ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2005, Vol.34, No.2

SOFT IP CUSTOMIZATION MODELS BASED ON HIGH-LEVEL ABSTRACTIONS

Robertas Damaševičius, Vytautas Štuikys

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

Abstract. System-on-Chip (SoC) design raises an abstraction level in hardware (HW) design beyond a domain language specification. This requires the introduction and adoption of high-level analysis and specification methods that provide reusability, layericity, orthogonality, heterogeneity and customizability for HW design. HW design has many levels of abstraction. The transition between these levels can be described using a concept of design process. As there may be many representation and design methods for a designer to choose from, while implementing a certain design process, we describe several Design Flow Models aimed at implementing customization in soft IP-based HW design. These models apply several concepts taken from software engineering (object-oriented and pattern-based design, metaprogramming, parsing, and markup languages) to HW design.

Key words: High-level abstraction, customization, reuse, design process, metaprogramming.

1. Indroduction

Today, the entire embedded systems comprised of multiple processors, memories and application-specific circuitry are implemented on a single semiconductor chip. Such systems, called *Systems-on-Chip* (SoC), have hardware and software parts, where "hardware" corresponds to implemented circuit elements, and "software" corresponds to programmed instructions performed by hardware. In recent years, the most frequently discussed topics at conferences, forums and meetings in HW design community are centered on three basic trends as follows: (1) Unification of the design methodologies [1]. (2) Moving towards higher levels of abstraction and *metamodeling* in design [2]. (3) Emergence of a new vision, *Ambient Intelligence,* for future microelectronic systems [3].

Currently, the researchers emphasize the role of parameterization, customization and integration of the pre-designed *soft IP (Intellectual Property*) components, described using a high-level hardware (HW) description language (HDL), such as VHDL, into HW systems. To bridge the enormous gap between the high-level specification of a design problem and its implementation, the researchers propose and designers use a variety of models (representations of a system) at the different levels of abstraction. They vary from the lower-level abstractions such as Gate, RTL to the higher-level ones such as design space layers [4], metacores [5], objects [6], UML diagrams [7, 8], architectural patterns [9, 10], formal models [11], frameworks [12], platforms [13, 14], Petri Nets [15] and SystemC models [16], which should ensure higher design quality, productivity and reuse.

High-level models are created using a variety of modeling languages and are used to make essential architectural design decisions within a certain design framework. Some models and abstractions are used at both lower and higher levels (e.g., FSM). These abstractions allow hiding the lowest physical layer (transistors and wires) and contributing to increasing design productivity in the domain.

One way to reduce the embedded system design time and costs is to reuse the pre-existing soft IPs systematically [17]. The design of soft IPs is a complex task due to the following reasons. Smart products usually require various combinations of high performance, low cost and low power. Developers usually design general-purpose embedded systems for reuse in numerous applications. Since the context of their usage is usually unknown, these designs often focus on the functionality issues only, thus yielding widely applicable, but not efficient designs. On the other hand, extreme specialization results in highly efficient design usable only in a single application.

Design of highly reusable soft IPs does not solve the problem of adaptation for a particular context. Customization of soft IPs to fit an application, if not pre-programmed, may require extensive design efforts. The designer has to find the balance between generalization of functionality and specialization of performance characteristics.

The contribution and novelty of this paper is four *Design Flow Models* that describe the application of high-level abstractions (metalanguages, UML, markup) and tools for automated implementation of customization of soft IPs and integration into a HW system. This paper is a summary of our previous research and experiments published in various papers [18-25].

The structure of the paper is as follows. Section 2 discusses the process-based view to HW design. Section 3 describes four *Design Flow Models* based on using UML, Design Patterns, Metaprogramming, Parsing and XML/XSLT style sheets for HW design. Section 4 summarizes our experiments in implementing the wrapping design process using the *Models*. Section 5 evaluates the results and presents a discussion. Finally, Section 6 presents the conclusions.

2. Process-based view to HW design

The IP-based design is a vision of how researchers and designers *introduce* reuse-based ideas (abstractions, models, instructions, rules, etc.), *implement* them by creating tools, and *use* tools in order to achieve a pre-specified design aim while designing and manufacturing sophisticated products, such as embedded systems. The SoC design methodologies generally require the following features:

(1) *Layericity* – HW has many layers of abstraction (HW, embedded SW, RTOS, etc.) for which the underlying, subsequent design-flow steps are abstracted. By carefully defining HW abstraction layers and developing new representation and automated design methods, an electronic system design flow is realized.

(2) *Orthogonality* – based on the principle of separation of concerns, the IP-based HW design methodology must clearly separate behavioral aspects from implementation, and communication from computation [26].

(3) *Reusability* – process-based HW design promotes reuse at all levels of abstraction, including reuse of soft IP components, high-level models, testbenches, architectures, etc. [27]

(4) *Customizability* – a platform can be customized for a range of applications within a certain domain using a variety of mechanisms from simple parameterization to sophisticated transformations [28].

There are many levels of abstraction in HW design. For simplicity, we can consider only two levels, the higher level (HL) and the lower level (LL). These levels differ in the considered domain entities (e.g., HL: objects and messages, LL: transistors and wires) and their representation methods used (e.g., HL: UML diagrams, LL: HDL specification). Each level consequently can be subdivided into multiple sub-levels. The relation between the levels of abstraction can be described using a concept of *design process* [23]. We understand a *design process* as a series of commonly used domain-specific actions, tasks or methods performed to achieve a certain design aim. We formulate the properties of design processes below:

(1) Design processes are *domain-specific* and can be used only for implementing certain well-known models in the domain.

(2) Design processes are *commonly used* by designers.

(3) Design processes are *transformative*, i.e. they are about transforming their input (programs, syntax trees) into output.

(4) Design processes are *executable*, i.e., they not only describe what is done, but also imply how it can be done using some well-defined method or approach.

(5) Design processes are *design context-specific*, i.e. they reside within a certain *design framework*.

For example, a common design process used in communication-based design is *wrapping* [20, 21], which adds the communication protocol to the existing soft IP to adapt it in order to allow for communicating with other components in the designed system. Wrapping relates between the same levels of abstraction in design flow (Figure 1).

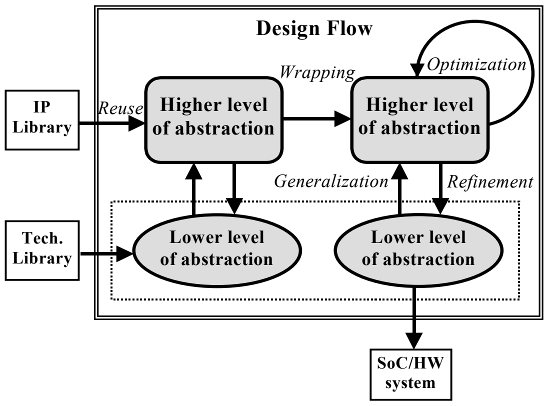
Design Flow
IP Library
Reuse
Higher level of abstraction
Wrapping
Higher level of abstraction
Optimization
Generalization
Refinement
Tech. Library
Lower level of abstraction
Lower level of abstraction
SoC/HW system

Figure 1. Abstractions and design processes

The transition between higher and lower levels of abstraction can be defined as *refinement*. This means the translation between a high-level specification (e.g., UML diagram, Petri net) and a lower level implementation (e.g., VHDL program) in a design environment. The opposite process is *generalization*, used in metamodeling, metaprogramming and development of generic components. *Reuse* means the usage of a soft IP component (with or without modifications) in the designed system. *Optimization* means the improvement of the component's characteristics (area, speed, power usage) via a functionality-preserving modification of the component.

Design processes are introduced into the domain through raising the level of abstraction. Generally, we can distinguish four different levels of abstraction (see Figure 2) in system design:

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

(2) *Metaprogram* level – the development of the generic components (programs) using the internal

mechanisms of the domain language (*polymorphism*) or an external language (*metalanguage*).

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

(4) *Metamodel* level – the representation of model semantics using abstract *metamodels*. A composition of several metamodels implements a *platform*.

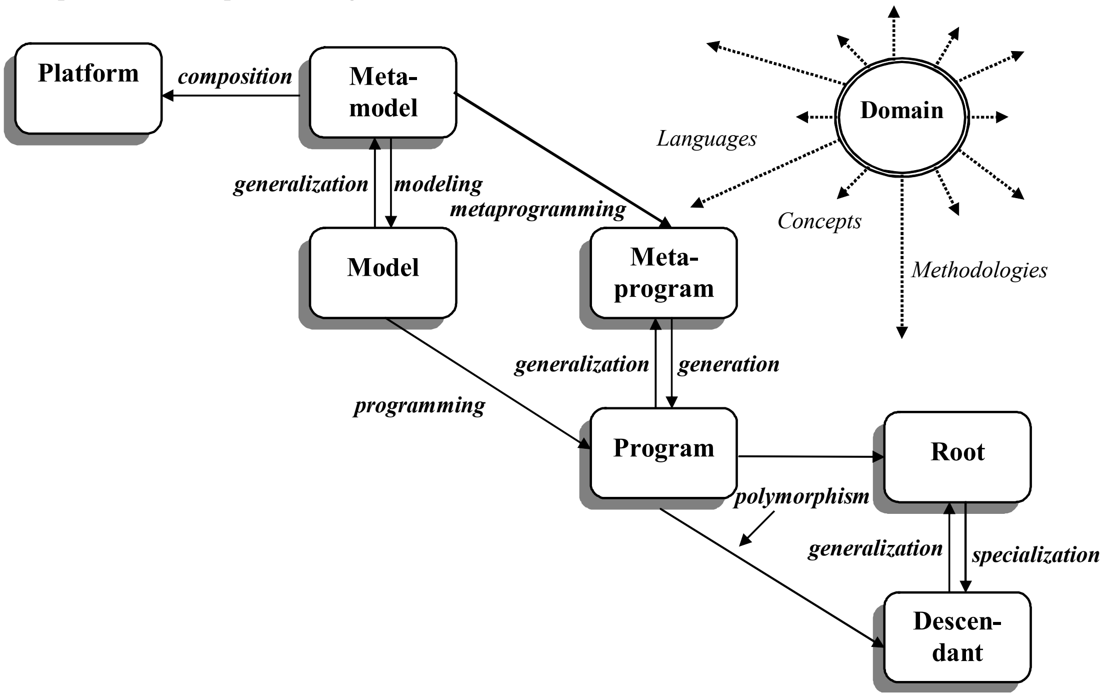

Figure 2. Different levels of abstraction in domain

While the first two levels are usually introduced using the textual specification mechanisms of the programming languages (either object-oriented or the metaprogramming ones), the last two levels can be introduced using a graphical notation such as UML.

We summarize our perception of process-based HW design as a two-dimensional space of wrapping and refinement processes, the *basic abstract processes* within our approach. The first one relates with a design (sub)problem *specification* using some abstraction. The second one relates with the *transformation* of the higher-level specification into a representation at a lower-level of abstraction.

Furthermore, each of these processes contain different sub-processes (*operations*). The partition and selection of processes from the design process space is a particular concern of a designer. The ideal case is the one in which the design process is described using a unique abstraction hiding the lower-level design processes, and implemented using a well-defined design flow. Next, we describe several design flow models, which we have identified during our research.

3. Design flow models

3.1. Model 1: UML-based

Model 1 describes the usage of UML for HW design. A designer has three main problems to solve. (1) How to raise the level of abstraction from the domain language specification to the UML specification? (2) How to modify the UML specification to implement a certain design process? (3) How to refine the UML specification to domain language?

To solve these problems, the design flow must provide the following features. Firstly, it must ensure a mapping between the UML subset used to model a target system and the domain language (e.g., VHDL) abstractions. Secondly, it must implement a set of translation rules between UML and domain language. A mapping is usually described formally using a metamodel that describes the syntactic components of the used UML diagram and the corresponding domain language abstractions (Table 1).

Table 1. UML-VHDL metamodel

UML model element	VHDL abstraction
Interface	Entity
Class	Architecture
Inheritance	-
Composition	Port map
Realization	Of
Public attribute	Port (external signal)
Private attribute	Signal (internal)
Method	Process
Parameter	Generic

The aim of the translation rules is to describe how an instance of the UML metamodel (i.e., any UML model described using a subset of UML defined in a metamodel) can be transformed from and to an instance of a target model (i.e., a domain language specification that describes the implementation of a system specified using UML). These rules can be implemented manually by a designer, or automatically using a dedicated translation tool.

A target system is not specified freely, but rather using patterns, which are refined to architectures that implement common domain models. *Design patterns* are an abstraction used for representing, abstracting and encapsulating common design solutions in UML, as well as for describing contexts to which they can be applied in an implementation-independent way [29]. The problem is how design patterns should be refined. Design patterns usually are very abstract and can be refined into a target model in a variety of ways that could lead to different implementations.

The design flow in Model 1 is as follows (Figure 3). A designer uses the translation rules to derive the UML diagram of soft IP(s), and then composes it with the additional glue code specified in UML. The specification of the glue code is obtained by defining the design pattern used to implement a well-known domain model. A target system is obtained when the UML diagram is translated into the domain language-based specification. This model was also discussed in [21, 23, 25].

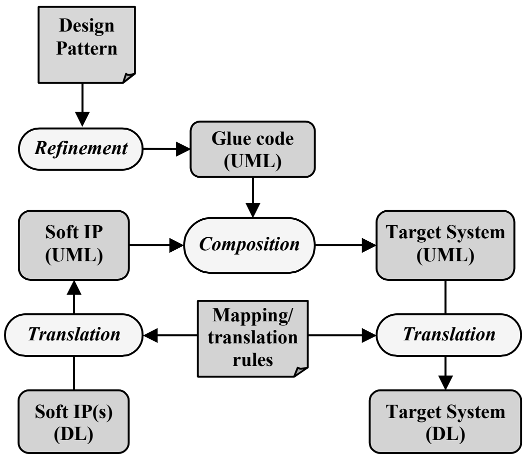

Figure 3. Design flow in Model 1

The problems a designer faces in Model 1 are as follows: (1) Standard UML is not enough for expressing HW domain concepts such as timing. (2) Validation of the object-oriented model of HW. (3) Selection of a suitable domain language. VHDL is closer to HW domain, whereas SystemC is closer to SW domain, but has many restrictions on synthesizability.

3.2. Model 2: Metaprogramming-based

Model 2 describes the application of metaprogramming paradigm [22] for HW design. *Metaprogramming* is a higher-level programming technique that provides a means for manipulating with domain programs as data. The aim of metaprogramming is to create a *metaspecification* – a specification of a program generator for a narrow domain of application. A metaspecification consists of a family of related domain program instances that implement a well-proven domain model. The instances are encapsulated with their modification algorithm that describes generation of a particular instance depending upon the values of generic parameters.

Heterogeneous metaprogramming means programming with two different languages simultaneously. The lower-level language (*domain language, DL*) is used for expressing the basic domain functionality. The higher-level language (*metalanguage, ML*) 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. Then a metaspecification is used as a set of instructions for a metalanguage processor to generate the specific domain program instances.

The problems a designer faces in Model 2 are as follows. (1) It requires two design environments, thus the validation process is more complex. (2) Overgeneralization also could become a problem. (3) The selection of parameter values is a cumbersome task that requires the detailed domain knowledge.

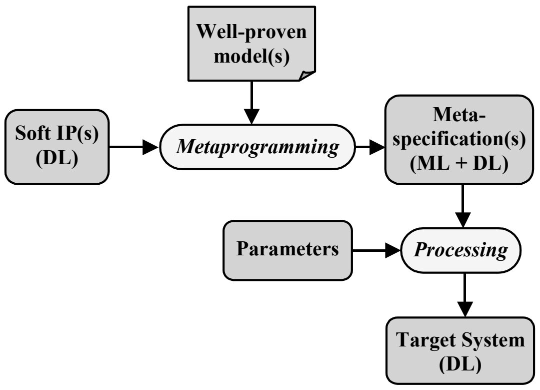

Figure 4. Design flow in Model 2

The design flow in Model 2 is as follows (Figure 4). A designer uses a well-proven domain model to develop a parameterized metaspecification that represents a family of the available target system implementations. A particular instance of a target system is obtained when a metaspecification is processed using values of the generic parameters supplied by a designer. This model was also discussed in more details in [19, 20, 22].

3.3. Model 3: Parsing-based

Model 3 is an extension of Model 2 with *parsing*. The aim is to streamline the selection of the context-dependent parameter values for metaspecifications. Parsing is an application-specific domain analysis method that is concerned with automatic analysis of abstract domain representations – the DL program source code. Parsing decomposes the domain language specification according to the DL syntax into an *Abstract Syntax Tree* (AST). AST then can be used for a variety of purposes such as domain visualization, modelling, extraction of the application-specific information for further customization, optimization and domain code generation.

The problem with parsing-based model is the implementation of parsing itself, which may be very complex. However, for many DLs custom open-source parsers exist that can be easily integrated into a design flow.

The design flow in Model 3 is as follows (Figure 5). It is similar to Model 2. The difference is that application-specific design context represented by soft IP(s) is analyzed automatically and the obtained data are used to generate a target system. This model was also discussed in more details in [24].

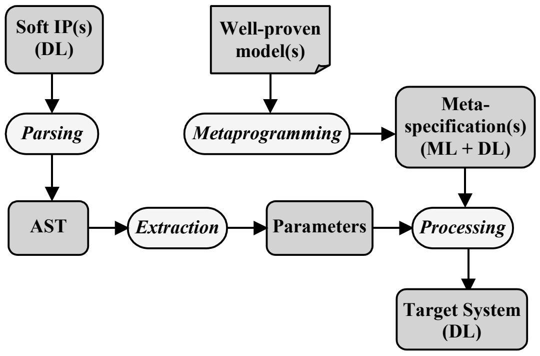

Figure 5. Design flow in Model 3

3.4. Model 4: Markup-based

Recently, XML began to be used as a metalanguage capable of specifying parameterized components and design architecture independently of the modelling language in the embedded system design domain [30]. XML is a markup language that provides a common syntax for describing the structure of data according to their content or meaning. XML also can be described as a metalanguage that allows defining customized domain-specific markup languages for different types of documents.

XML has several advantages as a representation language. The hierarchical topology of data organization reflects the logic structure of data. The tagged data embed the information structure within the data, which makes the processing easier. A designer is not restricted to a limited set of tags defined by proprietary vendors. By defining his own set of tags, he can create a markup language oriented at a specific domain of application. The rules specified by those tags need not be limited to formatting rules. XML allows defining any tags with any rules for any target domain, such as tags representing business rules or data description.

XML is often used with XSL, a family of recommendations for defining transformation and presentation of XML documents. A part of XSL is XSLT – a language for describing transformation of XML documents. An XSLT program (style sheet) is a set of template rules for transforming a source tree into a result tree. The transformation is achieved by associating patterns with templates. Each template rule has two parts. A pattern is used to match nodes in a source tree of the input XML document. A template is used to form a result tree of the output XML document. The structure of the result tree can be completely different from the structure of the source tree.

The design flow in Model 4 is as follows (Figure 6). A designer uses the translation rules to derive the XML-based specification of soft IP(s). Then the designer composes the transformation code in XSL and applies it to the XML-based specification of the original soft IP. The result is the XML-based specification of the system. A low-level implementation of the target system is obtained when the XML-based specification is translated into the domain language specification.

The problems the HW designer faces in Model 4 are as follows: (1) How to decompose the domain for hierarchical structuring of components (according to domain concepts, language syntax, etc.). (2) How to implement parameterization. The existing parameterization mechanisms are weak and not particularly suitable for developing generic components. (3) Low maturity of application in HW domain.

The benefits of using a markup language for HW design are as follows. It allows for convenient representation, access, and management of various types of structured domain information, including HW components and architectures. Furthermore, the designer has the ability to implement automatic modification/ transformation of HW components and generation of documentation files in various formats using XSL style sheets. The systematic application of XML/XSL in HW design could contribute to the increase in design productivity and reuse as well as to provide better documentation capabilities.

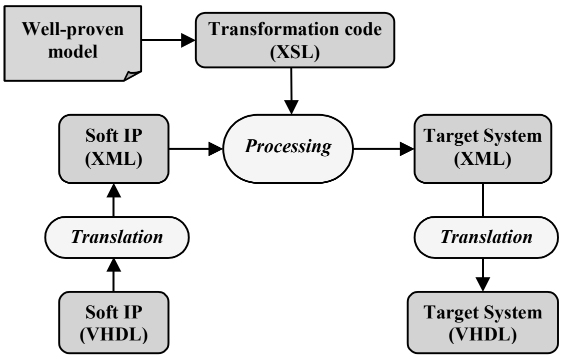

Figure 6. Design flow in Model 4

4. Summary of experiments

Here we present a summary of our experiments performed for implementing a wrapping design process using previously described Design Flow Models in the context of communication-based design [18-25].

The main purpose of communication control is to ensure the relevant transmission of data (e.g., operands, commands, addresses, etc.) to and from the IP. The transmission of data can be described using different rules or *protocols*, i.e. an agreed format for transmitting data between the IPs. In our experiments, we have considered 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 typical wrapping scheme is described in Figure 7. Suppose, we have two components that have to communicate with each other: Source IP and Target IP (see Figure 7, a). If their interfaces or communication schemes are incompatible, we have to insert a Wrapper between Source IP and Target IP in order to convert their communications signals between different communication schemes (see Figure 7, b). Finally, Wrapper is partitioned to allow the separation of Source IP and Target IP (see Figure 7, c).

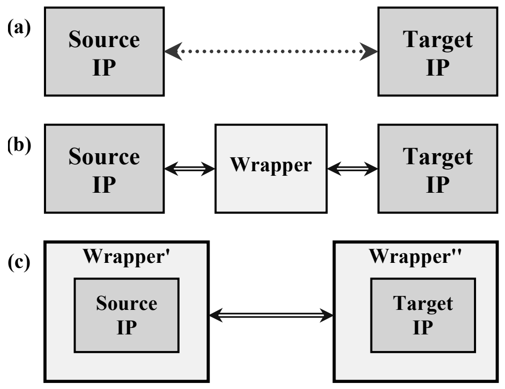

Figure 7. Interconnection of IPs: a) point-to-point, b) via wrapper, c) refined view

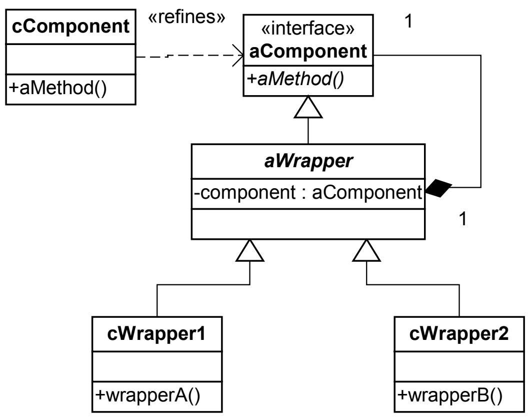

Figure 8. Wrapper design pattern

To specify a wrapping design process at a high level of abstraction, we use a Wrapper pattern that allows adapting an interface and behaviour of the IP component to the context of a given application. Figure 8 presents the UML class diagram of the Wrapper design pattern. The abstract class aComponent specifies an interface that is common for all components and their wrappers. The class cComponent provides an implementation for aComponent class. The abstract class aWrapper specifies a wrapper interface and contains an instance of aComponent class. The cWrapper classes provide the different implementations of a wrapper.

The target architecture is shown in Figure 9. FIFO wrapper (Figure 9, a) wraps *IP* with two instances of the FIFO buffer and additional control logic. The internal clock signal *clk_int* is used to run the control logic (*FSM*) and *IP*. The data are transferred to *IP* when *Push* signal is set to a high level and *Full* signal has a low level. The results are returned when *Pop* signal is set to a high level and *Empty* signal has a low level.

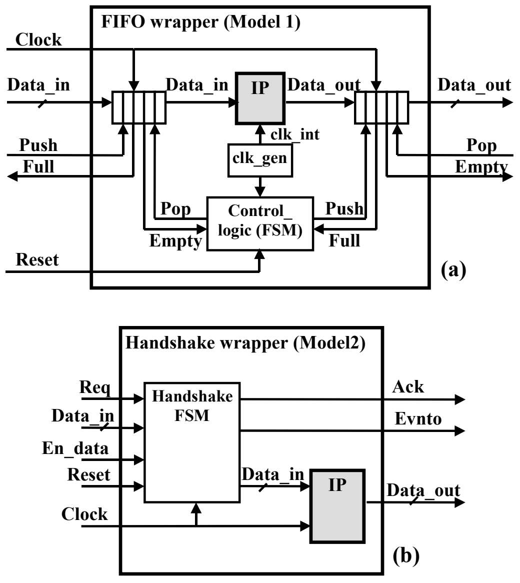

Figure 9. Target architecture: a) FIFO wrapper, and b) handshake wrapper

Handshake wrapper (see Figure 9, b) wraps *IP* with *Handshake FSM*. The data are transferred to *IP* when request signal *Req* is set to a high level and acknowledgement signal *Ack* is received from *Handshake FSM*. The result is returned as soon as *IP* processes the data. Note that signals *Data_in* and *Data out* represent the IP-specific data signals.

Below as an example, we present several high-level descriptions of a design problem. Figure 10, a) presents a metaspecification described in Java that generates a specific GATE component instance with respect to the supplied values of the parameters. Figure 10, b) shows the corresponding implementation in Open PROMOL metalanguage [18]. Figure 10, c) shows an example of the generated GATE instance in VHDL.

Figure 11 presents the structural representation of a handshake wrapper for a specific soft IP. It uses the <WRAPPER> tag to concisely specify a handshake wrapper of the *core_6502* component with a common *clock* signal. The wrapped IP is specified using the <IP> tag and is located on a remote server specified using the XInclude tag.

```
void generate_gate(String func, int width, int delay) {                 (a)
//    generating VHDL entity of a gate ...
      println("ENTITY GATE IS");
      print("\t PORT (X1, X2: 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;");

//    generating VHDL architecture of a gate ...
      println("ARCHITECTURE MODEL OF GATE IS");
      println("\t BEGIN");
      print("\t\t Y <= X1 "+func+" X2 AFTER "+delay+" ns;");
      println("END MODEL;");
}
```

```
@ - Generic Interface                                                   (b)
$
"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;
```

```
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;                                                              (c)
```

Figure 10. a) VHDL metaspecification, b) Open PROMOL metaspecification and, c) VHDL instance

```
<WRAPPER type="handshake" clock="clk">
  <IP name="core_6502"
      xmlns:xi="http://www.w3.org/2001/XInclude">
    <XI:INCLUDE
      href="http://soften.ktu.lt/~damarobe/xml/free6502.xml"/>
  </IP>
</WRAPPER>
```

Figure 11. XML-based specification of a wrapper

In Figure 12, we present an example, how the markup technology can be applied to structurally represent the domain models. As an example, we take the interface description of the FIFO model in VHDL language (entity) (see Figure 12, a). The description has two hierarchical levels of structure: entity at the higher level and its I/O ports at the lower level. Furthermore, each port has the following attributes: port name, direction (in or out), and type (*bit, std_logic,* etc.).

When introducing the markup information into the domain model, each level of abstraction is replaced with the corresponding tag, which may be defined freely by the designer. For example, the domain language level can be marked by **<vhdl>** tag, the component interface level - by **<entity>** tag, and port level – by **<port>** tag. Furthermore, the attributes of the abstraction are replaced by the property of the tag and its value. The property of tag can be defined freely, whereas the value of the property must be the same as the attribute of the abstraction. For example, *entity FIFO* can be marked as *<entity name= "FIFO"/>*. The resulting structured representation of the FIFO model is given in Figure 12, b).

```
<?xml version="1.0"?>
<vhdl>
<entity name = "FIFO">
 <port name="D_In"  dir="in"  type="bit"/>
 <port name="Pop"   dir="in"  type="bit"/>
 <port name="Push"  dir="in"  type="bit"/>
 <port name="Reset" dir="in"  type="bit"/>
 <port name="Clk"   dir="in"  type="bit"/>
 <port name="D_Out" dir="out" type="bit"/>
 <port name="Empty" dir="out" type="bit"/>
 <port name="Error" dir="out" type="bit"/>
 <port name="Full"  dir="out" type="bit"/>
</entity>
</vhdl>                                             (a)
```

```
<xsl:for-each select="vhdl/entity">
entity <xsl:value-of select="@name"/> is
  port (
  <xsl:for-each select="port">
    <xsl:value-of select="@name" />:
    <xsl:value-of select="@dir" />
    <xsl:value-of select="@type" />
    <xsl:if test="position()!=last()">;
    </xsl:if>
  </xsl:for-each>
  );
end <xsl:value-of select="@name" />;
</xsl:for-each>                                     (b)
```

Figure 12. Markup of FIFO model:
a) structural XML-based representation of FIFO interface;
b) XSL template to generate a VHDL model

The obtained representation can be used as input to domain analysis and design tools. For example, it can be freely translated back to VHDL using XSL (see Figure 12, c). The same markup representation can be translated into different domain languages such as SystemC, which is especially important for modelling of complex embedded systems.

In our experiments, we have used several coarse-grained soft IPs such as ALUs, CPU cores, micro-controllers and microprocessors. The synthesis results show the following average increase in chip area of the generated wrappers with respect to the original soft IPs: 4-16% for Handshake wrappers, and 21-47% for FIFO wrappers, and the average increase in estimated power usage of the wrapped soft IPs with generated wrappers with respect to the original soft IPs: 26% for Handshake wrappers, and 39% for FIFO wrappers.

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 & Discussion

The presented Design Flow Models allow for introducing design automation for well-understood subdomains of HW design more effectively by using higher-level abstractions. We compare the presented integration models in Table 2 and evaluate them below.

Table 2. Comparison of design flow models

Model	Design methodology	High level abstraction	Low level abstraction	Applicability	Separated concerns	Restrictions	Specification type	Transformation model
1	Object-oriented	UML class diagrams	structural VHDL	Well-proven domain models	Structural	Subsets of UML and VHDL	Graphical	Metamodel, translation rules
2	Meta-programming	Meta-language	behavioral VHDL	Design families	Generic	Domain variability	Textual	Meta-specification
2	Parsing	Meta-language	VHDL, AST	Customization of soft IPs	Generic, analysis	Design context	Textual, data	Metaspecificati on, parsing
4	Markup	XML, XSL	VHDL	Representation & transformation of domain models	Structural	Domain content	Textual	Style sheet

(1) *UML-based model* allows for specifying a HW design problem at a higher abstraction level graphically. The design content is captured immediately and intuitively, thus increasing design comprehensibility. The level of abstraction is raised to the system level, which allows dealing with growing complexity of HW designs.

(2) *Metaprogramming-based model* concentrates on automatic generation of a particular component instance from a metaspecification and achieving larger reuse, because a metaspecification together with its processor is a program generator for a narrow domain.

(3) *Parsing-based model* allows for automatic analysis of design context and extraction of the application-specific information for further customization of the existing soft IPs.

(4) *Markup-based model* allows for convenient representation, access, management, and distribution of various types of HW/SW components and architectures. Furthermore, the designer has the ability to implement automatic modification of soft IPs and generation of a documentation using XSL. The systematic application of markup languages for HW design could increase design productivity, IP reuse and remote sharing as well as provide better documentation capabilities.

All proposed models focus on automatic generation of a target code, however, in a different way, thus suggesting different solutions. The basis of Model 1 is a relationship metamodel between UML and VHDL. The basis of Model 2 is a metaprogramming paradigm, linking, for example, Java as a metalanguage and VHDL as a domain language. Model 3 deals with analysis and representation of domain components as abstract syntax trees, further used for generating customized code. In Model 4, XSL style sheet is used for matching and transforming markup structures and generating VHDL code.

The analyzed HW design processes are applied at a level of abstraction above and before the traditional HW modelling, testing, synthesis and manufacturing processes. They greatly depend upon the results of domain analysis and are oriented at customization, transformation and "glass-box" reuse of HW models described using a high level language such as VHDL. As such, the research is very much in the area of SW engineering, though research objects are partly taken from the HW domain. The presented design processes have some similarity with traditional design processes implemented in HW synthesis tools as well as in SW compilers and interpreters in general as follows. (1) Programs are translated into other (usually lower-level abstraction) representations of programs. (2) Several commercial HW synthesis tools can also wrap their proprietary components with wrappers.

However, there are many differences as follows.

(1) The design methodology based on metaprogramming, which is presented in this paper, is independent of synthesis tools, proprietary technological libraries and HW description languages.

(2) Components taken from different IP providers can be customized, which is not the case with the known synthesis tools.

(3) Metaprogramming paradigm is independent of domain of application itself, which was proven by the presented experiments (the same methodology was used in HW, SW and embedded SW domains).

(4) Though increase in reuse and design productivity brought by the application of design processes and metaprogramming in a narrow well-defined domain is also useful for a designer of the end-product, the main benefactors may be for the creators and providers of reusable component repositories, who must quickly and cost-efficiently satisfy a plethora of various and continuously evolving requirements from their clients.

The comparison shows that the analysed technologies introduce an additional level of abstraction above the domain-level. Thus, there are some similarities in structural aspects of design.

However, the technologies aim to represent different views to the domain: metaprogramming – genericity, OOD – communication, and markup technology – structure. All these views are relevant to HW design. Thus, a designer may choose a certain abstraction to introduce a higher level of abstraction in a domain depending upon the specific requirements. A more promising approach would be to integrate these abstractions in a common design environment, i.e., to specify HW systems at a high level using UML diagrams, generate specific instances using the meta-programming techniques, and distribute and assemble HW components from different remote IP libraries using XML/XSLT.

Considering the presented evaluation of high-level abstractions and technologies, the following recommendations (see Table 3) can be given for a designer, who is focusing on a specific design aim in his work or implementing a particular design process.

Table 3. Recommendations of usage

Design process	Recommended technology	Reason for usage
Parameterisation	Metaprogramming	Provides more flexibility, higher modification capabilities and more convenient representation of parameters (no types, etc.).
Representation	Object-oriented	Objects better reflect the real-world features of the domain; UML provides graphical notation for high-level specification.
Abstraction	Markup	Markup provides capabilities for representing design context structurally and hierarchically.
Generalization	Metaprogramming	Metaspecification is specifically oriented at describing families of domain components.
Specialization	Object-oriented / Metaprogramming	Class hierarchies provide a convenient framework for specialization. Metaprogramming allows for generating specialized component instances.
Customisation	Metaprogramming with parsing / Markup	Smart customisation requires automatic analysis of design context using built-in parsing tools.
Generation	Metaprogramming	Describes the generation process explicitly.
Reuse	Metaprogramming / Object-oriented	Metaprogramming supports reuse via generation, and object-orientation – via polymorphism.
Separation of concerns	Metaprogramming / Object-oriented / Markup	All technologies support separation of concerns, though in a different way; particular selection must be made depending on a specific design task.
Variability	Metaprogramming	Metaprogramming abstractions, such as external functions, allow to flexibly expressing variability in a domain.

6. Conclusions

We have proposed a soft IP customization framework based on the concept of design process and metaprogramming. The framework is independent of specific HW synthesis tools, proprietary technological libraries and HW description languages, and allows implementing customization of soft IP components taken from different providers and sources. HW and SW design can be unified at a high level of abstraction using UML. Different high-level abstractions can be used in the same system design flow.

We have described four Design Flow Models for using UML, Design Patterns, Metaprogramming, Parsing, and XML/XSL style sheets in HW design. UML allows describing a system in an implementation-independent way. Design Patterns allow abstractly describing common design solutions. Metaprogramming allows describing families of design models in a generic way. XML/XSL allows for convenient representation, access, management, and distribution of various types of the structured domain information.

Depending on the particular requirements and availability of tools, the designer can select a proper higher-level abstraction and a design flow model to achieve higher design reuse, automation and productivity in narrow well-understood HW design sub-domains such communication control. Combinations of models can be used for the integration of several high-level abstractions within the same design framework to achieve higher separation of concerns in a design.

References

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

[2] **B. Liccardi, T. Maier-Komor, J.A. Oswald, M. El-kotob, G. Färber.** A Meta-Modeling Concept for Embedded RT-Systems Design. *Proc. of* 14*th Euromicro Conference on Real-Time Systems*, 19-21 *June* 2002, *Vienna, Austria*.

[3] **E. Aarts, R. Roovers.** IC Design Challenges for Ambient Intelligence. *Proc. of DATE* 03, *Munich, Germany*, 3-7 *March* 2003, 2-7.

[4] **H.P. Peixoto, M.J. Jacome, A. Royo, J.C. Lopez.** The Design Space Layer: Supporting Early Design Space Exploration for Core-Based Designs. *Proc. of the DATE*'1999, *Munich, Germany*, 9 -12 *March* 1999, 676-683.

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

[6] **F. Doucet, R.K. Gupta.** Microelectronic System-on-Chip Modeling using Objects and their Relationships. *Online Symposium for Electrical Engineers* (OSEE2000), 2000.

[7] **G. de Jong.** A UML-based design methodology for real-time and embedded systems. *Proc. of DATE* 2002, *Paris, France*, 4-8 *March* 2002, 776-778.

[8] **G. Martin.** UML for embedded systems specification and design: motivation and overview. *Proc. of DATE* 2002, *Paris, France*, 4-8 *March* 2002, 773-775.

[9] **B. Selic.** Architectural Patterns for Real-Time Systems. *L. Lavagno, G. Martin, B. Selic (Eds.), UML for Real, Kluwer Academic Publishers*, 2003, 171-188.

[10] **B.P. Douglass.** Fine Grained Patterns for Real-Time Systems. *L. Lavagno, G. Martin, B. Selic (Eds.), UML for Real, Kluwer Academic Publishers*, 2003, 149-170.

[11] **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), March 1997, 366-390.

[12] **E.A. Lee**. What's Ahead for Embedded Software? *IEEE Computer Magazine*, 33(9), *September* 2000, 18-26.

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

[14] **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*, *Vol.*19, 2002, 2-12.

[15] **A. Yakovlev, L. Gomes, L. Lavagno** (Eds.). Hardware Design and Petri Nets. *Kluwer Academic Publishers*, 2000.

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

[17] **M. Keating, P. Bricaud**. Reuse Methodology Manual for System-on-a-Chip Designs. *Kluwer Academic Publishers*, 1999.

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

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

[20] **R. Damaševičius, V. Štuikys.** Wrapping of Soft IPs for Interface-based Design Using Heterogeneous Metaprogramming. *INFORMATICA, Lithuanian Academy of Sciences*, 2003, *Vol.*14, *No.*1, 3-18.

[21] **R. Damaševičius, G. Majauskas, V. Štuikys.** Application of Design Patterns for Hardware Design. *Proc. of DAC'2003*, *June* 2-6, *Anaheim, CA, USA*, 48-53.

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

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

[24] **V. Štuikys, R. Damaševičius.** Soft IP Customization Model Based on Metaprogramming Techniques. *INFORMATICA, Lithuanian Academy of Sciences*, 2004, *Vol.*15, *No.*1, 111-126.

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

[26] **K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey, A. Sangiovanni-Vincentelli.** System Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Trans. on Computer-Aided Design* 19(12), 2000.

[27] **A. Sangiovanni-Vincentelli.** Platform-Based Design: A Path to Efficient Design Re-Use. *Proc. of the First Int. Symposium on Quality of Electronic Design* (ISQED 2000), 20-22 *March* 2000, *San Jose, CA*, 209-210.

[28] **F. Vahid, T. Givargis.** Platform Tuning for Embedded Systems Design. *IEEE Computer*, 34(2), 2001, 112-114.

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

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