ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2007, Vol. 36, No. 1

VARIABILITY–ORIENTED EMBEDDED COMPONENT DESIGN FOR AMBIENT INTELLIGENCE

Vytautas Štuikys, Robertas Damaševičius

Software Engineering Department, Kaunas University of Technology
Studentų St. 50, LT–51368 Kaunas, Lithuania

Abstract. Ambient Intelligence is a new vision of future digital environments characterized by ubiquity, transparency and intelligence. The user is surrounded by embedded systems that are invisible, context-aware, personalized and adaptable to the user requirements. Design of Ambient Intelligence systems is, essentially, design of sophisticated and interconnected embedded systems that operate within a common human-oriented environment. Such embedded systems contain a variety of embedded components with different functionality, characteristics and requirements. Embedded components are common hardware/software components that are basic blocks for building embedded systems and have a great deal of variability. This paper focuses on embedded component design for Ambient Intelligence systems and proposes a design framework based on the systematic domain analysis methods, well-proven domain models, well-documented design processes, UML-based object-oriented specification, meta-programming-based representation of variability within generic embedded components, and automatic domain code generation. We demonstrate validity of our approach for two domains of application: communication control and fault-tolerance.

Keywords: Ambient Intelligence, code generation, domain analysis, embedded system, generalization, metaprogramming, object-oriented specification, variability.

1. Introduction

Ambient Intelligence (AmI) is a vision for future communication and human-machine interaction systems [Riva, 03]. Common features of such systems are ambient computing, omnipresent communication to support context-awareness and intelligent interfacing [Ducatel, 01]. The domain is not well understood yet, regardless of a wide stream of research on the topic [Boekhorst, 02] [Basten, 03a] [Basten, 03b] [Lindwer, 03] [Weber, 03] [Remagnino, 03] [Cai, 05].

From the pure structural perspective, an AmI-oriented environment can be treated as a specifically organized collection of integrated *embedded systems* (ES) satisfying requirements and constrains of AmI-oriented design. They are as follows: higher diversity and complexity of systems and components, increased quality, productivity and reuse content, standardization, stricter requirements for time-to-market and fault-tolerance, design for variation and low power [Basten, 03b]. These factors should be considered in the context of underlying domain technology, which is further scaling down and causes exponential complexity growth of the designed systems [ITRS, 03]. The outcome is that systems of yesterday become components of today. Furthermore, the blurring boundaries between hardware (HW) and software (SW) design [Eggermont, 02] [Vahid, 03] for a long perspective requires introduction of the higher-level abstractions in the design process [ITRS, 03].

The researchers can respond to this challenge by either improving the currently used design methodologies or creating the new ones. New AmI-oriented systems must be based on the fundamental principle stating that *any system consists of components*. This principle is common for any technical system as well as for a mature engineering discipline, and it is sometimes called "*a law of nature*" [Szyperski, 99]. The design methodologies for AmI systems must exploit adequately this principle, too.

ES are used as subsystems in a variety of smart products such as mobile phones, DVD players, and kitchen appliances. These systems implement a large diversity of functions; however, they are composed of a limited number of common SW/HW components such as DSP, MPEG, codecs, etc. We call these basic design blocks *embedded components*, the nodes for the future AmI systems. We use this term as a generic name for IP (*Intellectual Property*) components (IPs), (embedded) SW components, HW components (soft IPs), and SW/HW subsystems. However, this term should be also treated as an abstraction covering various forms of representation such as a generic specification and metamodel of reusable components.

Since there is a great variety of embedded components, which are required for the design of AmI

systems, reuse and variability management of component assets has become increasingly important in embedded system design. The designers are shifting their focus from designing separate application-specific domain systems to developing generic components [Becker, 01], platforms [Mihal, 02], or entire product lines [Diaz-Herrera, 00], which implement common functionality of component (system) family that satisfies the specific needs of a particular market segment, and provide variability management mechanisms for instantiating the specific component (instance) customized for particular performance and application requirements. Such product lines could be successfully used for developing embedded components on an industrial scale to match huge customer demand and varying requirements, as well as maintain quality-of-service and shorten time-to-market.

The key to successful design for variability is systematic management of domain variations, while exploiting the commonalities. These commonalities permit reuse of shared assets, such as architectures, reusable components, test cases and documentation. When a new AmI system is developed, the emphasis should be placed on automatic integration of existing embedded components or generation of customized components rather than programming. Domain analysis and knowledge mining for extraction of anticipated variability, development of methods and abstractions for facilitating and improving variability management mechanisms, and creation of tools for implementation of variability and generation of customized ready-to-use components are essential to design success.

Our contribution is (1) a general variability-oriented design framework for developing generic embedded components, (2) adoption of high-level abstractions such as UML class diagrams and metaprograms for specification of embedded components, management of design variability and implementation of domain code generation for well-understood domains.

The outline of this paper is as follows. Section 2 describes the concept of Embedded Component, its representation forms and design processes. Section 3 describes the Embedded Component design methodology in detail. Section 4 presents our experiments. Section 5 presents evaluation of results and conclusions.

2. Embedded Component Design Framework

2.1. Embedded Component Concept, its Representation Forms, and Model

The designers use a broad variety of models in the ES domain [Selic, 03b]. For instance, *models of computation* (MoC) are formal and abstract definitions of a component [Edwards, 97] [Lee, 98]. Examples of MoC are finite state machine (FSM), Boolean circuit, Petri nets, etc. MoC allow analyzing the intrinsic properties of a component such as execution time or memory space of an algorithm while ignoring many implementations issues. The design process is iterative – a design is transformed from an informal description into a detailed specification usable for manufacturing. Multiple MoC are needed to express the heterogeneous nature of most ES.

Component models [Agaësse, 97] [Haase, 99] [Meguerdichian, 01] [Nitsch, 03] [Siegmund, 00] [Štuikys, 02] [Vermeulen, 00] [Zhu, 01] 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 IP reuse, or enable convenient IP retrieval and sharing. The design process focuses on design space exploration, parameterization, and generation of specific components (IPs). The proposed solutions are usually language-centric (pre-processing, extensions of languages, etc.).

Architectural models such as *platforms* [Mihal, 02] [Sangiovanni-Vincentelli, 01] address system-level design based on IP reuse. Platforms are common architectures based on principal components that remain fixed within a certain degree of parameterization. Such platforms can support a variety of applications in a given application domain. Platforms emphasize not the design of functionality, but the communication-based design independent of the behaviour of particular components. The design process focuses on refinement of a platform for a specific application.

We introduce a concept of Embedded Component in the context of the AmI-oriented design as follows. *Embedded Component* is a design abstraction common to SW/HW systems. It has three representation forms: *generic specification (GS), reusable instance (RI)* and *embeddable instance (EI)* (see Figure 1). These forms correspond to three hierarchic abstraction layers as follows. Generic specification is at the *generic* layer of abstraction (the highest one), reusable instance is at the *reuse* layer (the middle one), and embeddable instance is at the *embedding* layer (the lowest one). A higher abstraction layer is refined into a lower one using a well-defined *design process*. Below, we describe these representation forms of Embedded Component in more detail as follows.

Generic specification is parameterized description of a family of related component instances. A family includes several (from dozens to hundreds or even thousands) reusable instances that differ in functionality and requirements. The environment for a generic specification is a generic library of IP providers or large design organizations.

Reusable instance is a particular instance generated from a generic specification that can be reused in several applications. Usually it is not specific enough to be embedded into a particular target system and needs to be adapted to its environment of application. The environment for a reusable instance is either

libraries of IP providers, IP exchanges, or libraries of design teams.

Embeddable instance is a component instance that was adapted to a particular context of application and, generally, is not reusable. The environment for an embeddable instance is an ES as a part of the AmI system. An embeddable instance can be implemented in a real system either (1) "use-as-is" without any modification, or (2) with customization/modification of a reusable instance.

A *Generic Embedded Component Model* (see Figure 2) describes the structure of a generic specification. A *metainterface* represents the generic parameters at a higher level of abstraction and hides the families of reusable instances and details of their implementation from the user. The generic interface and generic functionality represent the parameterized interface and functionality of reusable instances, respectively. The reusable instances are derived from a generic specification automatically using transformation generation tools.

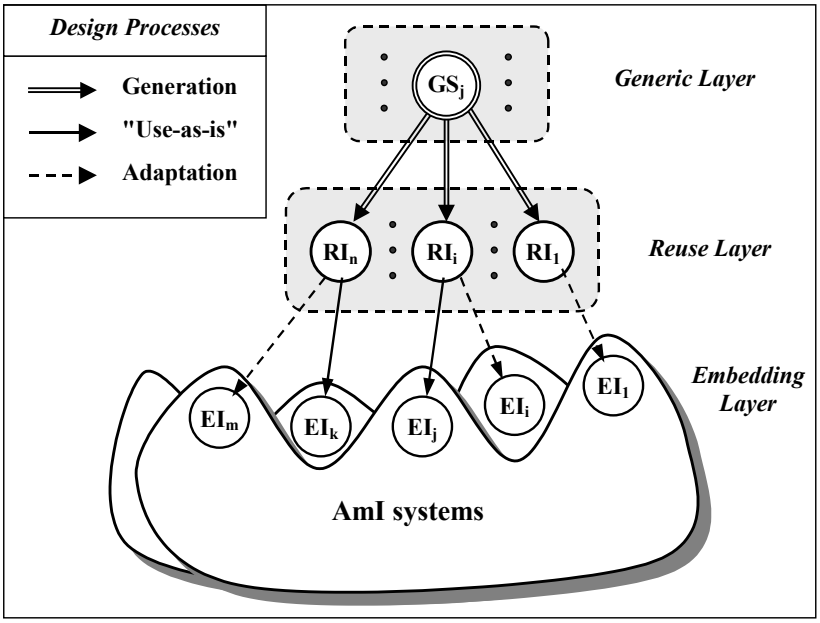

Figure 1. Relationship between Embedded Component forms

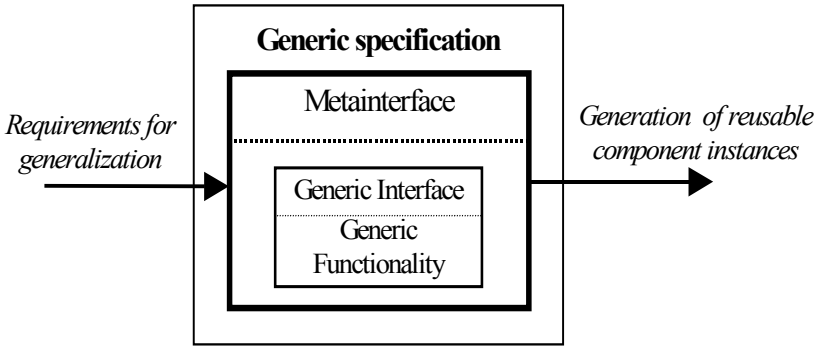

Figure 2. Generic Embedded Component Model

The Generic Embedded Component Model supports:

1) *Domain-independence* – the model is common for HW, SW, and embedded SW domains [Štuikys, 03]. Here we have in mind the representation of components only, but not the semantic aspects.
2) *Design automation* – a generic specification is a generic form representing a family of the related reusable instances, and a *generation/transformation process* can be defined, supported and executed to obtain the reusable instances automatically.
3) *Common methodological background* for transforming, customizing and delivering of Embedded Components.

2.2. Design processes

Design processes are a very important part of the suggested framework. We describe these processes in more detail latter. Now, we focus on the formulation of the pre-conditions of the processes and present the framework in the whole (see Figure 3).

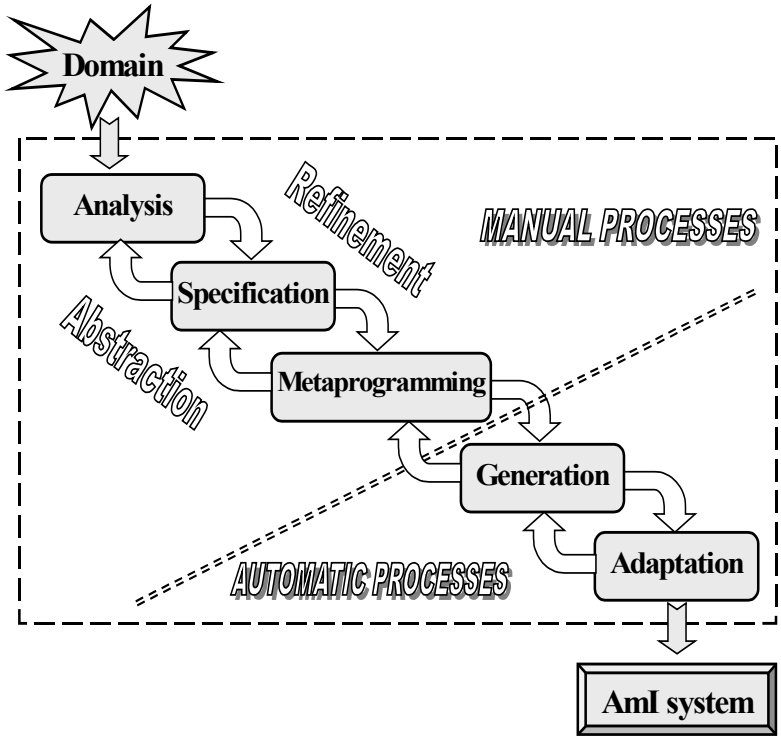

Figure 3. Embedded Component design processes for AmI

(1) *Analysis* process comprises the consideration of roadmaps for ES and AmI domains [ITRS, 03] [Eggermont, 02], including requirements for AmI, and systematic analysis of experimental AmI systems, HW and SW systems.

(2) *Specification* process is applied for describing Embedded Components at a higher-level of abstraction using object-oriented approach based on UML diagrams [Booch, 98] and design patterns [Gamma, 95].

(3) *Design* process includes development of generic representations of Embedded Components (i.e., generic specifications) using the *metaprogramming* techniques [Štuikys, 02] in order to manage the design variability in AmI domain.

(4) *Generation* process is automatic creation of reusable Embedded Component instances according to the pre-specified requirements of a user or application.

(5) *Adaptation* is a process of adapting a reusable instance and integrating it into a real AmI system. As the context of such a system may be unknown for an Embedded Component designer, we do not consider this process in the paper.

3. Design Methodology in Details

3.1. Well-Understood Domains and AmI

SW engineering considers "domain" as its basic theme of exploration [Jacobson, 97]. The topic is so important and widely discussed that a new discipline, *Domain Engineering*, has emerged [Diaz-Herrera, 01] [Harsu, 02]. A domain is an environment from which SW engineers extract knowledge required for designing a system. In the context of reuse, domains can be categorized as *vertical* (narrow) and *horizontal* (broad) ones. From the evolutionary perspective, domains can be treated either as poorly or well-understood ones. AmI covers both vertical and horizontal domains and, as a topic for research and creating the future advanced systems, it is not well-understood in many aspects yet. However, design of an AmI system relates to the variety of sub-domains (HW, SW, and ES design) that are already well-understood.

To support the basic constrains and requirements (reliability, functionality, variability for AmI systems, etc.), the designers should pay an increased attention to the well-understood domains in the first place. It is particular important for designing Embedded Components because the main intention of introducing them is to ensure a high reuse context [ITRS, 03] [Eggermont, 02]. There is no precise definition in the literature what a well-understood domain or sub-domain is. The term "well-understood domain" should be conceived here as a domain that contains well-proven model(s). There are many well-proven models in both HW and SW domains. Examples selected below refer to the aspects of componentry in HW and SW design and experiments we have carried out.

1) *Communication models* such as handshake [Berkel, 94] and FIFO [Gajski, 97] protocols are models for describing aspects of communication between HW components given in a high-level HW description language (HDL), such as VHDL, Verilog or SystemC.

2) *Triple Redundancy Model* (TRM) is used in the reliability-critical applications to mask HW faults. TRM implements the two-out-of-three voting concept, as well as the erroneous output indication (if needed). There are Space, Data and Time Redundancy sub-models of TRM [Entrena, 01] [Fuhrman, 95] [Pflanz, 98] as follows: a) *Space* redundancy is based on the addition of extra hardware in order to perform simultaneously the same operations of a system. b) *Time* redundancy is based on the usage of additional time to perform system functions, thus achieving soft error tolerance. c) *Data* redundancy is based on the addition of redundant data (auxiliary data path, error-detecting and correcting codes) to ensure reliability during transfer of data via system interconnections.
3) *Wrapper models* are widely used in HW design, e.g.: a) *Bus wrapper* provides an implementation of a particular data protocol for communication with other components. This solution is used in *Virtual Socket Interface* methodology [Cyr, 01] to connect the IPs to on-chip buses in System-on-Chip (SoC) designs. b) *Protocol wrapper* provides an implementation of an OSI protocol layer. This solution is used in the FPX networking platform [Braun, 02] to implement high-level networking functions by abstracting the operation of the lower-level packet processing functions. c) *Memory wrapper* adapts the physical memory interfaces to a communication network that may have a different number of access ports. This approach is used in [Gharsalli, 02] to facilitate the integration of the standard memory components into SoC designs.

3.2. General Domain Analysis Methods for Design for Variability

The more complex system/component has to be designed, the higher is the role of domain analysis. Domain analysis methods can be categorized as *ad hoc* and *systematic* ones. The first ones prevail in current design practice, especially in HW and ES design. The role of systematic methods increases alongside with the growth of design complexity. We expect that they will become extremely important in design of future AmI systems. Thus, we present below a brief survey of the *systematic* domain analysis methods that can be used to design the Embedded Components for AmI systems.

Multi-Dimensional Separation of Concerns [Ossher, 01] claims that *design concerns* can be understood in terms of an *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 exists at different layers of abstraction such as in ES design.

Feature-Oriented Domain Analysis [Kang, 90] [Prieto-Diaz, 91] deals with analysis and documentation of distinctive *features* of SW systems. Features are domain characteristics that define both common domain aspects as well as differences between related domain systems. The underlying concepts are: *aggregation* (composition of separated concerns into a generic component), *decomposition* (abstraction and isolation of domain commonalties and variations), *parameterization* (substitution of the application-specific concerns with values of component parameters), *generalization* (capturing domain commonalties, while expressing the variations at a higher level of abstraction), and *specialization* (tailoring a generic component into a specific component that incorporates the application-specific details).

Family-oriented Abstraction, Specification and Translation [Weiss, 99] method focuses on grouping similar entities or concepts into *families*. A family is a collection of system parts that are treated together because they are more alike than different. Grouping is based on *commonalties* that fall along many dimensions (structure, algorithm, etc.). Commonality defines the shared context that is invariant across abstractions of the application. The individual family members (instances) are distinguished by their differences, called *variabilities*. Variability captures the properties that distinguish abstractions in a domain. The aim of the method is to uncover variability and provide means for its implementation.

Conceptually, the analyzed methods have many in common. First, they emphasize decomposition of a design space into smaller dimensions related to the specific design concerns. Secondly, they describe management of variability and commonality in a domain and, finally, integration of similar entities (features) at a higher layer of abstraction. From that perspective, any of the analyzed methods can be used for designing Embedded Components. The selection depends upon the skills of a designer and the available tools.

We summarize our analysis as follows. We assume that SW/HW components designed for AmI systems have *variant* and *invariant* parts. These parts represent variability and commonality in the domain. The primary goal of using domain analysis methods is to recognize the parts, and express them in a suitable form for further refinement. At a higher level, we use domain concerns alongside with the domain-oriented abstractions to describe the variant and invariant parts more precisely. Finally, we formulate the issues of domain analysis either explicitly or implicitly. The *explicit* artifacts are *taxonomy* of domain objects and their features (requirements, parameters), methods, processes and models for ES design. The *implicit* artifacts are domain knowledge in the form of a

conceptual model used for further refinement of the obtained domain artifacts and models.

3.3. Domain-Specific Analysis Methods

The domain-specific analysis methods mostly focus on *design space exploration*, i.e., analyzing and searching the domain for the optimal design solutions (in terms of power, area, delay, etc.). Design space exploration tasks today often deal with high-level synthesis problems, such as automated resource allocation, binding of computation and communication to resources, and scheduling.

As complexity of the design systems increases, it is becoming more and more unlikely that a designer will find an optimal design solution using his experience, domain knowledge and prior design decisions as a basis only. A disciplined approach to design space exploration is inevitably needed in order to evaluate ever-increasing design spaces. Here, we analyze an approach for the automated and systematic design space exploration widely known as Y-Chart [Kienhuis, 97] [Gerstlauer, 02].

The Y-chart identifies three sub-domains: (1) Functional sub-domain: functional components, algorithms, etc. (2) Structural sub-domain: processors, memories, busses, etc. (3) Physical sub-domain: hardware resources, delays, constraints, etc.

The Y-Chart also defines system, register-transfer, gate, and transistor levels where each level is defined by the type of objects. The higher-level objects are hierarchically composed of the lower-level ones. At each level, the design can be described in the form of a behavioral, a structural model, or a physical model as follows.

(1) *Behavioral model* describes the desired functionality as a composition of abstract functional entities that get activated, process input data, produce output data, and terminate.
(2) *Structural model* describes the net-list of physical components and their connectivity. Structural objects represent real components and wires that are processing the data.
(3) *Physical model* describes the physical placement of the sub-components on the chip.

The design activities begin at the highest-level sub-domain that corresponds to the highest-level of abstraction in the domain. Then, a successive refinement process between each sub-domains is applied according to various abstraction levels. The design process ends at the lowest level of abstraction in physical sub-domain.

In the Y-Chart, system design is the process of moving from a behavioral description to a structural description under a set of constraints where the structural objects are each designed at the next lower level.

At the system level, system design is the process of deriving a structural description of the system and the system architecture from a behavioral system specification. Behavioral objects at the system level are general functions and algorithms that communicate by transferring data through global variables. Structural objects are processing elements, e.g. general-purpose processors, custom hardware, components, and memories that communicate via buses. For each design task, the models at the input and output of the flow have to be defined such that the transformation between the models becomes possible.

An application and architecture are modeled separately and explicitly mapped onto each other. Next, a performance analysis for alternative application instances, architecture instances and mappings has to be done, thereby exploring the design space of the target system.

3.4. UML-Based Specification

The Object-oriented design (OOD) methodology is based on the concept of using high-level models organized around real world concepts. This approach has actually become the *de-facto* standard for SW design. Objects are usually modeled using UML [Booch, 98]. Recently, UML has gained acceptance in HW and ES design community, too [Jong, 02] [Martin, 02] [Chen, 03] [Edwards, 03] [Goudarzi, 04]. Advantages of using UML for OO HW design are as follows: 1) high level specification of a designed system, 2) better soft IP reusability and adaptability, 3) better documentation for further reuse and maintenance of a system.

Design patterns are another abstraction for representing common design solutions in UML notation [Gamma, 95]. Design patterns are used to abstract and encapsulate common design solutions as well as to describe contexts to which they can be applied in an implementation-independent way. They originated in SW domain for creating the SW systems using previous successful design experience. However, recently there are evident signs and efforts for adapting them for HW design [Yoshida, 01] [Åström, 01] [Doucet, 02] [Damaševičius, 03] [Selic, 03a] [Damaševičius, 04a, b].

In the context of Embedded Component design for AmI systems, the OO specification of a system using UML is extremely important due to the following reasons. 1) Describing a system in an abstract and implementation-independent way significantly raises the level of abstraction. 2) Using the standard UML diagrams eases the communication between different design teams. 3) Using the already verified design solutions ensures a higher design quality. 4) Using graphical design tools, catalogues of design patterns, and automatic code generation tools can increase design productivity as well as accelerate design reuse, sharing and transfer.

Additionally, the usage of design patterns may reduce the gap between the development of SW and HW parts, as the OO and pattern-based design is widely used in SW domain. It can be very useful to co-design the HW and SW parts of a system using the same

design methodology, and partition these parts as late as possible in the design cycle. The same high-level description can be implemented either in HW, or in SW running on an embedded processor. This allows achieving greater flexibility for the system designer.

However, in order to exploit the full potential of design patterns in HW design domain, much work still is to be done. As has been shown in [Yoshida, 01] only a few SW design patterns can be introduced in HW design without changes. The others require a certain degree of adaptation, and the rest ones from currently known list of 23 main SW design patterns [Gamma, 95] are not adopted at all, yet.

3.5. Metaprogramming

Several related efforts in the area are *aspect-oriented programming* [Kiczales, 97], *generative programming* [Czarnecki, 00] [Sztipanovits, 02], and *generic programming* [Gibbons, 03]. *Metaprogramming* is a kind of higher-level programming the interest and attention for which is constantly growing [Sheard, 01]. It can be applied in different domains and in the various contexts where the *multi-language design* paradigm [Kleinjohann, 98] [Jerraya, 99] is used. In this context, we consider metaprogramming as a design technology for managing variability and implementing domain code generation [Štuikys, 02].

The main aim of metaprogramming is to create a *metaspecification* – a program generator for a narrow domain of application. Conceptually, a metaspecification is based on the generic embedded component model (Figure 2). Whereas structurally, a metaspecification (see Figure 4) consists of a generic interface, related domain program instances and a modification algorithm that describes generation of a particular instance depending upon values of the generic parameters. The modification algorithm can include *meta-if* (conditional generation) and *meta-for* (repetitive generation) constructs as well as sophisticated application-specific patterns composed of nested combinations of simpler metaconstructs.

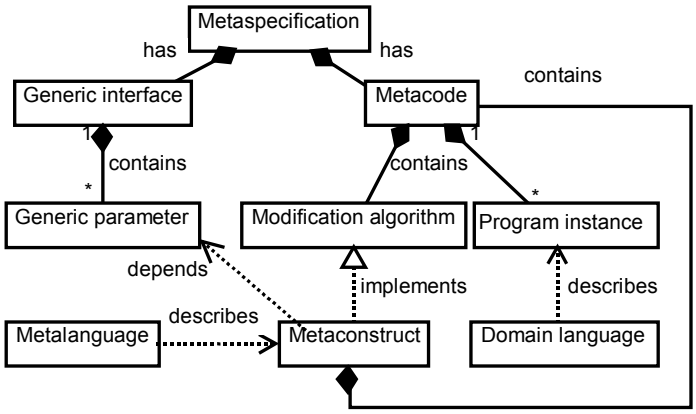

Figure 4. Detailed architecture of a *metaspecification*

Depending upon the representation of a metaspecification, there are two basic dimensions of metaprogramming: the homogeneous and heterogeneous ones. Here we focus on the latter.

Heterogeneous metaprogramming is based on *explicit separation of concerns* and the usage of two different languages in the same metaspecification. 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 designer uses a metalanguage as a higher-level abstraction to integrate together the different domain program instances and make up a metaspecification. The main mechanism allowing to implementing this integration is *external (generic) parameterization*. The latter separates the programming concerns explicitly and bridges the higher- and lower-levels while developing a metaspecification. Then a metaspecification is used as a set of instructions for a metalanguage processor to generate the specific domain program instances.

Heterogeneous metaprogramming can be implemented in several ways [Štuikys, 03]. One way is to use some general-purpose programming language (e.g., Java, C++) in the role of a metalanguage and a domain-specific language (e.g., VHDL, Verilog or SystemC) as a domain language. The other way is to use a dedicated language in the role of a metalanguage. 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 advantages of heterogeneous metaprogramming are as follows. 1) The usage of a domain-independent metalanguage allows automatic (pre-programmed) adaptation to limitations of a particular synthesizer, and automatic documentation generation. 2) The end-user has access to customized instances because

they are explicitly separated. 3) An instance is much more readable than a generic component.

The disadvantages of heterogeneous metaprogramming are as follows. 1) It requires two design environments, thus the validation process is more complex. 2) Clashing of component names must be prevented (manually or automatically).

3.6. Design of Generic Embedded Components Using Metaprogramming

Generic aspects of Embedded Component are represented in a metaspecification. Each metaspecification has an interface for describing the generic parameters, and a body that contains the generic interface and generic functionality of Embedded Components. Metaspecifications serve for 1) concise representation of the families of qualified instances that have the related functionality, and 2) selection and generation of the particular domain component instances depending upon the values of the generic parameters introduced through a metalanguage interface.

Development of a metaspecification can be described as follows (see Figure 5). The domain (represented by one or more available component instances, models and the requirements) is analyzed, and the modification concerns are identified and separated. These concerns represent the variable aspects in a domain that depend upon generic parameters. The separated concerns are expressed through generic parameters, implemented using the metaprogramming techniques, and then integrated back with the fixed aspects of a domain that are orthogonal with respect to the generic parameters. The result is a metaspecification that encapsulates a family of the related Embedded Component instances and implements a Generic Embedded Component. For illustrative example, see Figure 6.

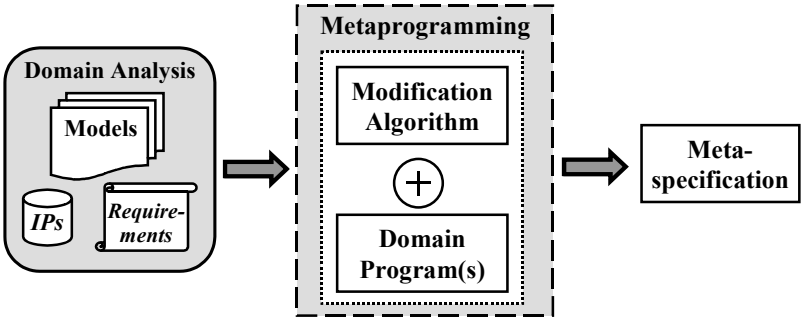

Figure 5. Framework for development of a *metaspecification*

```
void generate_gate(String func, int no, int width) {
// generating VHDL entity of a gate ...
   println("ENTITY GATE_"+func+"_"+no+"x"+width+" IS");
   print("PORT (X0");
   for(int i=1; i<no; i++) print(", X"+i);
   print(" : IN STD_LOGIC");
   if (width > 1)
      print("_VECTOR("+(width-1)+" downto 0)");
   println(";");
   print("\t Y : OUT STD_LOGIC");
   if (width>1)
      print("_VECTOR("+(width-1)+" downto 0)");
   println(");");
   println("END GATE_"+func+"_"+no+"x"+width+";");
// generating VHDL architecture of a gate ...
   println("ARCHITECTURE BEHAVE OF ");
   println("\t\t GATE_"+func+"_"+no+"x"+width+" IS");
   println("BEGIN");
   print("\t\t Y <= X0");
   for(int i=1; i<no; i++) print(" "+func+" X"+i);
   println(";");
   println("END BEHAVE;");
}
```

```
ENTITY GATE_AND_3x8 IS
PORT (X0, X1, X2: IN STD_LOGIC_VECTOR(7 downto 0);
       Y: OUT STD_LOGIC_VECTOR(7 downto 0));
END GATE_AND_3x8;

ARCHITECTURE BEH OF GATE_AND_3x8 IS
BEGIN
      Y <= X0 AND X1 AND X2;
END BEH;
```

Figure 6. Generic gate specification (metalanguage - Java, domain language - VHDL), and its VHDL instance (when *function* = AND, *no* = 3, *width* = 8)

In general, a metalanguage is used to express all possible variability in a domain, while a domain language is used to express the invariant part or commonality in a domain. As metaparameters obtain values from a restricted set relevant to a domain, a metaspecification describes the family of the related component instances in a domain. Thus, a metaspecification that expresses in total commonality and variability of a domain as well as a metalanguage environment (processor, compiler) is a *domain generator*.

3.7. Generation from UML-based Specifications Using Metaprogramming Techniques

Design of a generic embedded component can be split into two parts: the structural and behavioral ones. Our aim here is to demonstrate how the structural part of a given design problem can be specified at a higher abstraction layer using UML.

To implement the transformation process from the UML-based specifications, the design methodology must (1) ensure a *mapping* between the UML subset used to model HW and the HDL abstractions. (2) Implement a set of *translation rules* (and an automatic translation program, if possible) between the UML-based specification of a HW model and the HDL-based specification of a HW component.

A mapping is described semi-formally using UML *metamodel*, i.e., the model that describes the syntax of UML diagrams using a subset of UML. A metamodel consists of a class diagram, where classes describe the syntactic components of the used UML diagram. A metamodel for mapping UML to VHDL was initially described in [Damaševičius, 04a] and is extended now. Other mappings also can be used (see, e.g., [McUmber, 99] [Björklund, 02]). Below, we present a mapping between UML class diagrams and a structural subset of VHDL (see Figure 7). 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 7, a) describe different types of connections and associations between classifiers. *Features* (Figure 7, 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.

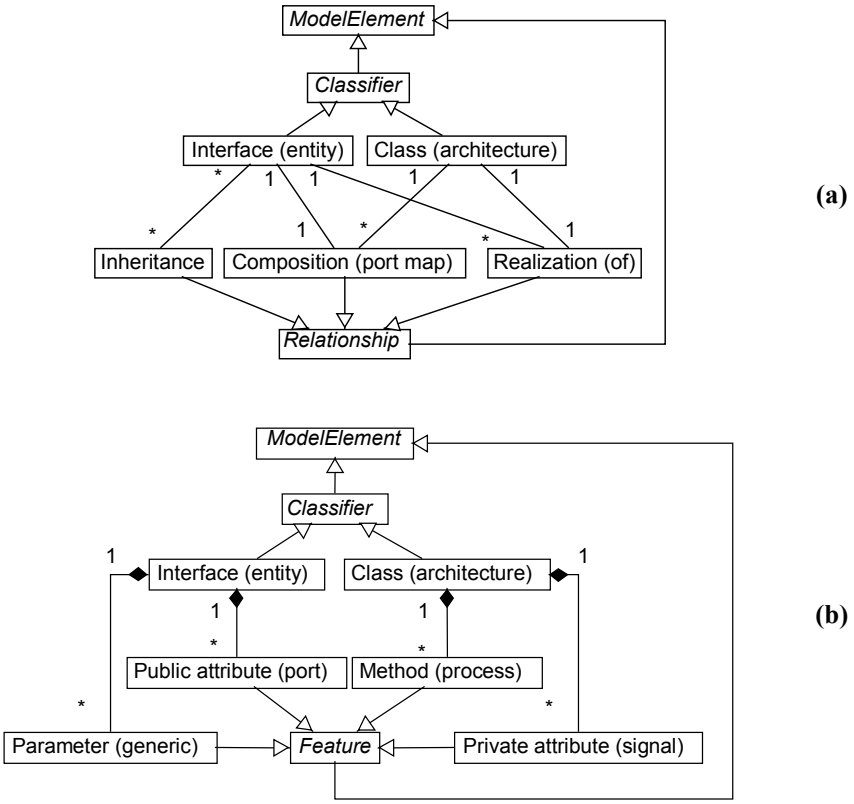

Figure 7. A mapping between UML class diagrams and VHDL structural abstractions: (a) relationships and (b) features

Once the mapping between UML and HDL has been defined, rules that describe the translation process between UML and HDL can be formulated. The aim of the translation rules is to describe how an

instance of a UML metamodel (i.e., any UML model described using a subset of UML defined in a metamodel) can be transformed into an instance of a target model (i.e., a concrete HDL 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 [Selic, 02].

We have implemented code generation using metaprogramming *integrated with* in *UMLStudio* [PragSoft, 03]. The tool provides capabilities to generate code from UML diagrams. The generation process is specified using a built-in scripting language *PragScript* that provides access to the data stored by UMLStudio projects. (Note that scripting languages are a kind of metalanguages). A script written in PragScript is, in fact, a *metaspecification* that provides a generic interface to UMLStudio. UMLStudio allows the end-users to write their own scripts if they require code generation in selected language. Using PragScript, we have written a metaspecification, which implements generation of a VHDL structural code from UML class diagrams.

We summarize the difficulties of using UML for HW design as follows:

1) *Specification of interconnections between HW components*. Block-based diagrams are more common for HW designers. They are more straightforward and are oriented at interconnecting components. Whereas UML class diagrams are more intuitive and oriented at reusing and customizing components.
2) *Specification of generic domain functionality*. UML specifications are usually used to specify concrete systems and are not good for describing families of "look-alike" systems and thus managing variability in a domain.
3) *Model validation problem*. HW models must be validated much more accurately than SW models. The problem is that UML models describe systems at a high level of abstraction and leave many details for a designer to implement later.
4) *Increased initial development time*. The designers must get used to a new design paradigm and spend much time for developing a library of basic OOD models.

Furthermore, a great deal of work still must be done in adapting UML for HW and ES design, standardizing UML extensions for parallelism and real time design, developing UML-based tools and integrating them into a HW/SW co-design and SoC design flow before the full potential of the model-based and OOD might be exploited.

4. Experiments and Case Study

We demonstrate applicability of our approach by designing an Embedded Component for two applications: communication control and fault-tolerance. Two different well-proven models are considered: protocols and triple redundancy model (TRM). Below, we describe the experiments using an Embedded Component design framework shown in Figure 3. First, we begin with the analysis of a domain.

4.1. Analysis

The main purpose of communication control is to ensure relevant transmission of data (e.g., operands, commands, addresses, etc.) to and from the IP. Transmission can be described using different rules or *protocols*, i.e. an agreed format for transmitting data between the IPs. In Figure 8, we present a generalized architecture of a communication control circuit. Here, IP is a third-party soft IP that implements basic functionality of a circuit, and ASPC (FSM) is an application-specific protocol controller (finite state machine) that controls the reading and writing of data to and from IP, respectively. We consider here two common communication protocols, namely, handshake protocol that deals with an asynchronous flow of data, and FIFO protocol that deals with sudden bursts of data in a producer-consumer model.

The main purpose of fault-tolerance is to ensure operation of ES under normal as well as exceptional conditions. Fault-tolerance usually relies on *redundancy,* i.e., the addition of resources, time, or information beyond that is needed for normal system operations (see Figure 9). Here IP_1, IP_2, IP_3 are instances of the same IP component, and ASIC (voter) is an application-specific integrated circuit that implements majority voting. There are three types of redundancy as described in sub-section 3.1.

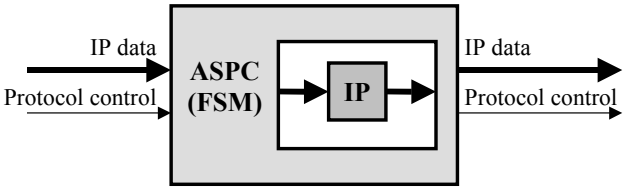

Figure 8. Generalized architecture of a communication control circuit

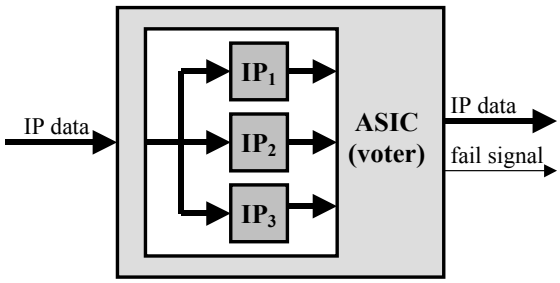

Figure 9. Generalized architecture of a fault-tolerant circuit

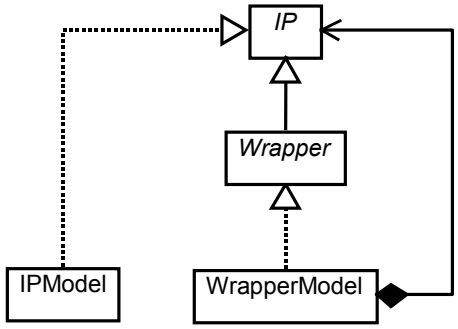

Figure 10. Specification of a Wrapper design pattern

4.2. Specification

To specify an Embedded Component, we have used UML class diagrams and applied a Wrapper design pattern (see Figure 10) [Damaševičius, 03]. Wrapper design pattern allows adapting an interface and behavior of the IP component to the context of a given application. It allows specifying well-proven domain models within well-understood domains (see Section 3.1). Below, we explain it briefly.

The abstract class (*entity* in VHDL) *Wrapper* inherits the I/O ports of the *IP,* and declares new I/O ports for wrapper functionality. The class (*architecture* in VHDL) *IPModel* implements the functionality of entity *IP*. The architecture *WrapperModel* implements the functionality of *Wrapper* and contains component *IP*. This description means that *WrapperModel* wraps *IPModel* with a new functionality.

4.3. Design

We have designed the universal wrapper generator to automatically generate five different wrappers (Handshake, FIFO, Space TMR, Time TMR, and Data TMR) for the black-box third-party soft IP cores described in VHDL. The design flow for implementing the analyzed architectures (see Figures 8, 9) is shown in Figure 11 and presented below.

1) The designer specifies a design problem in UML class diagrams using a Wrapper design pattern. We use *UMLStudio* [PragSoft] as a front-end tool to draw UML diagrams. The designer develops an UML metamodel 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.
2) Since the structural VHDL model is not enough for a wrapper, and UML class diagrams cannot describe functionality, several Java metaspecifications were developed. These metaspecifications capture the behavioral models of wrapper functionality using the *heterogeneous metaprogramming* techniques (metalanguage – Java, domain language – VHDL). Each metaspecification is a Java class, which encapsulates a generic domain entity (e.g., FIFO buffer, voter, etc.). Java processor processes metaspecifications and generates specific behavioral VHDL models for a target system using values of the generic parameters specified via class constructor.
3) The VHDL parser analyses the supplied soft IP source code, constructs a syntax tree, and extracts the values of the parameters for generation.
4) The universal wrapper generator performs wrapping of the 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.

The generation process is fully automatic. The user only has to supply values of the parameters for the wrapper generator. The result is a fully synthesizable reusable instance in VHDL that can be used as an Embedded Component for ES design.

4.4. Generation of Reusable Instances

We use two kinds of metaspecifications in our design flow (see Figure 11): (1) a script developed using embedded *UMLStudio* scripting language *PragScript*,

and (2) the metaspecifications of behavioral VHDL models developed using an external metalanguage (Java).

The first metaspecification is for describing the *structural* variability of a component family, while the second one is for representing the *behavioral* variability. Note that an external metalanguage can be used for specifying both structural and behavioral variability of a design problem, which leads to metaprogramming-only implementation [Štuikys, 02]. Here our aim was to integrate the different kinds of meta-specifications into a unified design flow.

A metaspecification that describes a generic embedded component is used as a set of instructions for a metalanguage processor (compiler) to generate the domain language code (reusable instances) depending upon the values of the generic parameters specified by a designer or other program.

The tools for implementing generation process are conventional compilers because general-purpose programming languages such as C++, Java can be used in the role of metalanguages, too (see Figure 7 and [Štuikys, 03]).

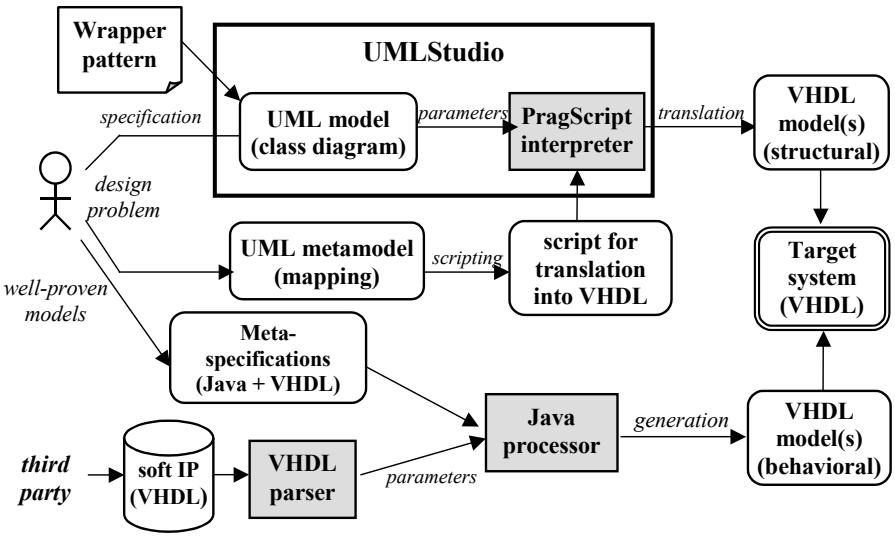

Figure 11. Implementation of wrapping for well-proven models

4.5. Results

In our experiments, we have used the freely available third-party soft (HW) IPs as follows. 1) Free-6502 core [Kessner, 99] is a CPU core compatible with 8-bit 6502 microprocessor. 2) DRAGONFLY core [LEOX, 01] is a 8-bit controller that can be used for serial communication management, FLASH and SDRAM control, etc. 3) AX8 core [Wallner, 01] is a 16-bit AT90Sxxxx compatible micro-controller core. 4) i8051 micro-controller [Givargis, 00] is compatible with 8-bit microprocessor designed by Intel.

Synthesis results of the original soft IPs and the generated wrappers (Synopsys; CMOS 0.35 μm technology) are presented in Tables 1-4.

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 Space TRM wrapper, 44% for the Time TRM wrapper, 10% for the Data TRM wrapper, 10% for the Handshake wrapper, and 50% for the FIFO (size = 4) wrapper.

Table 1. Synthesis results (FIFO and handshake models; area)

Soft IP	Area, cells (IP)	Increment area, cells (Handshake)	Over-head	Increment area, cells (FIFO(4))	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 %

Table 2. Synthesis results (triple redundancy models; area)

Soft IP	Area, cells (Space TRM)	Over-head	Area, cells (Time TRM)	Over-head	Area, cells (Data TRM)	Over-head
Free-6502	678	15 %	2024	43 %	348	7 %
Dragonfly	698	12 %	3973	68 %	1024	17 %
AX8	956	12 %	3576	45 %	874	11 %
i8051	406	2 %	4314	18 %	1142	5 %

Table 3. Synthesis results (FIFO and handshake models; est. power usage)

Soft IP	Power, uW (IP)	Power, uW (Handshake wrapper)	Overhead	Power, uW (FIFO(4) wrapper)	Overhead
Free-6502	**8.2693**	**0.8607**	**10 %**	**4.9414**	**60 %**
Dragonfly	**19.9421**	**5.2775**	**26 %**	**5.3653**	**27 %**
AX8	**31.2318**	**13.4852**	**43 %**	**5.3563**	**17 %**
i8051	**50.5518**	**16.6699**	**33 %**	**10.2537**	**20 %**

Table 4. Synthesis results (triple redundancy models (TRM); est. power usage)

Soft IP	**Power, uW (space TRM)**	**Overhead**	**Power, uW (data TRM)**	**Overhead**	**Power, uW (time TRM)**	**Overhead**
Free-6502	25.844	212 %	11.120	34 %	12.211	47 %
Dragonfly	58.247	192 %	23.639	18 %	21.721	9 %
AX8	34.605	11 %	34.082	9 %	40.595	30 %
i8051	100.262	98 %	63.912	26 %	56.517	12 %

The synthesis results show the following average increase in estimated power usage of the wrapped soft IPs with generated wrappers with respect to the original soft IPs: 26% for the Handshake wrapper, and 39% for the FIFO (buffer size = 4) wrapper. 201% for space redundancy, 26% for data redundancy and 23% in time redundancy for the generated fault tolerant components with respect to the original soft IPs. Note that space redundancy here means that there are 3 instances of the original soft IP. The considerable power usage of protocol wrappers can be explained by the fact that the protocol-based communication to transfer data requires more switching power than direct point-to-point communication.

Furthermore, the experiments we have carried out show that using the third-party soft IPs as black-box entities and well-proven models for their modification enables us to simplify the design validation problem. This result follows from the fact that we use the qualified soft IPs and apply thorough testing procedures only for the newly created functionality introduced by the performed modifications.

5. Evaluation and Conclusions

Future Ambient Intelligence (AmI) systems will require a diversity of components with much higher complexity in order to respond to new requirements and constraints. Management of complexity and variability in design through raising the level of abstraction has already been approved in the past and should be exploited further.

In this paper, we have suggested the Generic Embedded Component Model within a general design framework as a higher-level design abstraction to support design of the AmI systems. The model has the following properties. 1) It is common for design of SW as well as HW components. 2) It contains different representation forms (generic specification, family of related instances, and embeddable instances). 3) It supports generative reuse.

We have also proposed to apply the combined systematic domain analysis methods and to use the high-level abstractions and object-oriented specification methods and metaprogramming for representing the model and automatically generating the reusable instances. We have restricted the application of the domain analysis methods for the well-understood domains only, because Embedded Components for AmI systems must be based on the well-proven models in the first place.

The suggested design framework particularly focuses on the usage of heterogeneous metaprogramming techniques. The techniques allow us: 1) To describe the Generic Embedded Components at a higher level of abstraction. 2) To parameterize them with respect to the variety of user- and application-specific requirements. 3) To flexibly manage variability in a domain. 4) To generate the customized ready-to-use Embedded Component instances for well-understood domains of application.

The techniques can be applied in two different modes: either as a built-in implementation embedded into higher-level design tools (e.g., UML-based tools, but there is a little progress now due to UML restrictions), or as an independently used technique to support generative reuse. We have demonstrated suitability and validity of the proposed model for HW domain by developing a universal wrapper generator for two applications: communication control and fault-tolerance.

Future work still requires many efforts for adopting and integrating the higher-level abstractions such as UML and metaprogramming techniques into a unified design flow in order to fully exploit their capabilities for embedded system design. If one could combine the retrieval of third-party soft IPs with automatic domain analysis and design space exploration (for different soft IP characteristics tradeoffs), at the same time providing information for variability management of soft IPs, the approach could substantially increase design productivity and flexibility for developing future AmI-oriented embedded components and systems.

References

[Agaësse, 99] J. F. Agaësse and B. Laurent. Virtual Components Application and Customization. *Proc. of Design, Automation and Test in Europe DATE* 99, *Munich, Germany, March* 9-12, 726-727.

[Åström, 01] P. Åström, S. Johansson, P. Nilsson. Application of Software Design Patterns to DSP Library Design. *Proc. of the* 14th *Int. Symposium on System Synthesis* (ISSS'01), *October* 1-3, 2001, *Montreal*, *Canada*, 239-243.

[Basten, 03a] T. Basten, L. Benini, A. Chandrakasan, M. Lindwer, J. Liu, R. Min, and F. Zhao. Scaling into Ambient Intelligence. *Proc. of Design, Automation and Test in Europe DATE* 03, *Munchen, Germany*, 3-7 *March* 2003. *Los Alamitos*: *IEEE CS Press*, 2003, 76-81.

[Basten, 03b] T. Basten, M. Geilen, H. de Groot, Eds. Ambient Intelligence: Impact on embedded-system design. *Kluwer Academic Publishers, Boston, November* 2003.

[Becker, 01] M. Becker. Generic Components – A Symbiosis of Paradigms. *G. Butler, S. Jarzabek (Eds.): Generative and Component-Based Software Engineering, Second International Symposium, GCSE* 2000, *Erfurt, Germany, October* 9-12, 2000. *Lecture Notes in Computer Science, Vol*.2177 *Springer* 2001, 100-113.

[Berkel, 94] K. Van Berkel. Handshake Circuits: An Asynchronous Architecture for VLSI Programming. *Cambridge University Press*, 1994.

[Björklund, 02] D. Björklund and J. Lilius, "From UML Behavioral Descriptions to Efficient Synthesizable VHDL", in NORCHIP 2002, 11-12 November 2002, Copenhagen, Denmark.

[Boekhorst, 02] F. Boekhorst. Ambient Intelligence, the Next Paradigm for Consumer Electronics: How Will it Affect Silicon? *Proc. of Int. Solid-State Circuits Conference* (ISSCC), *San Francisco, CA*, 2002, 28-31.

[Booch, 98] G. Booch, I. Jacobson, J. Rumbaugh, J. Rumbaugh. The Unified Modeling Language, User Guide. *Addison-Wesley*, 1998.

[Braun, 02] F. Braun, J. Lockwood, M. Waldvogel. Protocol Wrappers for Layered Network Packet Processing in Reconfigurable Hardware. *IEEE Micro*, 22(3), 2002, 66-74.

[Cai, 05] Y. Cai (Ed.). Ambient Intelligence for Scientific Discovery – Foundations, Theories, and Systems. *Lecture Notes in Computer Science Vol*.3345, *Springer* 2005.

[Chen, 03] R. Chen, M. Sgroi, L. Lavagno, G. Martin, A. Sangiovanni-Vincentelli, J. Rabaey. UML and Platform-based Design. *L. Lavagno, G. Martin, and B. Selic, Eds. UML for Real. Kluwer Academic Publishers, Boston, November* 2003, 107-126.

[Cyr, 01] G. Cyr, G. Bois, M. Aboulhamid. Synthesis of Communication Interface for SoC using VSIA Recommendations. *DATE* 2001 *Designer's Forum, March* 13-16, 2001, *Munich, Germany*, 155-159.

[Czarnecki, 00] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools and Applications. *Addison-Wesley*, 2000.

[Damaševičius, 03] 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*, 48-53.

[Damaševičius, 04a] R. Damaševičius, V. Štuikys. Application of UML for Hardware Design Based on Design Process Model. *Asia South Pacific Design Automation Conference* (ASP-DAC 2004), *Yokohama, Japan*, *January* 27-30, 2004, 244-249.

[Damaševičius, 04b] R. Damaševičius, V. Štuikys. Application of the Object-Oriented Principles for Hardware and Embedded System Design. *INTEGRATION, the VLSI Journal*, *Elsevier Ltd*., 2004, 38(2), 309-339.

[Diaz-Herrera, 00] J.L. Diaz-Herrera, V.K. Madisetti. Embedded Systems Product Lines. *Proc. of* 22nd *Int. Conference on Software Engineering (ICSE), Workshop on Software Product Lines: Economics, Architectures, and Implications, Limerick, Ireland, June* 2000, 90-97.

[Diaz-Herrera, 01] J.L. Diaz-Herrera. Domain Engineering and Object Technology. *S.K. Chang (Ed.), Handbook of Software Engineering and Knowledge Engineering, Vol*.I: *Fundamentals. Singapore: World Scientific Publishing*, 2001.

[Doucet, 02a] F. Doucet, R.K. Gupta. Microelectronic System-on-Chip Modeling using Objects and their Relationships. *Online Symposium for Electrical Engineers* (OSEE 2000), 2000.

[Ducatel, 01] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leitjen, J.C. Burgelman. Scenarios for Ambient Intelligence in 2010. *IST Advisory Group Report, IPTS, Seville, Spain,* 2001.

[Edwards, 03] M. Edwards, P. Green. UML for Hardware and Software Object Modeling. *L. Lavagno, G. Martin, and B. Selic, Eds. UML for Real, Kluwer Academic Publishers*, 2003, 127-148.

[Edwards, 97] S. Edwards, L. Lavagno, E.A. Lee, A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Proc. of the IEEE*, 85 (3), 1997, 366-390.

[Eggermont, 02] E.D.J. Eggermont (Ed.). Embedded Systems Roadmap 2002. *STW Technology Foundation/PROGRESS, Utrecht, the Netherlands*, 2002.

[Entrena, 01] L. Entrena, C. Lopez, E. Olias. Automatic Generation of Fault Tolerant VHDL Designs in RTL. *Forum on Design Languages FDL'*2001, *Lyon, France*.

[Fuhrman, 95] C.P. Fuhrman, S. Chutani, H.J. Nussbaumer. Hardware/software fault tolerance with multiple task modular redundancy. *IEEE Symposium on Computers and Communications* (ISCC'95), *June* 27 - 29, 1995, *Alexandria, Egypt*, 171-177.

[Gajski, 97] D. Gajski. Principles of Digital Design. *Prentice Hall*, 1997.

[Gamma, 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley*, 1995.

[Gerstlauer, 02] A. Gerstlauer, D.D. Gajski. System-Level Abstraction Semantics. *Proc. of International Symposium on System Synthesis* (ISSS'02), *Kyoto, Japan, October* 2002.

[Gharsalli, 02] F. Gharsalli, S. Meftali, F. Rousseau, A.A. Jerraya. Automatic Generation of Embedded Memory Wrapper for Multiprocessor SoC. *Proc. of Design Automation Conference DAC* 2002, *June* 10-14, 2002, *New Orleans, LA, USA*, 596-601.

[Gibbons, 03] J. Gibbons, J. Jeuring (Eds.). Generic Programming. *Kluwer Academic Publishers*, 2003.

[Givargis, 00] T. Givargis. Intel 8051 micro-controller, 2000, *http://www.cs.ucr.edu/~dalton/i8051/i8051syn/*.

[Goudarzi, 04] M. Goudarzi, S. Hessabi, A. Mycroft. Object-Oriented Embedded System Development Based on Synthesis and Reuse of OO-ASIPs. *Journal of Universal Computer Science*, *Vol*.10, *No*.9, *September* 2004, 123-1156.

[Haase 99] J. Haase. Design Methodology for IP Providers. *Proc. Design Automation and Test in Europe Conference DATE*'1999, *Munich, Germany, March* 1999, 728-732.

[Harsu, 02] M. Harsu. A survey on domain engineering. *Report* 31, *Institute of Software Systems, Tampere University of Technology, December* 2002.

[ITRS, 03] ITRS (International Technology Roadmap for Semiconductors), 2003.

[Jacobson, 97] I. Jacobson, M. Griss, P. Jonsson. Software Reuse: Architecture, Process and Organization for Business Success. *Addison-Wesley*, 1997.

[Jerraya, 99] A.A. Jerraya, M. Romdhani, Ph.Le Marrec, F. Hessel, P. Coste, C. Valderrama, G.F. Marchioro, J.M. Daveau, N.-E. Zergainoh. Multi-language Specification for System Design and Co-design. *In A.A. Jerraya, J. Mermet, Eds. System Level Synthesis. Kluwer Academic Publishers*, 1999.

[Jong, 02] G. de Jong. UML-based design methodology for real-time and embedded systems. *Proc. of Design Automation and Test in Europe Conference DATE* 2002, 4-8 *March* 2002, *Paris, France*, 776-778.

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

[Kessner, 99] D. Kessner. Free-6502 core, 1999, *http://www.free-ip.com/6502/.*

[Kiczales, 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin. Aspect-Oriented Programming. *Proc. of the European Conference on Object-Oriented Programming. Lecture Notes in Computer Science*, *Vol*.1241, *Springer-Verlag*, 1997, 220-242.

[Kienhuis, 97] B. Kienhuis, E. Deprettere, K. Vissers, P. van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. *In: L. Thiele, J. Fortes, K. Vissers, V. Taylor, T. Noll, and J. Teich, Eds., Proc. of ASAP* '97, 338-349.

[Kleinjohann, 98] B. Kleinjohann. Invited Talk: Multilanguage Design. *Proc. of Int. IFIP WG* 10.3/*WG* 10.5 *Workshop on Distributed and Parallel Embedded Systems* (*DIPES*'98), *Paderborn, Germany*, 1998.

[Lee, 98] E.A. Lee, A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on CAD, Vol*.17, *No*.12, 1998, 217-1229.

[LEOX, 01] LEOX Team, DRAGONFLY micro-core, 2001, *http://www.leox.org.*

[Lindwer, 03] M. Lindwer, D. Marculescu, T. Basten, R. Zimmermann, R. Marculescu, S. Jung, E. Cantatore. Ambient Intelligence Visions and Achievements: Linking abstract ideas to real-world concepts. *Proc. of Design, Automation and Test in Europe DATE* 03, *Munchen, Germany*, 3-7 *March* 2003, *Los Alamitos*: *IEEE CS Press*, 2003, 10-15.

[Martin, 02] G. Martin. UML for embedded systems specification and design: motivation and overview. *Proc. of Design Automation and Test in Europe Conference DATE* 2002, 4-8 *March* 2002, *Paris, France, Los Alamitos: IEEE Computer Society Press*, 2002, 773-775.

[McUmber, 99] W.E. McUmber, B.H.C. Cheng. UML-Based Analysis Of Embedded Systems Using a Mapping to VHDL. *IEEE High Assurance Software Engineering* (*HASE* 1999), *November*, 1999.

[Meguerdichian, 01] S. Meguerdichian, F. Koushanfar, A. Mogre, D. Petranovic, M. Potkonjak. MetaCores: Design and Optimization Techniques. *Proc. of Design Automation Conference DAC*'2001, *Las Vegas, Nevada, USA, June* 18-22, *ACM*, 2001, 585-590.

[Mihal, 02] 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, No*.19, 2002, 2-12.

[Nitsch, 03] C. Nitsch, C. Lara, U. Kebschull. A Novel Design Technology for Next Generation Ubiquitous Computing Architectures. *Proc of Int. Parallel and Distributed Processing Symposium* (*IPDPS*'03), *April* 22-26, 2003, *Nice, France*, 191b.

[Ossher, 01] H. Ossher, P. Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. *In M. Aksit, Ed. Software Architectures and Component Technology: The State of the Art in Software Development. Boston: Kluwer Academic Publishers*, 2001.

[Pflanz, 98] M. Pflanz, H.T. Vierhaus. Generating Reliable Embedded Processors. *IEEE Micro, Vol*.18, *No*. 5, 1998, 33-41.

[Pragsoft] PragSoft Corp. *UMLStudio*, *http://www.pragsoft.com.*

[Prieto-Diaz, 91] R. Prieto-Diaz, G. Arango. Domain Analysis and Software Systems Modeling. *Los Alamitos: IEEE Computer Society Press*, 1991.

[Remagnino, 03] P. Remagnino, G.L. Foresti, N.Monekosso. Coarse to fine scene understanding, first steps towards an Ambient Intelligence system. 8*th National Congress of Italian Association for Artificial Intelligence AI*IA* 2003, *Workshop on Ambient Intelligence*, 23 *September*, 2003, *Pisa, Italy*.

[Riva, 03] G. Riva, P. Loreti, M. Lunghi, F. Vatalaro, F. Davide. Presence 2010: The Emergence of Ambient Intelligence. *In G. Riva, F. Davise, and W.A.I. Jsselsteijn, Eds. Being There: Concepts, effects and measurement of user presence in synthetic environments*, *Amsterdam: Ios Press*, 2003, 59-82.

[Sangiovanni-Vincentelli, 01] A. Sangiovanni-Vincentelli, G. Martin. A Vision for Embedded Systems: Platform-Based Design and Software Methodology. *IEEE Design and Test of Computers, Vol*.18, *No*.6, *November/December* 2001, 23-33.

[Selic, 02] B. Selic. Complete High-Performance Code Generation from UML Models. *Embedded Systems Conference* 2002, *San Francisco, USA*.

[Selic, 03a] B. Selic. Architectural Patterns for Real-Time Systems. *In L. Lavagno, G. Martin, and B. Selic, Eds. UML for Real,. Kluwer Academic Publishers*, 2003, 171-188.

[Selic, 03b] B. Selic. Models, Software Models and UML. *In L. Lavagno, G. Martin, and B. Selic, Eds. UML for Real, Kluwer Academic Publishers, Boston*, 2003, 1-16.

[Sheard, 01] T. Sheard. Accomplishments and Research Challenges in Metaprogramming. 2*nd Int. Workshop on Semantics, Application, and Implementation of Program Generation* (*SAIG*'2001), *Florence, Italy. Lecture Notes in Computer Science, Vol.*2196, *Springer*, 2-44.

[Siegmund, 00] R. Siegmund, D. Mueller. A Method for Interface Customization of Soft IP Cores. *In R. Seepold, and M. Navidad (Eds.), Virtual Component Design and Reuse. Kluwer Academic Publishers*, 2000.

[Sztipanovits, 02] J. Sztipanovits, G. Karsai. Generative Programming for Embedded Systems. *In D.S. Batory, C. Consel, and W. Taha, Eds. Generative Programming and Component Engineering, Proc. of ACM SIGPLAN/SIGSOFT Conference GPCE* 2002, *Pittsburgh, PA, USA, October* 6-8, 2002. *LNCS Vol.*2487, *Springer* 2002, 32-49.

[Szyperski, 99] C. Szyperski. Component Software beyond Object-Oriented Programming. *Addison-Wesley*, 1999.

[Štuikys, 02] V. Štuikys, R. Damaševičius, G. Ziberkas, G. Majauskas. Soft IP Design Framework Using Metaprogramming Techniques. *In B. Kleinjohann, K. H. (Kane) Kim, L. Kleinjohann, and A. Rettberg, Eds. Design and Analysis of Distributed Embedded Systems, Kluwer Academic Publishers, Boston, July* 2002, 257-266.

[Štuikys, 03] V. Štuikys, 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.

[Vahid, 03] F. Vahid. The Softening of Hardware. *IEEE Computer*, 36(4), 2003, 27-34.

[Vermeulen, 00] A. Vermeulen, F. Catthoor, D. Verkest, H. De Man. Formalized Three-Layer System-Level Reuse Model and Methodology for Embedded Data-Dominated Applications. *Proc. Design Automation and Test in Europe DATE*'2000, 92-98.

[Wallner, 01] D. Wallner. AX8 core, 2001, *http://hem.passagen.se/dwallner/vhdl.html*.

[Weber, 03] W. Weber. Ambient intelligence: industrial research on a visionary concept. *Proc. of the* 2003 *Int. Symposium on Low Power Electronics and Design, Seoul, Korea, New York: ACM Press*, 2003, 247-251.

[Weiss, 99] D.M. Weiss, C.T.R. Lai. Software Product-Line Engineering: A Family-Based Software Development Approach. *Reading: Addison-Wesley*, 1999.

[Yoshida, 01] N. Yoshida. Design Patterns Applied to Object-Oriented SoC Design. *In 10th Workshop on Synthesis and System Integration of Mixed Technologies* (*SASIMI* 2001), *October* 18-19, 2001, *Nara, Japan*.

[Zhu, 01] J. Zhu. MetaRTL: Raising the Abstraction Level of RTL Design. *Proc. Design Automation and Test in Europe DATE* 2001, *Munich, Germany, March* 13-16, 71-76.

Received December 2006.

DOI: 10.5755/j01.itc.36.1.11830