagrams cannot describe functionality, several meta-programs in Java were developed. These meta-programs capture the behavior of wrapper using Java as a meta-language and VHDL as a domain language. Each meta-program is a Java class, which encapsulates a generic domain entity (e.g., FIFO buffer, voter, etc.). Java processor processes meta-programs and generates specific behavioral VHDL models for a target system using values of the generic parameters specified via a class constructor.

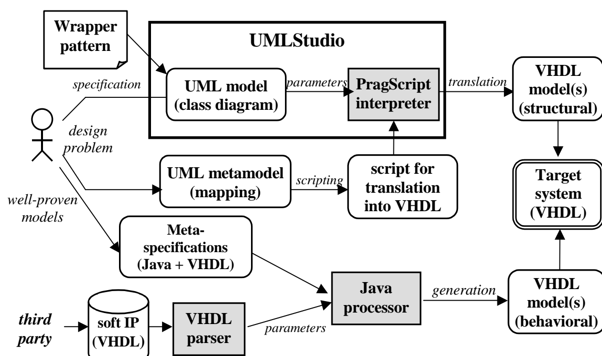

Fig. 7 Implementation of wrapping for well-proven models

The VHDL parser analyses supplied soft IP source code, constructs a syntax tree, and extracts the values of the parameters for generation. The wrapper generator performs wrapping of the third-party soft IP by generating the instances of the component instances that belong to a specified wrapper, and the port map statements to map the signals of the wrapper to the soft IP.

We use two kinds of meta-programs in our design flow (see Figure 7): (1) a script developed using embedded UMLStudio scripting language PragScript, and (2) the meta-programs of behavioral VHDL models developed using an external meta-language (Java). The first meta-program is for describing the structural variability of a component family, while the second one is for representing the behavioral variability.

TABLE I
SYNTHESIS RESULTS (FIFO AND HANDSHAKE COMMUNICATION MODELS)

Soft IP	IP area, cells	Wrapper area, cells (Handshake)	Over-head	Wrapper area, cells (FIFO)	Over-head
Free-6502	4670	471	10 %	2210	47 %
Dragonfly	5883	921	16 %	4568	78 %
AX8	8020	836	10 %	4199	52 %
i8051	24258	1016	4 %	5063	21 %

E. Results

In our experiments, we have used freely available third-party soft IPs as follows: 1) Free-6502 core [38] is a CPU core compatible with 8-bit 6502 microprocessor. 2) DRAGONFLY core [45] is a 8-bit controller that can be used for serial communication management, FLASH and SDRAM control, etc. 3) AX8 core [69] is a 16-bit AT90Sxxxx compatible micro-controller core. 4) i8051 micro-controller [33] is compatible with 8-bit micro-processor designed by Intel.

The wrapper generator was implemented as a set of meta-programs using heterogeneous meta-programming (Java as a meta-language, and VHDL as a domain language). Each meta-program is a Java class, which encapsulates a generic domain entity (e.g., FIFO buffer, FSM, voter, etc.) and generates a specific instance of it in VHDL according to the values of the generic parameters specified via the Java class constructor.

The synthesis results of the original soft IPs and the generated wrappers (Synopsys; CMOS 0.35 um technology) are presented in Table 1. The synthesis results show the following average increase in chip area of the generated wrappers with respect to the original soft IPs: 10% for the Handshake wrapper, and 50% for the FIFO (size = 4) wrapper.

IV. EVALUATION AND DISCUSSION

This paper has analyzed the application of meta-design techniques for HW design domain. We have particularly focused on generalization for developing domain meta-models and meta-programs, and generation for generating customized design solutions automatically. The systematic application of generalization for designing domain systems allows to concisely represent domain content, relationships between domain entities, and to simplify representation of similar

domain entities. Generalization is introduced at different levels of abstraction: for modeling domain models as well as for programming generic domain components.

For modeling, generalization can be used in the context of UML to represent domain component hierarchies, as well as to develop meta-models (design patterns) for describing common solutions to the recurring design problems.

For programming, generalization can be used in the context of product family design to implement generic domain components. Generic components are meta-programs that describe families of the related domain functionality in a narrow and well-defined domain. Generalization is introduced using the meta-programming techniques. This usually involves the usage of two separate languages in one meta-program: a domain language expresses domain content, and a meta-language expresses generalization and variability in a domain.

While the advantages of meta-design are not obvious at the analytical or modeling levels, it is most obvious then meta-models and meta-programs are used to specify and generate domain code. The results of our experiments show that we can generate protocol wrappers for any given soft IP (described in VHDL) automatically. Thus, the main objective of meta-design, i.e. the increase in design productivity, is achieved. Furthermore, wrappers can also be used for wrapping soft IPs that were not know beforehand at design time. Thus, the second aim of meta-design – adaptability for unanticipated requirement and design context changes is achieved. Meta-programming is implemented using Java as a meta-language. The developed meta-programs (program instance generators) are open code, thus allow for evolutionary and collaboration-based design, which is the third aim of meta-design. All used third-party and newly developed tools were integrated into a unified application-oriented design flow.

We summarize meta-design by presenting its main principles:

(1) Meta-design is design for reuse and design for change technology oriented at SW evolution and adaptation to changing requirements and usage context rather than design from scratch.

(2) Meta-design aims at integrating the existing design tools at a higher level into a unified meta-environment and design flow.

(3) Meta-design is based on the usage of the higher-level abstractions in modeling as well as in programming stages of design.

(4) Meta-modeling aims at capturing commonalities and patterns of structure of SW models in a domain.

(5) Meta-programming aims at expressing generalization and capturing variability in domain component space.

(6) The main techniques of meta-design are separation of concerns, generalization and generation. Separation of concerns focuses on identifying, separating and extracting domain commonalities and variabilities. Generalization allows grouping domain commonalities, and parameterizing domain variabilities. Generation allows to automatically instantiate customized source code implementations of a design problem from a higher-level specification.

To implement the principles of meta-design, the traditional system development environments and frameworks are not enough. The latter ones are usually closed and focus on the development of the final product. Meta-design should offer (1) domain-specific languages that exploit the existing user knowledge about the domain and product requirements, (2) provide meta-programming environments that seamlessly integrate the existing programming environments into a unified meta-design flow, (3) exploit the power of sharing and collaboration-based design; and (4) provide support for IP customization, transformation and reuse.

The most important role in meta-design should be played by meta-designers, who set the conditions and provide mechanisms that allow the users to become designers by anticipating both their needs and the potential changes that could occur at use time. Meta-designers should provide the possibility of modifying the system during use time, in order to allow the users to apply the system in the context of usage that was not foreseen at design time.

Finally, meta-design should address the numerous constraints in HW design domain such as timing, power consumption and heat dissipation. The systems must be modeling taking many different, often contradictory, system design aims in account, which requires the adequate support at the modeling and meta-modeling levels. Currently, UML is being adapted for modeling real-time systems by extending it with the timing concept [43]. Power also should be modeled at a higher level [80], which may allow for early estimation and analysis of design characteristics. Such an analysis already at the early stage of design can provide an answer whether the designed system would match the imposed design constraints. Based on the results of the analysis the designer can select the system architecture that can lead to a more efficient implementation.

V. Conclusions

In this paper, we have analyzed the capabilities and application scope of the meta-design techniques. Generalization is perhaps the most important meta-design technique. It deals with the development of generic components (meta-programs) that encapsulate related domain functionality, as well as higher-level domain models and meta-models that allow expressing domain content in an abstract and more general fashion.

Especially, meta-design is important in HW and embedded system design domains, where design complexity continues to grow exponentially. Current trends to applying meta-design in system design focus on meta-programming and meta-modeling. When systematically applied, meta-design provides means for achieving higher reuse (through generalization), increase design quality and productivity (through generation), and allow integration with other design methodologies such as object-oriented design (through the usage of UML).

REFERENCES

[1] J.F. Agaësse and B. Laurent, "Virtual components application and customization", in: *Proc. of Design, Automation and Test in Europe (DATE 99)*, Munich, Germany, March 9-12, 1999, 726-727.

[2] C. Alexander, *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[3] S.W. Ambler, *Technique Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press, 1998.

[4] C. Atkinson and T. Kühne, "The Role of Metamodeling in MDA", *International Workshop in Software Model Engineering (in conjunction with UML '02)*, Dresden, Germany, October 2002.

[5] L. Baker, P. Clemente, B. Cohen, L. Permenter, B. Purves, and P. Salmon, "Foundational concepts for model driven system design", Technical paper, Vitech Corporation, 1997.

[6] R.R. Barton, "Metamodeling: a state of the art review", in: *Proc. of the 1994 Winter Simulation Conference*, Lake Buena Vista, FL, USA, 1994, 237-244.

[7] D. Batory, "Product-line architectures", *Smalltalk and Java in Industry and Practical Training*, Erfurt, Germany, 1998, 1-12.

[8] D. Batory, R.E. Lopez-Herrejon, J.-P. Martin, "Generating product-lines of product families", in: *17th IEEE Conference on Automated Software Engineering (ASE 2002)*, 23-27 September, 2002, Edinburgh, Scotland, UK, 81-92.

[9] D. Batory, S. Dasari, B., Geraci, V. Singhal, M. Sirkin, and J. Thomas, "Achieving reuse with software system generators", *IEEE Software*, September 1995, 89-94.

[10] M. Becker, "Generic components: a symbiosis of paradigms", in: G. Butler and S. Jarzabek (Eds.), *Generative and Component-Based Software Engineering, 2nd Int. Symposium, GCSE 2000*, Erfurt, Germany, October 9-12, 2000, LNCS 2177, Springer, 100-113.

[11] J. Bezivin, N. Farcet, J.-M. Jezequel, B. Langlois, and D. Pollet, "Reflective model driven engineering", in: P. Stevens, J. Whittle, and G. Booch (eds.), *Proc. of the 6th Int Conference on The Unified Modeling Language – Modeling Languages and Applications (UML 2003)*, October 20-22, 2003, San Francisco, CA, USA, Lecture Notes in Computer Science, 2863, Springer, 175-189.

[12] G. Booch, I. Jacobson, J. Rumbaugh, and J. Rumbaugh, *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[13] L.P. Carloni, F. De Bernardinis, A. Sangiovanni-Vincentelli, and M. Sgroi, "The art and science of integrated systems design", in: *Proc. of 28th European Solid-State Circuits Conference (ESSCIRC 2002)*, 2002, Florence, Italy, 25-36.

[14] J.K. Cross and D.C. Schmidt, "Meta-programming techniques for distributed real-time and embedded systems", in: *Proc. of 7th IEEE Int. Workshop on Object-Oriented Real-Time Dependable Systems*, January 7-9, 2002, San Diego, CA, USA, 3-10.

[15] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2001.

[16] K. Czarnecki and U.W. Eisenecker, "Separating the configuration aspect to support architecture evolution", in: *Proc. of 14th European Conference on Object-Oriented Programming (ECOOP'2000), Int. Workshop on Aspects and Dimensions of Concerns*, Cannes, France, June 11-12, 2000.

[17] R. Damaševičius and V. Štuikys, "Wrapping of soft IPs for interface-based design using heterogeneous metaprogramming". *INFORMATICA*, 14 (1), 3-18, Lithuanian Academy of Sciences, Vilnius, 2003.

[18] R. Damaševičius, G. Majauskas, and V. Štuikys, "Application of design patterns for hardware design", in: *Proc. of 40th Design Automation Conference (DAC 2003)*, 2-6 June, 2003, Anaheim, CA, USA, 48-53.

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

[20] S. Djokic, G. Succi, W. Pedrycz, and M. Mintchev, "Meta Analysis – a Method of Combining Empirical Results and its Application in Object-Oriented Software Systems", *Proc. of the 7th Int. Conference on Object-Oriented Information Systems*, Calgary, Alberta, August 2001.

[21] P. Donohoe (Ed.), *Software Product Lines: Experience and Research Directions*, Kluwer Academic Publisher, Boston, 2000.

[22] F. Doucet and R.K. Gupta, "Microelectronic System-on-Chip modeling using objects and their relationships", in: *Online Symposium for Electrical Engineers (OSEE 2000).*

[23] B.P. Douglass, "Fine grained patterns for real-time systems", in: L. Lavagno, G. Martin, and B. Selic (eds.), *UML for Real*, 149-170. Kluwer Academic Publishers, Boston, 2003.

[24] C. Dumoulin, P. Boulet, J. Dekeyser, and P. Marquet, "MDA for SoC Design, intensive signal processing experiment", in: *Forum on Design Languages (FDL'03)*, Frankfurt am Main, Germany, 2003.

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

[26] A. Evans, R. France, K. Lano, and B. Rumpe, "Meta-modeling Semantics of UML", in: H. Kilov (ed.), *Behavioural Specifications for Businesses and Systems*. Kluwer Academic Publishers, 1999.

[27] U. Fayyad, G.G. Grinstein, and A. Wierse (eds.), "Information Visualization in Data Mining and Knowledge Discovery", in: U. Fayyad, G.G. Grinstein, and A. Wierse (eds.). *Information Visualization in Data Mining and Knowledge Discovery.* Morgan Kaufman, London/San Francisco, 2002.

[28] J.M. Fernandes, R.J. Machado, and H.D. Santos, "Modeling industrial embedded systems with UML", in: *Proc. of 8th IEEE/IFIP/ACM Int. Workshop on Hardware/Software Co-Design (CODES'2000)*, 2000, San Diego, CA, USA, 18-23.

[29] G. Fischer and E. Giaccardi, "Meta-Design: A Framework for the Future of End-User Development", in Lieberman, H., Paternò, F., and Wulf, V. (Eds), *End User Development - Empowering People to Flexibly Employ Advanced Information and Communication Technology*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[30] G. Fischer and E. Scharff, "Meta-Design—Design for Designers", *Proc. of 3rd Int. Conference on Designing Interactive Systems (DIS 2000)*, New York, pp. 396-405.

[31] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[32] J.P. van Gigch, *System Design Modeling and Metamodeling*, Plenum Press, New York, 1991.

[33] T. Givargis, 2000. Intel 8051 micro-controller, http://www.cs.ucr.edu/~dalton/i8051/i8051syn/

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

[35] D. Kafura, *Object-Oriented Software Design and Construction with C++*, Prentice Hall, 1997.

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

[37] A.S. Karrer and W. Scacchi, "Meta-Environments For Software Production", *International Journal of Software Engineering and Knowledge Engineering,* Vol. 3(1), 1993, pp. 139-162.

[38] D. Kessner, 1999. Free-6502 core, http://www.free-ip.com/6502/

[39] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli, "System level de-sign: orthogonalization of concerns and platform-based design", *IEEE Trans. on CAD of ICs and Systems*, 19 (12), 2000, 1523-1543.

[40] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming", *Proc. of the European Conference on Object-Oriented Programming (ECOOP'1997).* Lecture Notes in Computer Science, 1241, Springer-Verlag, 220-242.

[41] H. Kühn, and M. Murzek, "Interoperability Issues in Metamodelling Platforms", in: *Proc. of the 1st Int. Conf. on Interoperability of Enterprise Software and Applications (INTEROP-ESA'05)*, Geneva, Switzerland, February 2005, Springer Verlag.

[42] J. de Lara, H. Vangheluwe, and M. Alfonseca, "Meta-modelling and graph grammars for multi-paradigm modelling in AtoM", *Software and Systems Modeling* 3(3), Aug 2004, pp. 194–209.

[43] L. Lavagno, G. Martin, and B. Selic, *UML for Real*. Kluwer Academic Publishers, Boston, 2003.

[44] E.A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation", *IEEE Transactions on CAD*, 17 (12), 1998, 1217-1229.

[45] LEOX Team, 2001. DRAGONFLY micro-core, http://www.leox.org

[46] B. Liccardi, T. Maier-Komor, J.A. Oswald, M. Elkotob, and G. Färber, "A meta-modeling concept for embedded RT-systems design", in: *14th Euromicro Conference on Real-Time Systems*, 19-21 June, 2002, Vienna, Austria.

[47] W. Löwe and M. Noga, “Metaprogramming applied to web component deployment”, *Electronic Notes in Theoretical Computer Science*, 65(4), 2002.

[48] A. Ludwig and D. Heuzerouth, “Metaprogramming in the large”, in: G. Butler and S. Jarzabek (Eds.), *Generative and Component-Based Software Engineering*. Lecture Notes in Computer Science, 2177, 178-187. Springer, 2001.

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

[50] S. Meguerdichian, F. Koushanfar, A. Mogre, D. Petranovic, and M. Potkonjak, “MetaCores: design and optimization techniques”, in: *Proc. of Design Automation Conference (DAC’2001)*, June 18-22, 2001, Las Vegas, Nevada, USA, 585-590.

[51] A. Mihal, C. Kulkarni, C. Sauer, K. Vissers, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, S. Malik, “A Disciplined Approach to the Development of Architectural Platforms”. *IEEE Design and Test of Computers*, 19, 2-12, 2002.

[52] D. de Niz and R. Rajkumar, “Model-based embedded real-time software development”, in: *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003), Workshop on Model-Driven Embedded Systems (MDES 2003)*, May 27-30, 2003, Washington DC, USA.

[53] Object Management Group (OMG), 2001. *Model-Driven Architecture: A Technical Perspective*. Technical Document.

[54] H. Ossher and P. Tarr, “Multi-Dimentional Separation of Concerns and the Hyperspace Approach”, in: *Software Architectures and Component Technology: The State of the Art in Software Development,* M. Aksit, Ed., Kluwer Academic Publishers, Boston, 2001.

[55] M.J. Pont and M.P. Banner, “Designing embedded systems using patterns: a case study”, *Journal of Systems and Software*, 71(3), 201-213, 2004.

[56] A. Rajawat, M. Balakrishnan, and A. Kumar, “Interface synthesis: issues and approaches”, in: *Proc. of the 13th Int. Conference on VLSI Design*, January 3-7, 2000, Calcutta, India, 92-97.

[57] D. Riehle and H. Zellighoven, “Understanding and using patterns in software development”, *Theory and Practice of Object Systems*, 2 (1), 3-13, 1996.

[58] J. Rowson and A. Sangiovanni-Vincentelli, “Interface-based design”, in: *Proc. of the 34th Design Automation Conference (DAC 97)*, June 9-13, 1997, Anaheim, CA, USA, 178-183.

[59] A. Ryman, “Requirements for a metaprogramming language”, *Presentation at the 24th meeting of IFIP Working Group 2.4*, Kingston, Canada, 1990.

[60] A. Sangiovanni-Vincentelli and G. Martin, “A vision for embedded systems: platform-based design and software methodology”, *IEEE Design and Test of Computers,* 18 (6), 23-33, 2001.

[61] B. Selic, “Architectural patterns for real-time systems”, in: L. Lavagno, G. Martin, and B. Selic (eds.), *UML for Real*, 171-188. Kluwer Academic Publishers, Boston, 2003.

[62] Semiconductor Industry Association, *The International Technology Roadmap for Semiconductors*, 2001.

[63] 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. Lecture Notes in Computer Science, 2196, 2-44, 2001.

[64] V. Štuikys and R. Damaševičius, “Metaprogramming techniques for designing embedded components for ambient intelligence”, in T. Basten, M. Geilen, and H. de Groot (eds.), *Ambient Intelligence: Impact on Embedded System Design*. Kluwer Academic Publishers, Boston, 2003, pp. 229-250.

[65] V. Štuikys, R. Damaševičius, and G. Ziberkas, “Open PROMOL: An Experimental Language for Target Program Modification”, in: A. Mignotte, E. Villar, and L. Horobin (eds.), *System on Chip Design Languages*. Kluwer Academic Publishers, 2002.

[66] P.D. Terry, *Compilers and Compiler Generators: An Introduction with C++*. International Thomson Computer Press, 1997.

[67] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, 2002.

[68] T.L. Veldhuizen, “Using C++ template metaprograms”. *C++ Report* 7(4), 36-43, 1995.

[69] D. Wallner, AX8 core, 2001, http://hem.passagen.se/dwallner/vhdl.html

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

[71] N. Yoshida, “Design patterns applied to object-oriented SoC design”, *Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI 2001)*, 18-19 October, 2001, Nara, Japan.

[72] J. Zhu, “MetaRTL: raising the abstraction level of RTL design”, in: *Proc. Design Automation and Test in Europe (DATE 2001)*, March 13-16, 2001, Munich, Germany, 71-76.

[73] Q. Zhu, A. Matsuda, S. Kuwamura, T. Nakata, and M. Shoji, “An object-oriented design technique for System-on-Chip using UML”, in: *Proc. of the 15th Int. Symposium on System Synthesis (ISSS 2002)*, 1-4 October, 2002, Kyoto, Japan, 249-254.

[74] J.W. Janneck and R. Esser, “Higher-order modeling and automated design-space exploration”, in: *Proceedings High-Performance Computing (HPC) 2002*.

[75] M. Gries, “Methods for Evaluating and Covering the Design Space during Early Design Development”. *Integration, the VLSI Journal,* Elsevier, 38(2):131-183, December 2004.

[76] R. Damaševičius and V. Štuikys, “Application of the Object-Oriented Principles for Hardware and Embedded System Design”. *Integration, the VLSI Journal*, 2004, 38(2), pp. 309-339. Elsevier.

[77] M. Palesi and T. Givargis, “Multi-Objective Design Space Exploration Using Genetic Algorithms”, in: *International Workshop on Hardware/Software Codesign (CODES),* Estes Park, May 2002.

[78] G. Palermo, C. Silvano and V. Zaccari, “Multi-Objective Design Space Exploration of Embedded Systems”. *Journal Of Embedded Computing*, Vol. 1, No. 11, November 2002.

[79] D. Ikeda, S. Hirokawa and Y. Yamada, “Pattern Discovery of Genome Sequences by Substring Amplification”, in: *Proc. of Int. Symposium on Information Science and Electrical Engineering*, pp. 637-640, November, 2003.

[80] R. Damaševičius and V. Štuikys, “Estimation of Power Consumption at Register-Transfer and Behavioral Modeling Levels Using SystemC”. Submitted to *Journal on Low Power Electronics (JOLPE).*