Application of Design Patterns for Hardware Design

Robertas Damaševičius, Giedrius Majauskas, Vytautas Štuikys
Kaunas University of Technology, Software Engineering Department
Studentų 50, 3031-Kaunas, Lithuania, Ph: (370 37) 300 399
E-mail: {damarobe, giedmaja}@soften.ktu.lt, vystu@if.ktu.lt

ABSTRACT
Design patterns, which encapsulate common solutions to the recurring design problems, have contributed to the increased reuse, quality and productivity in software design. We argue that hardware design patterns could be used for customizing and integrating the Intellectual Property (IP) components into System-on-Chip designs. We formulate the role of design patterns in HW design, and describe their implementation using metaprogramming. We propose a Wrapper design pattern for adapting the behavior of the soft IPs, and demonstrate its application to the communication interface synthesis.

Categories and Subject Descriptors
B.5.2 [**Hardware**]: Register-Transfer-Level Implementation | Design; D.2.2 [**Software**]: Software Engineering | Design Tools and Techniques.

General Terms
Design.

Keywords
Design patterns, system-level design processes, wrapping, UML, metaprogramming.

1. INTRODUCTION
Ever-growing complexity of hardware (HW) design is a great challenge for a designer. It is nearly impossible to create the System-on-Chip (SoC) design from scratch and ensure its quality in a reasonable time. The reuse of the predefined Intellectual Property components (IPs) is only a partial solution to this problem, because the designer usually has to modify the IPs or to write the glue code in order to integrate the IPs. This manual work is usually time-consuming and error-prone. Thus, the HW designers are seeking to adapt the solutions for large-scale design problems from other domains, such as software (SW) design.

One of the solutions for describing the structural or behavioral relationship between components in SW design is *design patterns* [1]. These are used to abstract and encapsulate common design solutions as well as to describe contexts to which they can be applied in a language-independent way.

DAC 2003, June 2-6, 2003, Anaheim, California, USA.

The benefits of using design patterns in HW design can be stated as follows. (1) Describing a HW system in an abstract and implementation-independent way can significantly raise the level of abstraction. (2) Using the standard UML [2] diagrams eases the communication between different design teams. (3) Using the already developed object-oriented (OO) design and testing methodologies can ensure higher HW design quality. (4) Using graphical design tools, catalogues of design patterns, and automatic code generation tools can significantly increase HW design automation and productivity as well as accelerate design reuse, sharing and transfer.

Although the OO design techniques may seem foreign to HW designers, many of the concepts and principles are similar. For example, the development of SW systems using the classes from a reuse library is comparable to SoC design using the IP blocks; SW classes communicate with messages, and HW blocks communicate with signals, etc.

The aim of this paper is to consider the application of design patterns for *system-level* HW design, and their contribution to managing the design complexity problem, raising the level of abstraction and ensuring higher design quality.

Our contribution is as follows. (1) We analyze the role of design patterns and problems of their application in HW design. (2) We define a new HW design pattern, called *Wrapper*, for adapting the interface and behavior of an IP to the context of the usage, and describe it using UML. (3) We propose to implement the HW design patterns using metaprogramming. (4) We demonstrate an application of the Wrapper design pattern for the communication interface synthesis.

The structure of the paper is as follows. In Section 2, we review the related work. In Section 3, we analyze the role of design patterns in HW design. In Section 4, we propose the Wrapper HW design pattern. In Section 5, we present a metaprogramming-based scheme for implementing design patterns in HW design. In Section 6, we present a case study. In Section 7, we evaluate the results and present a discussion. Finally, we conclude in Section 8.

2. RELATED WORK
Several authors consider the problem of adapting the OO design concepts and, particularly, the application of design patterns for HW design. Kumar *et al.* [3] acknowledge the importance of the OO modeling techniques for improving the HW design process. Some of the advantages are the improved modifiability of models, quick composition of new components, and ability to identify, reuse and specialize common components. Nebel and Schumacher [4] analyze the OO modeling techniques as a means to increase productivity in HW design. The largest possible gain is expected when the subjects of the OO design are not the physically existing objects (e.g., gates), but *abstract concepts* for solving the design

problems. Yoshida [5] analyzes the applicability of the known SW design patterns to SoC design. He uses an *Abstract Factory* pattern for design parameterization, and a *State* pattern to encourage FSM (Finite State Machine) state reuse and extension. The author concludes that some SW design patterns can be applied for HW design, however, further research is needed to discover new HW design patterns. Doucet and Gupta [6] present a methodology, which uses design patterns to capture the *Models of Computation* (abstractions of design functionality) formally in the context of the system-level HW design. Two HW design patterns are presented: a *Bus-Protocol* pattern for specifying on-chip bus structures and associated protocol behaviors, and a *DLX Processor Architecture* pattern for describing the architecture of the pipeline processor. Vanmeerbeeck *et al.* [7] use design patterns to describe the inter-process communication, and present a *Resource Manager* pattern for managing all access requests from different processes to a specific resource. Åström *et al.* [8] demonstrate how SW design patterns can be applied to HW design. Four design patterns are considered: *Composite, Object Adaptor, Abstract Factory* and *Decorator,* which are used to design a C++ based library of DSP models.

Recently, SystemC [9] has emerged as a C++ modeling platform for HW design. As an extension of C++, SystemC allows the usage of the OO modeling techniques (including design patterns). For example, Charest and Aboulhamid [10] use a *Singleton* design pattern to deal with configurability in HW designs. However, SystemC does not encourage the usage of inheritance and other OO techniques when designed for synthesis [11].

Another related area of research is *platform-based design* [12]. Generally, design patterns and platforms have the same aims, i.e., to describe a common architectural solution and to allow the reuse of HW and SW components. However, design patterns are based on the principles of the OO design, whereas platforms are developed using the principles of the component-based design.

UML also has attracted a considerable amount of attention from the researchers and designers. For example, Fernandes *et al.* [13] overview the UML diagrams, and present how UML can be used to model embedded control systems. Martin [14] discusses the capabilities and lacks of UML for embedded system design, and formulates the requirements for future extensions to UML to support the platform-based HW design. Chen *et al.* [15] consider the requirements to modeling SoC platforms in UML, and developing the OO methodologies for embedded system design. Zhu *et al.* [16] propose a design methodology for SoC based on UML and C++/SystemC. UML is used as a modeling language extended for parallelism, structure, and timing. The authors report the reduction of the design time by about one-third compared to the conventional methods.

The summary of the related works is as follows. The authors (1) emphasize the importance of the structuring, encapsulation and reuse of HW designs at the highest levels of abstraction, (2) suggest using the OO modeling techniques, including design patterns and UML, for HW and embedded systems design, and (3) seek to discover and describe HW design patterns.

Our approach has some similarities to [8]. However, there are substantial differences as follows. (1) We focus on discovering HW design patterns, rather than adapting the already known SW design patterns. (2) We apply our approach at the system-level of abstraction. (3) Our approach is a language-independent one, therefore, we can use a standard HDL (e.g., VHDL) rather than C++/SystemC, thus we can create a synthesizable design. (4) The proposed HW design pattern is a generic solution for several application domains. (5) We use the metaprogramming-based code generation to implement the HW design patterns, which allows us achieving higher design quality and productivity.

3. ROLE OF PATTERNS IN HW DESIGN

3.1. Recent Shifts in HW Design

The reuse of the pre-designed IPs, such as µPs, DSPs, RAMs, etc., is essential for SoC design success. The combination of the large and complex IP blocks and embedded SW in SoC is precipitating a fundamental shift from the content-based HW design to the integration-based one. The traditional *content-based* design approach is focused on creating the original design content from scratch and verifying it.

By contrast, the *integration-based* approach shifts from the content creation to the problems of evaluating, customizing, and integrating multiple IPs developed by independent IP vendors into SoC designs. For these designs, describing how and what IP blocks will be utilized, how system functionality will be partitioned between HW and SW parts, and how the components will be interconnected and verified, is the primary focus of a designer. This requires more in-depth system-level design, modeling and reuse methodologies and techniques, HW/SW co-design and verification tools, qualification at all levels of the design process, including the usage of the third party IP providing and design consulting services.

3.2. HW Design Processes

Reflecting the recent shifts in HW design, the HW design processes are categorized as follows (see Figure 1):

(1) *Register Transfer-level* (RTL) design processes are the lower-level *content-based* processes, which are concerned with IP design from scratch. They involve a manual programming in a HDL, as well as bit-level modeling, testing and synthesis.

(2) *System-level* (SL) design processes are the higher-level *integration-based* processes, which are concerned with the assembly of the HW systems from the IP blocks. They involve a variety of design activities, which enable IP reuse, customization and integration. As complexity of the designed systems is constantly increasing, the SL processes are becoming increasingly important in HW design.

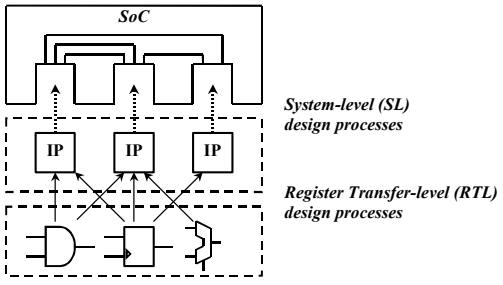

Figure 1. HW design processes.

We categorize the SL design processes as follows: (1) *Specification* processes deal with the analysis and specification of design problems, e.g., separation of concerns, composition and generalization. (2) *Implementation* processes are concerned with

the aspects of implementation of design solutions that deal with design problems, e.g., customization and wrapping of soft IPs.

Separation of concerns is a process of finding and isolating the different aspects of system's functionality. When concerns are implemented separately, we can derive different variants of a system by configuring and integrating the separated concerns, as well as reuse the concerns in another context of application.

Composition is a process of gluing the separate pieces of the existing code into a system. We distinguish two types of composition: (1) *logical* composition, when components are connected via their interfaces (ports), and (2) *physical* one, when domain programs are composed from the HDL statements using inlining or another language-specific mechanism.

Generalization is a process of deriving a generic specification, which describes a family of the 'look-alike' components, where variations of the domain functionality are represented at a higher level of abstraction. Generalization encapsulates the multi-aspect view (e.g., functional, architectural, etc.) to the component, enhances its reusability and increases applicability. We implement generalization using the higher level (generic/meta) abstractions usually aiming at concise expressing of the different design aspects, and widening a context of the usage.

Customization is a design process, which changes the characteristics of the IP without modifying its original functionality (architecture). *Wrapping* is a particular case of customization when additional functionality is added to the basic IP functionality in order to adapt it to the context of the usage.

3.3. Motivation for Using Design Patterns

The existing HW design and reuse methodologies (e.g., [17]) are mostly dealing with the RTL design styles and coding rules. This accumulated experience and lessons learned in the design practice are very important and useful when achieving design reuse. However, the SL design issues are not addressed properly. The application of the OO techniques for the SL design can overcome this gap. Though, design patterns can be used for modeling the RTL design processes in HW design, we suggest that their main application should be the SL design. However, the role of the design pattern as "encapsulated design experience" is too general and vague, and needs a rectification for HW domain. In our view, the HW design patterns should be used to describe the *commonly used SL design processes*. In contrast, the SW design patterns are used to describe the common SW architectures.

Design patterns especially could contribute to ensuring the higher HW design quality. The quality of a HW system can be understood as a combination of the quality of the third-party IPs, which compose the system, and the quality of the SL design processes, which are used to integrate the IPs. We assume that the IP providers ensure the quality of the IPs. However, the HW designer still must ensure the quality of the SL design processes, which can depend upon many factors such as personnel skills, management models, availability of tools, etc., but the most important factor is *design methodology*. Design patterns have increased the quality of SW design [1]. We hope that the application of design patterns to the SL HW design processes will lead to the higher quality of HW designs, too.

Other contributions of design patterns include (1) managing of design complexity through the usage of the UML diagrams, and (2) raising the level of abstraction above the HDL level.

3.4. Examples of Application of Design Patterns and Wrappers in HW Design

Although the term itself is not frequently used in HW design, we can describe some widely used non-formal design patterns, such as models of computation, data flow models, communication models, wrappers, etc. For example, FSM is a well-known solution for the control-based applications. The designers usually implement FSMs using the *State* design pattern [5]. This solution gives the following benefits. (1) The designer can use well-known optimization methods to reduce the chip area. (2) There are plenty of validation solutions available.

For system composition, several other design patterns are used, such as shared busses and communication coprocessors (bridges) [18]. The *bus pattern* is used to share communication medium between several units effectively. The *communication coprocessor pattern* is used to separate the computation and communication tasks in the system.

Wrappers are already used in a number of applications. However, they were not described as a design pattern, yet. Some of the examples of application of wrappers are as follows:

(1) A *reliability wrapper* is used for the reliability-critical applications in order to determine HW faults and reduce the probability of an erroneous output. The wrapper provides an implementation of a majority voter, error detection/avoidance circuits, and self-repair circuits. This approach is used in [19] to generate the fault-tolerant embedded processors.

(2) A *bus wrapper* is used for the communication synthesis in SoC designs. The wrapper provides an implementation of a particular data protocol for communication with other components. This solution is used in VSIA's *Virtual Socket Interface* methodology [20] to connect the IPs to on-chip buses.

(3) A *protocol wrapper* is used for layered Internet packet processing. The wrapper provides an implementation of an OSI protocol layer. This solution is used in the FPX networking platform [21] to simplify and streamline the implementation of the high-level networking functions by abstracting the operation of the lower-level packet processing functions.

(4) A *memory wrapper* is used for the automatic adaptation of the physical memory interfaces to a communication network that may have a different number of access ports. The wrapper provides an implementation of a memory controller, access manager, internal communication bus, arbiter and other memory-specific control logic. This approach is used in [22] to facilitate the integration of the standard memory components into SoC designs.

3.5. Adaptation of UML for HW Design

Much of the difficulties when applying the OO modeling techniques to HW design are related to the lack of the standard OO HDL. Although there are efforts in developing the OO extensions of the standard HDLs (e.g., SUAVE [23]), or adapting the existing OO languages to HW design (e.g., SystemC [9]), most of HW designers are still using the standard HDLs (such as VHDL, Verilog). This can be explained by the fact that (1) most of the existing soft IPs were developed using a standard HDL, and (2) there are few efforts to directly synthesize the OO concepts to RTL. Therefore, we need to agree how we map the OO concepts to the abstractions of the non-OO HDL, such as VHDL (see Figure 2).

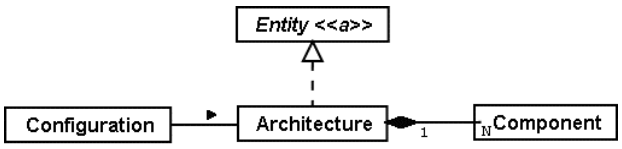

Figure 2. Application of the OO concepts for VHDL.

We consider that an *abstract class* corresponds to the VHDL *entity*. A class that implements an abstract class corresponds to the VHDL *architecture*. The configuring and instantiation of the class instances (objects) corresponds to the VHDL *configuration*. Class *attributes* correspond to the VHDL *ports (public)* and *signals (private)*, and class *methods* – to the different VHDL *processes*. The *composition* relationship describes the composition of a system from the components and corresponds to the VHDL *port map* statement. The *inheritance* relationship means that a VHDL entity inherits the I/O ports.

4. WRAPPER DESIGN PATTERN

In this section, we propose a new HW design pattern, called *Wrapper*. A description of the design pattern is a document composed of plain text descriptions, UML charts and sample codes. Here, we describe the Wrapper design pattern using a common description scheme [1].

[Intent] Wrapper allows adapting an interface and behavior of the IP component to the context of a given application.

[Applicability] Use Wrapper when you need to adapt the component to the requirements of its environment.

[Structure] For an UML *Class* diagram, see Figure 3. Entity *Wrapper* inherits the I/O ports of the entity *IP,* and declares new I/O ports for the *Wrapper* functionality. Architecture *IPModel* implements the functionality of the *IP*. Architecture *WrapperModel* implements the functionality of the *Wrapper* and contains component *IP*. Essentially, this description means that *WrapperModel* wraps *IPModel* with new *Wrapper* functionality.

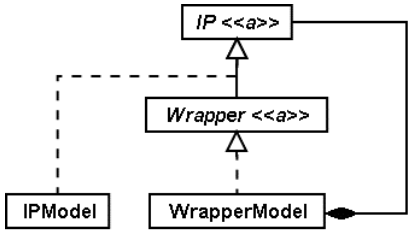

Figure 3. Wrapper design pattern.

[Consequences] Wrapping can be nested, i.e., we can apply this design pattern to the IP again and again. Many different wrappers can be applied to the same component, as well as the same wrapper can be applied to the different IPs. However, a designer must be cautious for an area and delay overhead, which can be introduced by the wrapper component.

5. IMPLEMENTATION OF DESIGN PATTERN USING METAPROGRAMMING

It is possible to generate the code representing a design pattern automatically, if the domain is well defined and formalized. There are several examples of the automatic generation of similar wrappers described in the literature (e.g., [18, 22]). Generally, we have (1) to extract the environmental parameters of a design, and (2) apply the generative techniques for the code generation. This can be accomplished using a specialized code generator, metalanguage, or the internal HDL capabilities for generic programming (e.g., *generics* in VHDL), etc.

We implement HW design patterns using the *metaprogramming* (MPG) techniques [24, 25]. In general, MPG is a higher-level programming technique, which provides a means for manipulating with other (domain) programs as data. The *heterogeneous* MPG is based on the usage of two different languages in the same generic specification. The lower-level language (*domain language*) is used for expressing basic domain functionality. The higher-level language (*metalanguage*) is used for expressing generalization and describing domain program modifications.

The main aim of MPG is to create a *metaprogram* – a program generator for a narrow application domain. The metaprogram consists of a family of the related domain program instances encapsulated with their modification algorithm, which describes the generation of a particular instance depending upon the values of the generic parameters. A designer uses a metalanguage as a higher-level abstraction to integrate together the different domain program instances and make up a metaprogram. The metaprogram is then used as a set of instructions for a metalanguage processor to generate the specific domain program instances.

We implement a particular design pattern (e.g., Wrapper) by developing a parameterized code generator (metaprogram) that applies a respective design process (e.g., wrapping) to a given soft IP (see Figure 4). The role of MPG is to serve as a bridge between the abstract description of the SL design process and its implementation, as well as to provide a means and guidelines for developing domain code generators. So far, the MPG step (the development of the generator) is performed manually. However, the wrapper generation is performed automatically for any soft IP.

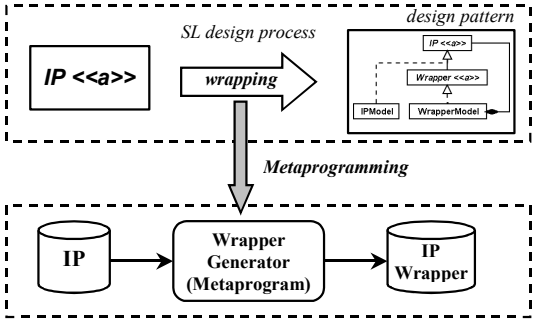

Figure 4. Relationship between design process, design pattern and metaprogramming.

We demonstrate the application of the Wrapper pattern for HW design, including the automatic generation of the VHDL code, in our case study.

6. CASE STUDY

In this case study, we deal with the problem of *interface synthesis* [26], i.e., the generation of interfaces between the communicating IPs. We apply a Wrapper design pattern to generate the handshake wrappers using a single-rail 4-phase handshake data protocol [27]. The data transmission scheme is as follows. The IP communicates with a micro-controller (MC), which drives the IP through a channel (bus) using a handshake data protocol (DP) (see Figure 5,

a). Our aim is to adapt the IP for handshaking in order to establish a direct communication between the MC and the IP as it is shown in Figure 5, b. For this, we need to modify the IP interface and wrap the IP with a control logic. Note that in this case study we generate only a client side of the application. The wrapped IP is now ready for integration into a larger-scale HW system.

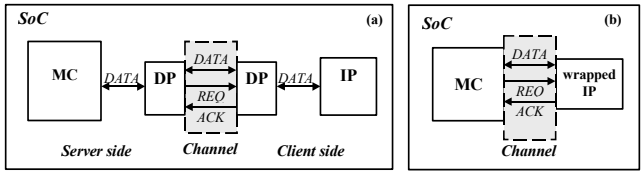

Figure 5. Data transmission scheme using handshaking: a) before, and b) after interface synthesis.

We have constructed a wrapper generator to automatically generate the handshake wrappers for the third-party soft IPs described in VHDL. The handshake wrapper was implemented as a generic metaprogram parameterized by the parameter values, which are extracted from the interface of the soft IP. The wrapper generator uses a VHDL parser [28] to analyze the given soft IP interface, and several Java (metalanguage) classes to generate VHDL (domain language) code according to the Wrapper design pattern. The wrapper generator performs wrapping by generating an instance of the handshake FSM from a generic metaprogram, and a *port map* statement, which maps signals of the handshake wrapper to the IP. The generation process is fully automatic. The architecture (UML *Class* diagram) of the handshake wrapper is given in Figure 6. To describe the behavior of the wrapper, we have used other UML diagrams such as *State* diagram.

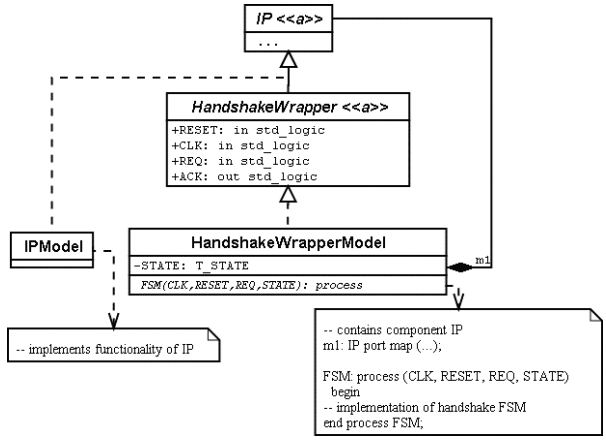

Figure 6. Architecture of the handshake wrapper.

We have performed the experiments using the third-party soft IPs as follows: (1) Dragonfly micro-core [29], (2) HC11 CPU [30], and (3) i8051 micro-controller [31]. Synthesis results (Synopsys; CMOS 0.35 um technology) for the original third party IPs and the generated wrapper components (Wr) are presented in Table 1.

Table 1. Synthesis results

IP	Area, cells (IP)	Area, cells (Wr)	Increase, %	Power, uW (IP)	Power, uW (Wr)	Decrease, %
Dragonfly	5883	6804	16	19.9421	14.6646	26
HC11	7394	8951	21	19.0632	14.6237	23
i8051	24266	25282	4	50.5518	33.8819	33

The synthesis results show an increase in chip area (~14%), and a moderate decrease in the estimated power usage (~27%). This decrease can be explained by the fact that power is not used when data is not transmitted to the IP.

7. EVALUATION AND DISCUSSION

The application of the concept of design patterns for HW design allowed us to better understand the domain, and to specify the system-level HW design processes and target architectures in a semi-formal way. Faced with a problem of interface synthesis, we have introduced and successfully applied the Wrapper design pattern. To automatically apply the pattern, we have developed the metaprogramming-based VHDL code generator, which generates a handshake wrapper for any given soft IP virtually in seconds, thus substantially increasing design productivity.

The advantages of using design patterns in HW design are as follows. (1) The design content is captured immediately and intuitively, thus increasing design comprehensibility. (2) The pattern-based design can be easily supported by the automated validation and code generation tools, thus increasing design reuse, quality and productivity. (3) The level of abstraction is raised to the system level, which allows dealing with growing complexity of HW designs.

Additionally, the usage of design patterns may reduce the gap between the development of SW and HW parts of the SoC, as the object-oriented 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.

The idea of applying design patterns to HW design holds great promises; however, there are many gaps, which must be bridged before these promises could be fulfilled. (1) *Conceptual gap*: SW designers think in terms of the OO design (objects and messages), whereas HW designers are used to think in terms of the component-based design (components and wires). (2) *Methodological gap:* how the SL design processes described using the UML-based HW design patterns could be (semi-) automatically transformed into metaprograms? (3) *Physical gap*: how the physical constraints (such as the timing ones) should be reflected in an OO model (design pattern)? (4) *Technological gap*: how objects or even entire design patterns could be directly synthesized to RTL?

We hope that the recent trends towards blurring the boundaries between SW and HW domains (e.g., HW/SW co-design, SystemC, etc.) will overcome some of these problems, too.

8. CONCLUSIONS AND FUTURE WORK

The application of the object-oriented modeling techniques, including design patterns, has only recently became a hot topic in HW design. In this paper, we analyzed the role of design patterns in system-level HW design, and proposed a new HW design pattern, called Wrapper, for adapting the behavior of the IPs. It can be used in a variety of HW design applications from the fault-tolerant design to the communication synthesis. In addition, we have discussed the implementation of HW design patterns using the heterogeneous metaprogramming techniques. We expect that

the application of design patterns will contribute to the increase in HW design reuse, automation, quality and designer productivity.

Future work will focus on the discovery of other HW design patterns and the development of the HDL code generators for their implementation.

REFERENCES

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

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

[3] S. Kumar, J. H. Aylor, B. W. Johnson, and W. A. Wulf. Object-Oriented Techniques in Hardware Design. *IEEE Computer*, Vol. 27, No. 6, June 1994, pp. 64-70.

[4] W. Nebel, and G. Schumacher. Object-Oriented Hardware Modelling – Where to apply and what are the objects? In *Proc. of EURO-DAC with EURO-VHDL'96*, September 16-20, 1996, Geneva, Switzerland, pp. 428-433.

[5] 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.

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

[7] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, and I. Bolsens. Hardware/Software Partitioning of Embedded System in OCAPI-xl. In *Proc. of the Ninth Int. Symposium on Hardware/Software Codesign (CODES'2001)*, April 25-27, 2001, Copenhagen, Denmark, pp. 30-35.

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

[9] T. Grőtker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, Boston, 2002.

[10] L. Charest, and E. M. Aboulhamid. A VHDL/SystemC Comparison in Handling Design Reuse. In *Proc. of 2002 Int. Workshop on System-on-Chip for Real-Time Applications (IWSOC 2002)*, July 6-7, 2002, Banff, Canada, pp. 79-85.

[11] S. Virtanen, D. Truscan, and J. Lilius. SystemC Based Object Oriented System Design. In *4th Int. Forum on Design Languages (FDL'01),* September 3-7, 2001, Lyon, France.

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

[13] J.F. Fernandes, R.J. Machado, and H.D. Santos. Modeling Industrial Embedded Systems with UML. In *Proc. of the 8th Int. Workshop on Hardware/Software Codesign (CODES 2000)*, May 3-5, 2000, San Diego, CA, USA, pp. 18-22.

[14] G. Martin. UML for Embedded Systems Specification and Design: Motivation and Overview. In *Proc. of DATE'2002*, March 4-8, 2002, Paris, France, pp. 773-775.

[15] R. Chen, M. Sgroi, L. Lavagno, G. Martin, A. Sangiovanni-Vincentelli, and J. Rabaey. Embedded System Design Using UML and Platforms. In *Forum on Specification and Design Languages (FDL'2002)*, September 24-27, Marseille, France.

[16] Q. Zhu, A. Matsuda, S. Kuwamura, T. Nakata, and M. Shoji. An Object-Oriented Design Process for System-on-Chip using UML. In *Proc. of the 15th International Symposium on System Synthesis (ISSS 2002),* Kyoto, Japan, pp. 249-254.

[17] M. Keating, and P. Bricaud. *Reuse Methodology Manual for System-on-Chip Designs.* Kluwer Academic Publishers, Boston, 2001.

[18] D. Lyonnard, S. Yoo, A. Baghdadi, and A. Jerraya. Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip. In *Proc. of DAC 2001*, June 18-22, Las Vegas, Nevada, pp. 518-523.

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

[20] G. Cyr, G. Bois, and M. Aboulhamid. Synthesis of Communication Interface for SoC using VSIA Recommendations. In D. Sciuto (ed.), *DATE 2001 Designer's Forum,* March 13-16, 2001, Munich, Germany, pp. 155-159.

[21] F. Braun, J. Lockwood, and M. Waldvogel. Protocol Wrappers for Layered Network Packet Processing in Reconfigurable Hardware. *IEEE Micro*, Vol. 22, No. 3, February 2002, pp. 66-74.

[22] F. Gharsalli, S. Meftali, F. Rousseau, and A. A. Jerraya. Automatic Generation of Embedded Memory Wrapper for Multiprocessor Soc. In *Proc. of DAC 2002*, June 10-14, 2002, New Orleans, Louisiana, USA, pp. 596-601.

[23] P. J. Ashenden. Object-Oriented Extensions to VHDL. In *Int. Conference on Chip Design Automation (ICDA 2000),* August 21-25, 2000, Beijing, China.

[24] V. Štuikys, R. Damaševičius, G. Ziberkas, and 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,* pp. 257-266. Kluwer Academic Publishers, Boston, 2002.

[25] R. Damaševičius, and V. Štuikys. Wrapping of Soft IPs for Interface-based Design Using Heterogeneous Metaprogramming. *INFORMATICA*, 14 (1), 2003, pp. 3-18.

[26] A. Rajawat, M. Balakrishnan, and A. Kumar. Interface Synthesis: Issues and Approaches. In *Proc. of the 13th Int. conference on VLSI Design*, January 3-7, 2000, Calcutta, India, pp. 92-97.

[27] A.M.G. Peeters. *Single-Rail Handshake Circuits*. PhD. Thesis, Technische Univ. Eindhoven, the Netherlands, 1996.

[28] Andreas Dangberg (C-Lab). VHDL Parser, 1997, http://home.wtal.de/software-solutions/vhdl-parser/

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

[30] Green Mountain Computing Systems, Inc. HC11 CPU core, 2000, http://www.gmvhdl.com/hc11core.html

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