ISSN 1392 – 124X INFORMACINĖS TECHNOLOGIJOS IR VALDYMAS, 2004, Nr. 1(30)

A SUBSET-BASED COMPARISON OF MAIN DESIGN LANGUAGES

Robertas Damaševičius

Software Engineering Department, Kaunas University of Technology
Studentų 50, LT-3031, Kaunas, Lithuania

Abstract. Different design languages are used for hardware and software domains. The most popular ones are VHDL and C++. Recently, SystemC language was developed aiming to bridge both domains. In this paper, we compare these design languages by separating and analysing the capabilities of the different language subsets with respect to the implementation of the different programming paradigms. Our analysis illustrates that the considered design languages show a great deal of similarity, despite different domains of application. We demonstrate the usage of the language subsets in a case study (design of a generic calculator model).

Key words: domain analysis, design language, language subset, VHDL, SystemC.

1. Indroduction

With the arrival of complex Systems-on-Chip (SoC), the system designers often have to deal with the hardware (HW) and embedded software (SW) parts in the same design. These parts are usually designed using different design languages and methodologies, which requires application of the multi-language design approaches [1] and multi-language specifications [2], and raises a variety of problems such as language inter-operability, distributed design, co-simulation, validation, etc. As the complexity of HW designs continues to increase, there are many signs that HW and SW development methods are beginning to converge. There is a great interest in a HW design community to compare the existing design processes, technologies and abstractions in HW and SW domains in order to share their achievements.

Many different languages were developed for SW and HW design. Currently the standard HW description languages (HDLs) are VHDL [3], an offspring of ADA, and Verilog [4], which has a C-like syntax. Research continues in the development of new HDLs or extensions of the existing languages, e.g., Handel-C [5], HML [6], JHDL [7], Objective-VHDL [8], OpenJ [9], Rosetta [10], SDL [11], SpecC [12], SUAVE [13], Superlog [14], SystemC [15], SystemC^{SV} [16].

These developments either (1) reflect the recent trend towards blurring the boundaries between HW and SW domains through the usage of a single language (SpecC, SystemC), or (2) aim at enriching the HW design domain with the object-oriented concepts (Objective VHDL, SUAVE), or (3) introduce the domain-specific abstractions (SystemC^{SV}), or (4) aim at the system-level design (Rosetta, Superlog, SDL).

The implementation of the languages can be categorised as follows. (1) Adding new keywords to the existing languages such as C++ (Handel-C), Java (JHDL, OpenJ) or VHDL (Objective VHDL, SUAVE) for supporting the HW description at a high level. (2) Providing a basic set of HW primitives that can be easily extended into a higher level support (HML, SDL). (3) Providing class libraries that implement support for HW modeling (SystemC, SystemC^{SV}).

When languages are constructed on the top of the existing ones (which is usually the case), this leads to the stratification of languages according to the different subsets (levels, layers) introduced for various purposes by the different language development teams at a different time. For example, C was developed in the early 1970s for the algorithmic programming. Later, the object-oriented constructs were added to C in the early 1980s and C++ was born. Recently, C++ was extended for HW design to become SystemC.

The aims of this paper are (1) to analyse subset by subset the similarities and differences of the design languages from the different application domains, and (2) to demonstrate the usage of the analysed subsets.

The structure of the paper is as follows. We review the related works in Section 2. We analyse the subsets of the selected design languages (VHDL, C++, and SystemC) in Section 3. We present a case study in Section 4. Experimental results are given in Section 5. We present a discussion in Section 6. We finish with conclusions and future work in Section 7.

2. Related works

The analysis of domain languages or their specifications is a common activity in SW engineering. For example, Coplien [17] considers C++ as a language, which implements the procedural, modular, object-oriented and generic programming paradigms.

Gabrielli *et al.* [18] decompose VHDL into four abstraction levels: behavioural, data-flow, structural and mixed ones. The behavioural level describes the behaviour of a system without considering other constraints. The data-flow level deals with the flow of signals and data between the HW blocks. The structural level deals with the allocation of the components.

Allen and Gajski [19] decompose HW specifications into four levels: algorithmic, modular, cycle-accurate, and Register-Transfer Level (RTL). The algorithmic level specifies only the behaviour of a design, with no specific implementation details. At the modular level, the design is partitioned into components that communicate through a clearly specified protocol. Cycle accuracy introduces the notion of a clock and a time at which the events occur, but without specifying the implementation details of the events. RTL specifies the implementation of events without relying on a particular implementation technology. However, the algorithmic and modular levels are common to SW specifications, too. The notion of events and concurrency are also used in the object-oriented design.

Another cluster of papers deals with the comparison of design languages. We consider only some of them below. Li and Leeser [6] compare VHDL and HML with respect to the understandability of syntax, synthesisability, and type checking capabilities. Smith [20] compares VHDL and Verilog languages with respect to parameterisation capabilities, data types, design reusability, code readability and facilities for describing the synthesizable constructions. Gerstlauer *et al.* [21] compare VHDL, VHDL+ and SpecC with respect to the language features such as support for the state machines, exception handling, and communication abstractions. Bunker *et al*. [22] compare Objective VHDL, SpecC and Java with respect to the availability of the automatic verification techniques, integration with the existing design practice, formality of semantics, and learning curve. Bezerra [23] compares Java Forge, VHDL and Handel-C with an emphasis on the quality of the developed designs.

Recently, SystemC attracted much attention of the researchers as a new HW modeling language, and caused the need to compare it with other languages in the domain. For example, Bonanome [24] presents a comparison of Verilog and SystemC based on the existence or absence of constructs, timing mechanisms, built-in data types, simulation semantics and language determinism. Agliada *et al.* [25] compare VHDL and SystemC on the statement level in the context of VHDL to SystemC conversion. Charest and Aboulhamid [26] compare the capabilities of VHDL and SystemC to express domain commonalties and variations, to describe regular structures and to handle design reuse.

Our contribution to the field of domain analysis is as follows. We decompose the abstractions of the analysed design languages (VHDL, C++, SystemC) into different subsets and compare the capabilities of the languages subset by subset. We give a particular emphasis on the *generic/template* subset of the analysed languages.

3. Analysis of subsets in design languages

3.1. Principles of analysis

Design languages have many common features, which reflect the achievements in SW engineering and programming technologies at the time of their creation, as well as many differences related to the different domains of application. Our aim is to compare the design languages by analysing the relationship of the languages and roles of their subsets. We consider C++ as the principal language for SW design, VHDL - for HW design, and SystemC - for the system-level SW/HW modeling and co-design. The relationship between these design languages can be summarised in Figure 1. C++ is dedicated purely for SW domain. VHDL is dedicated for HW domain, however it has some SW background inherited from its parent language, ADA. SystemC is an extension of C++ with HW design concepts and can be used for both domains.

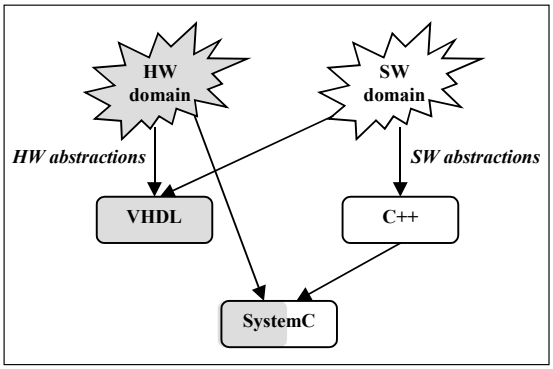

Figure 1. Relationship of design languages

A more general problem is how we establish the criteria for the comparison of the languages. One can compare the language features, capabilities, abstractions, etc. However, we apply an approach, which was also used by Coplien [17], i.e., we analyse the languages with respect to their suitability to express the concepts of the different programming paradigms. We apply the principle of the *multi-dimensional separation of concerns* [27] in our analysis below. We separate different subsets in each considered language. These subsets support different issues in SW engineering and various programming paradigms (algorithmic, structural, component-based, object-oriented, meta-

programming). We believe that by establishing the roles and relationships of their respective subsets, we can compare the languages as follows.

3.2. VHDL

VHDL is an IEEE standard HW description language [3]. It can describe the behaviour and structure of the electronic systems, but is particularly suited as a language to describe the structure and behaviour of the digital electronic HW designs. Below, we present a brief overview of VHDL language.

The design entity is the primary HW abstraction in VHDL. An *entity* is an interface, which abstracts corresponding implementations in one or more *architectures*. The architectures can contain signals, processes and instantiations of other entities. A *process* consists of a sequence of statements, which are executed sequentially, whereas the processes themselves are executed concurrently. A process may have some local *variables* to store its state. The *signals* are used for communication between the processes. VHDL currently supports definition of a weak form of abstract data type, using *types* and operations defined in *packages*. Further details can be found in [28, 29].

We separate the subsets of VHDL according to the different programming paradigms the language supports as follows:

(1) *Behavioural subset* is a subset for describing the HW design-specific concepts at the RTL, such as signals, data types (*bit, std_logic,* etc.), logical operators (*and, or, xor,* etc.). These are directly mapped to the technology-dependent HW objects, such as flip-flops and logic gates, during synthesis.

(2) *Structural subset* is a subset for expressing the structural programming concepts, such as *for* loop, *if* and *case* statements, which are used to describe the repetitive and conditional features of a design. This subset is SW design-specific, and is translated to the muxes, latches, registers, or sequences of HW objects (in case of the *for* loop) during synthesis.

(3) *Component subset* is a subset for expressing the concepts of component-based programming. It describes partitioning of a design into the different design units such as functions, procedures, blocks, entities, architectures, configurations, and packages.

(4) *Generic subset* is a subset for expressing the metaprogramming concepts, which are used to describe the families of similar designs. It includes *generics* for the declaration of the generic parameters, as well as the conditional (*if generate*) and repetitive (*for generate*) statements for the generation of the concurrent statements, which are performed during design elaboration prior to synthesis.

The subsets of VHDL are summarised in Table 1.

3. 3. C++ and SystemC

SystemC is a modeling language based on C++. It consists of a set of C++ class libraries and a simulation kernel that allow users to model the HW related concepts like concurrency, timing, etc. The language offers the following advantages:

(1) *Executable specification* – a model written in SystemC can be compiled and made executable.

(2) *Faster simulation* – the simulation speed is higher when compared to either VHDL or Verilog.

(3) *Higher abstraction levels* – when compared to the standard HDLs, SystemC can model the highly abstract concepts in an elegant and concise fashion.

(4) *Implementation independence* – a SystemC model does not specify a particular implementation. It can be implemented either in HW or in embedded SW, using a general-purpose processor or a DSP.

Table 1. Subsets of VHDL

Language Subset	Language Statements	Paradigm	Target Object
Behavioural	HW-specific data types, signal & variable declarations, assignments, logical expressions	Algorithmic	HW objects
Structural	processes, for loops, if statements, case statements	Structural	HW objects, collections of HW objects
Component	Blocks, functions, procedures, entities, architectures, configurations, packages	Component-based	Collections of HW objects
Generic	generics, if/for generate statements, generic port maps	Metaprogramming	HW design units

The important features of SystemC are briefly discussed below. The basic block of a SystemC program is a *module*. A module is similar to the entity in VHDL. It is an abstract representation of a functional unit, without specifying any implementation details. Each module has a set of *ports* through which it interacts with the outside world. Ports can be input, output, or input/output ones. Individual modules communicate with one another through *signals* that connect the ports of modules. The code that implements the algorithm of a module is encapsulated in one or more *processes*.

SystemC supports all C++ data types. For modeling HW, the additional data types are available. These include types for representing bits, bit vectors, 4-valued logic, variable precision integers, etc. In addition to these data types, the language also provides constructs that enable representing HW behaviour. There are *wait ()* statements that suspend execution, *write ()* and *read ()* functions to send and receive data from ports, and so on.

SystemC is an extension of C++, therefore, it has inherited all C++ features. The support of some of these features is, however, limited for synthesis. The non-synthesizable subset of C++/SystemC includes the local and nested class declarations, dynamic memory allocation, exception handling, function recursion, overloading, file I/O, inheritance, pointers,

floating-point data types and user-defined templates. More details of the language are discussed in [15, 30], and a complete list of features can be found in the SystemC user's guide [31]. After the analysis, we identify the subsets common to C++ and SystemC as follows:

(1) *Arithmetic subset* is a subset for describing SW algorithms. It includes natural data types (*int, long, float, double,* etc.), variables, arrays, operators, expressions, assignments, etc.

(2) *Structural subset* is a subset for expressing the structural programming concepts, such as *for/while* loops, *if/case* statements, which are used to describe the repetitive and conditional features of an algorithm.

(3) *Functional subset* is a subset for expressing the concepts of the component-based programming. A primary unit is a function that consists of an interface and body.

(4) *Object subset* is a subset for supporting the object-oriented programming. It includes classes, which encapsulate variables (states) and functions (methods).

(5) *Template subset* is a subset for expressing the meta-programming concepts. This subset describes the generic functions and classes. *Template* keyword is used to declare the generic parameters or types. Templates are evaluated during program compilation.

Additionally, SystemC has the HW design-specific subset as follows:

(6) *HW-specific subset* – a subset for expressing the HW design concepts. It includes modules, processes, channels, HW-specific data types (e.g., *sc_bit, sc_logic,* etc.), signals and operations with them.

The subsets of C++ and SystemC are summarised in Table 2.

Table 2. Subsets of C++ and SystemC

Language Subset	Language Statements	Paradigm	Target Object
Arithmetic	Natural data types, variables, arrays, expressions, assignments	Algorithmic	Machine code (assembler)
Structural	for/while loops, if/case statements	Structural	Statements
Function	Functions	Component-based	Collection of statements
Object	Classes	Object-oriented	Collection of variables and functions
Template	Templates	Meta-programming	Functions, classes
HW-specific[1]	*Signals, HW-specific data types, overloaded operations*	*Algorithmic*	*Collection of HW objects*

The relationship between the different subsets of the design languages is summarised in Figure 2.

[1] SystemC only

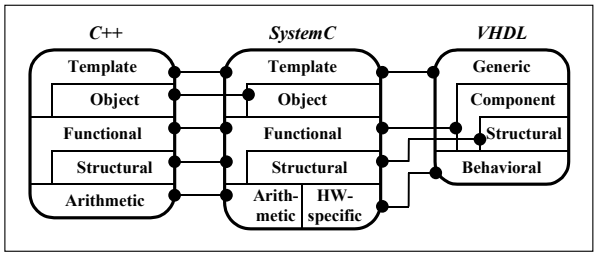

Figure 2. Relationship between subsets of the languages

4. Design of a generic calculator

In this case study, we demonstrate design a generic calculator model, which performs the simple arithmetic computations (addition and multiplication) with an arbitrary number of the expression members. We present the development process of the calculator model *subset by subset*, while at the same time comparing the capabilities of VHDL, C++ and SystemC. First, we implement the basic domain functionality (e.g., the 2-operand addition) as follows:

VHDL	C++	SystemC
signal x1, x2, y: bit_vector (15 downto 0); ... y <= x1 + x2;	int x1, x2, y; ... y = x1 + x2;	sc_signal<int> x1, x2, y; ... y = x1 + x2;

Note that operator "+" was overloaded in VHDL and SystemC for simplicity.

These three fragments, despite belonging to the different domains and languages, have many common features. First, variables (signals) are declared, with the explicit declaration of their type. Second, the required computations are described. Third, the value of the result is assigned. These fragments demonstrate the behavioural (arithmetic) subset. Note that VHDL and SystemC use the HW-specific data types, whereas C++ uses the SW-specific data types.

Now, we extend the basic functionality with the structural subset to implement the selection of the particular calculator operation as follows:

VHDL	C++/SystemC

constant ADD: integer:=1;
constant MULT: integer:=2;
signal op: integer;
...
if (op = ADD) then
  y <= x1 + x2;
end if;
if (op = MULT) then
  y <= x1 * x2;
end if;
	enum OPERATION
	{ADD, MULT};
	OPERATION op;
	...
	if (op == ADD)
	  y = x1 + x2;
	
	if (op == MULT)
	  y = x1 * x2;

The presented fragments demonstrate that the structural subset is very similar in all three languages.

Next, we demonstrate the implementation of the generic calculator model for an arbitrary number of the operands as follows (only addition case is shown). We have substituted a collection of variables with an

array (C++, SystemC) or a vector (VHDL). We used a vector type instead of an array in VHDL because of the synthesisability issues.

VHDL	C++/SystemC

variable temp: bit_vector (15 downto 0);
...
temp := x(15 downto 0);
for i in 2 to NUM loop
temp:=temp + x (i*16 downto (i-1)*16);
end loop;
y <= temp;
	const NUM = 3;
	...
	int temp = x[0];
	for(int i=1;i<NUM;i++)
	temp = temp+x[i];
	
	y = temp;

Next, we construct a design unit using the component subset of the languages. We develop a function in SystemC and C++, and an entity in VHDL as follows. SystemC function has no inputs and outputs specified, because it is not allowed for processes. VHDL port declaration considers the different lengths of the output signal depending on the operation specified.

VHDL	C++/SystemC

entity Calculator is
port (a: in bit_vector(NUM*16-1 downto 0);
c: out bit_vector
((OP-1)*(NUM-1)*16+15 downto 0));
end Calculator;
 	
architecture model of Calculator is
begin
-- calculator functionality
end model;
	int result (int num,
	int op, int *x) {
	// calculator
	// functionality
	}
	
	void Result() {
	// calculator
	// functionality
	}

The component subset allows separating component interface from its functionality and hiding the implementation details.

Next, we introduce the object subset to encapsulate variables and functions into a class. Since VHDL has no support for the object-oriented programming, we present a package, which can encapsulate type declarations, constants, procedures, and component declarations, instead. In C++/SystemC, the object subset is implemented using the class construct. SystemC class inherits a standard *sc_module* class, which implements the HW design concepts. Its constructor is defined using a library macro SC_CTOR. A process is registered using a library macro SC_METHOD.

Next, we introduce the meta-programming subset to describe the generic design units.

VHDL provides *generics* for writing the parameterised models. The generic component specification represents a set of possible component implementations or instantiations. A particular instance is generated when the instance-specific parameter values are passed into a generic component. The use of generics for parameterisation of structure and behaviour helps to create reusable design blocks.

VHDL provides the *generate* statement for generating the *repetitive structures*. These are the concurrent VHDL constructs that may contain further concurrent statements for replication, and help to effectively produce the iterative structures of a design. The *repetitive* form of generate statement specifies a discrete range of values, and for each of these it generates an instance of the statements in the body of the generate statement. The *conditional* form includes a Boolean expression, which governs whether the statements comprising the body of that *generate* statement are included. The generics provide a level of abstraction that allows isolating many implementation details when developing a HW system as well as reduce the risk of introducing errors when components are reused.

VHDL	SystemC

package calc_pack is
subtype op_type is
integer range 1 to 2;
constant ADD: op_type:=1;
constant MULT: op_type:=2;
 	
component Calculator
-- port declaration
end component;
end calc_pack;
	class Calculator:
	public sc_module {
	void Result();
	public:
	sc_in<T> x [NUM];
	sc_out<T> y;
	SC_CTOR(Calculator) {
	SC_METHOD(Result);
	}
	};

C++ templates provide the ability to write code generators and perform static computations. The template specifies only a generic skeleton for a class (or function) declaration. To complete the declaration, a programmer must supply a concrete value for each of the template's parameters. These parameters must be known at compile time. This causes the template to be *instantiated*: replacing all occurrences of the parameter with its value creates an instance of the template. C++ templates resemble a two-level language, where a *metalanguage* (i.e. templates) manipulates with the base code (classes and functions). Control structures (e.g., *if/else, for*) can be realised in templates using the *template recursion* technique [32] as a looping construct, and the class *template specialization* as a conditional construct.

The reasons behind the introduction of the generic parameters are as follows: (1) to express the generalisation of a model, when new generic parameters are introduced, and (2) to perform the customisation of a model, when a specific instance is derived depending on the values of the generic parameters. For example, a parameter OP (see below) generalises expressions regardless of the implementation, whether it is a variable or a generic parameter. However, in case of the generic implementation, only one specific expression is instantiated (adder or multiplier), whereas in a non-generic implementation both design units will be created. Therefore, genericity allows achieving larger reusability, as well as better performance.

Since the C++ metaprogramming subset has no repetitive and conditional generation constructs, we use template specialization for conditional generation and template recursion for repetitive generation of

code. The conditional generation is performed as follows:

VHDL	C++
g1: if OP = ADD generate	enum OPERATION {ADD, MULT}; template<OPERATION OP>
process (x) begin	class Calculator2x { }; class Calculator2x<ADD> {
-- adder -- functionality	public: static inline int Result (int a, int b) {
end process;	return a+b; }
end generate g1;	};

For VHDL and SystemC models, we introduced an additional generic parameter WIDTH, to specify data path width. For C++, we can not specify the size of the type directly, therefore, we introduced generic parameter T to specify a data type, thus indirectly specifying data size.

SystemC supports C++ templates without restrictions for modeling only, whereas the usage of templates for design of synthesizable HW models is not allowed. The implementations of the template class (C++/SystemC) and the generic entity (VHDL) are as follows:

VHDL	C++/SystemC

entity Calculator is
generic (WIDTH: integer;
NUM: integer;
OP: op_type);
-- port declaration
end Calculator;
	template <class T, int NUM,
	OPERATION op>
	class Calculator:
	public sc_module {
	/* variable (signal) and
	process (method) declarations */
	};

```
enum OPERATION {ADD, MULT};

template <class T, int NUM, OPERATION op>     // template
class Calculator: public sc_module  {         // object
    // Process
    void Result() {                           // functional
        T temp = x[0];                        // arithmetic
        if (op == ADD)                        // structural
            for (int i = 1; i < NUM; i++)     // structural
                temp = temp + x[i];           // arithmetic
        if (op == MULT)                       // structural
            for (int i = 1; i < NUM; i++)     // structural
                temp = temp * x[i];           // arithmetic
        y = temp;                             // arithmetic
    }                                         // functional
public:                                       // object
    // Signals
    sc_in<T> x[NUM];                          // HW-specific
    sc_out<T> y;                              // HW-specific
    sc_in<bool> clk;                          // HW-specific
    // Constructor
    SC_CTOR(Calculator) {                     // HW-specific
        SC_METHOD(Result);                    // HW-specific
        sensitive_pos << clk;                 // HW-specific
    }                                         // HW-specific
};                                            // object
```

Figure 3. Generic calculator implementation in SystemC with a decomposition into subsets

Finally, we present the results of our design efforts. The generic implementations of the calculator model are presented in Figures 3, 4, & 5, for SystemC, VHDL, and C++, respectively. Comments describe the subset to which the particular language statement belongs. Note that the *generic/template* subsets of the languages are shown in bold.

```
entity Calculator is                                          -- component
  generic (                                                   -- generic
        WIDTH : integer := 16;                                -- behavioral
        NUM : integer :=2;                                    -- behavioral
        OP : integer := ADD );                                -- behavioral
  port (                                                      -- component
        x: in std_logic_vector (NUM*WIDTH-1 downto 0);        -- behavioral
        y: out std_logic_vector                               -- behavioral
          ((OP-1)*(NUM-1)*WIDTH+WIDTH-1 downto 0) );          -- behavioral
end Calculator;                                               -- component

architecture model of Calculator is                           -- component
  begin                                                       -- component
  g1: if OP = ADD generate                                    -- generic
    process (x)                                               -- structural
      variable temp: std_logic_vector (WIDTH-1 downto 0);     -- behavioral
    begin                                                     -- structural
      temp := x(WIDTH-1 downto 0);                            -- behavioral
      for i in 2 to NUM loop                                  -- structural
        temp := temp + x(i*WIDTH-1 downto (i-1)*WIDTH);       -- behavioral
      end loop;                                               -- structural
      y <= temp;                                              -- behavioral
    end process;                                              -- structural
  end generate g1;                                            -- generic
  g2: if OP = MULT generate                                   -- generic
    process (x)                                               -- structural
    variable temp: std_logic_vector                           -- behavioral
        ((OP-1)*(NUM-1)*WIDTH+WIDTH-1 downto 0);              -- behavioral
    begin                                                     -- structural
      temp(WIDTH-1 downto 0) := x(WIDTH-1 downto 0);          -- behavioral
      for i in 2 to NUM loop                                  -- structural
        temp(i*WIDTH-1 downto 0) :=                           -- behavioral
              temp((i-1)*WIDTH-1 downto 0) *                  -- behavioral
              x(i*WIDTH-1 downto (i-1)*WIDTH);                -- behavioral
      end loop;                                               -- structural
      y <= temp;                                              -- structural
    end process;                                              -- structural
  end generate g2;                                            -- generic
end model;                                                    -- component
```

Figure 4. Generic calculator implementation in VHDL with a decomposition into subsets

5. Experimental results

A sample of the modeling results for the SystemC implementation of the generic calculator model is presented in Figure 6. The model is not synthesizable due to the usage of the template subset, therefore, no synthesis results are given.

Table 3. VHDL calculator synthesis results

Parameter values			Synthesis results		
Data width, bits	Op. type	No. of operands	Area, cells	Delay, ns	Estimated power usage
8	+	2	68	12.09	76.4425 uW
		3	136	14.56	169.2122 uW
		4	204	17.07	266.3093 uW
16	+	2	152	25.88	171.8903 uW
		3	304	28.38	387.6492 uW
		4	456	30.87	612.3716 uW
32	+	2	337	54.11	385.3689 uW
		3	666	56.98	857.5363 uW
		4	995	59.94	1.3466 mW
8	*	2	665	28.79	736.7024 uW
		3	2093	65.59	3.2794 mW
		4	4153	95.46	6.6230 mW
16	*	2	2805	68.34	4.3473 mW
		3	8419	130.09	14.5581 mW
		4	16888	185.35	29.9051 mW
32	*	2	11640	137.03	19.2108 mW
		3	35253	246.40	64.3500 mW
		4	16888	185.35	29.9051 mW

```
enum OPERATION {ADD, MULT};


// this class is never instantiated
template <OPERATION OP>
class Calculator2x { };

// template specialization for OP = ADD
class Calculator2x <ADD> {
public:
    template <class T>
    static inline T Result(T a, T b){
        return a+b;
    }
};

// template specialization for OP = MULT
class Calculator2x <MULT> {
public:
    template <class T>
    static inline T Result(T a, T b){
        return a*b;
    }
};
```

```
template <int NUM, OPERATION OP>
class Calculator {
public:
    template <class T>
    static inline T Result(T *a){ return Result(a,a+1); }
    template <class T>
    static inline T Result(T *a, T *b){
// template recursion to mimic the for loop
        return Calculator2x<OP>::Result(*a,
               Calculator<NUM-1,OP>::Result(b,b+1)); }
};
// recursion boundaries for NUM=2 and OP=ADD
class Calculator <2, ADD> {
public:
    template <class T>
    static inline T Result(T *a, T *b){
        return Calculator2x<ADD>::Result(*a, *b); }
};
// recursion boundaries for NUM=2 and OP=MULT
class Calculator <2, MULT> {
public:
    template <class T>
    static inline T Result(T *a, T *b){
        return Calculator2x<MULT>::Result(*a, *b); }
};
```

Figure 5. Generic calculator implementation with C++ templates: a 2-operand generic operation implementation (on the left), and generalised for an arbitrary number of operations (on the right)

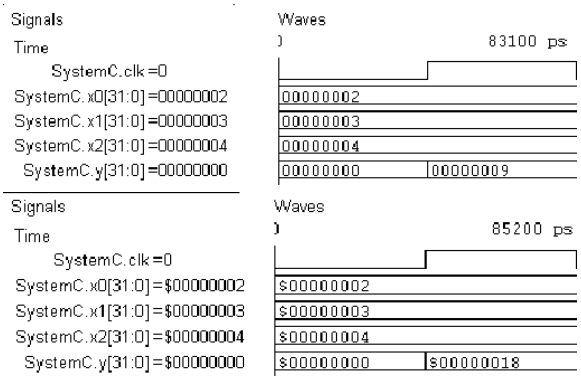

Figure 6. Modeling results of SystemC calculator: 3-operand adder (above), and 3-operand multiplier (below)

The synthesis results (we use Synopsys tools; 0.35um technology) for the VHDL implementation of the generic calculator model are presented in Table 3 (area optimisation was applied).

6. Discussion

SoC design raises new problems to the HW designers: the increased complexity, the interoperability between HW and embedded SW, and the requirements for the high-level system models and reusable design descriptions. In order to solve these and many other problems, the HW designers are trying to adopt the concepts and solutions from the SW domain such as object-oriented design, UML [33], design patterns, separation of concerns, metaprogramming, etc.

SystemC promises to bridge both SW and HW domains by providing (1) a modeling environment and simulation kernel for HW design, (2) the power and functionality of C++ for SW design, and (3) multiple abstraction levels for the system-level design. However, only some of the capabilities of SystemC are currently supported for synthesis to RTL implementation. Therefore, the research continues in adapting the existing synthesizable HDLs such as VHDL to the increased requirements for SoC design.

Currently, design of a large-scale system is usually performed as follows. (1) High level specification of a system using UML diagrams and a set of the pre-defined design patterns. (2) Development of a system model using C++/SystemC. (3) Verification of a system model using SystemC simulation environment. (4) Automatic translation (if possible) or manual conversion of a system model to the synthesizable HDL (VHDL, Verilog) specification for further synthesis.

The problem with this approach is that conversion of UML diagrams to SystemC code and further to the standard HDL specification is not straightforward and is a time-consuming and error-prone task. Automatic solutions are not always possible due to the complexity of the domain and a lack of the necessary abstractions (e.g., UML lacks timing concept). The available tools usually support only the translation of a limited subset of the high-level language. For example, often designers have to manually rewrite SystemC models to make them convertible to VHDL and synthesizable. In such cases, domain analysis and thorough knowledge of the design languages is essential. The existence of similar abstractions or subsets in the languages helps to implement such a translation more easily. However, more research is needed in mapping the advanced SW design concepts such as design patterns to the synthesizable HDL abstractions.

7. Conclusions and future work

VHDL, C++, and SystemC languages show a great deal of similarity, despite different domains of their application. Language subsets and their roles are comparable. The usage of the metaprogramming-oriented language subset allows achieving greater reusability and better performance in the domain.

However, the application of templates for developing generic SystemC models is still hindered by the limitations of the synthesis tools.

Future work will focus on further research on the convergence of SW and HW design methodologies, including the application of the object-oriented design techniques, UML class diagrams and design patterns for HW design.

References

[1] **A. A Jerraya, M. Romdhani, P. 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.

[2] **R. Damaševičius, V. Štuikys.** Separation of Concerns in Multi-language Specifications. *INFORMATICA, Vol.* 13, *No.* 3, *Institute of Mathematics and Informatics, Lithuanian Academy of Sciences, Vilnius,* 2002, 255-274.

[3] **IEEE.** *VHDL Interactive Tutorial: A Learning Tool for IEEE Std.* 1076 *VHDL. IEEE, CD-ROM*, 1996.

[4] **T. Kropf.** The Verilog Hardware Description Language. *Kluwer*, 1996.

[5] **S. M. Loo, B. Earl Wells, N. Freije, J. Kulick.** Handel-C for Rapid Prototyping of VLSI Coprocessors for Real Time Systems. *Proc. of the Southeastern Symposium on System Theory SSST*-2002, *March* 18-19, *Huntsville, AL*, 6-10.

[6] **Y. Li, M. Leeser.** HML: An Innovative Hardware Description Language and Its Translation to VHDL. *Proc. of the IFIP Int. Conference on Computer Hardware Description Languages and Their Applications (CHDL'*1995*), August* 1995, *Chiba, Japan*, 691-696.

[7] **P. Bellows, B. Hutchings.** JHDL – an HDL for reconfigurable systems. *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'*98*), April* 1998*, Napa, CA, USA*, 175-185.

[8] **M. Radetzki, W. Putzke-Röming, W. Nebel.** Objective VHDL: Tools and Applications. *Proc. of Forum on Design Languages (FDL)*, 1998.

[9] **J. Zhu, D. D. Gajski.** OpenJ: An Extensible System Level Design Language. *Proc. of Design Automation and Test Conference in Europe (DATE 1999)*, 9-12 *March* 1999, *Munich, Germany*, 480-484.

[10] **P. Alexander, D. Barton.** An Introduction to Rosetta. *Invited tutorial at HDLCon00, March* 2000*, San Jose, CA, USA.*

[11] **W. Glunz, T. Kruse, T. Rössel.** System-Level Hardware Design with SDL. *Proc. of the 6th SDL Forum SDL '*93, *October* 1993, *Darmstadt*, 17 – 28.

[12] **D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao.** SpecC: Specification Language and Methodology. *Kluwer Academic Publishers, Dordrecht*, 2000.

[13] **P. J. Ashenden, P. A. Wilsey, D. E. Martin.** SUAVE: Extending VHDL to Improve Modeling Support. *IEEE Design and Test of Computers, Vol*. 15*, No.* 2 (*April-June* 1998), 34-44.

[14] **P. Flake, S. Davidmann.** Superlog, a Unified Design Language for System-on-Chip. *Proc. of the Asia and South Pacific Design Automation Conference (ASP-DAC'*2000*), January* 25-28, 2000, Yokohama, Japan.

[15] **S. Swan.** An Introduction to System Level Modeling in SystemC 2.0. *White paper, OSCI, May* 2001.

[16] **R. Siegmund, D. Müller.** SystemC^{SV} - An Extension of SystemC for Mixed Multi-Level Communication Modeling and Interface-Based System Design. *Proc. of Design Automation and Test Conference in Europe DATE* 2001, *March* 13-16, *Munich, Germany*, 26-30.

[17] **J. O. Coplien.** Multi-Paradigm Design. *Ph.D. Thesis. Vrije Universiteit, Brussels, Belgium*, 2000.

[18] **A. Gabrielli, E. Gandolfi, M. Masetti.** Design of a Very High Speed Fuzzy Processor by VHDL Language. *European Design & Test Conference, User Forum*, 11-14 *March* 1996, *Paris, France*, 121-125.

[19] **R. Allen, D. Gajski.** The case for C/C++ design. *EEDesign, June* 9, 2000.

[20] **D. J. Smith.** VHDL & Verilog Compared & Contrasted – Plus Modeled Example Written in VHDL, Verilog and C. *Proc. of the 33rd Design Automation Conference (DAC'1996), June* 3-7, *Las Vegas, Nevada, USA*, 1996, 771-776.

[21] **A. Gerstlauer, S. Zhao, D.D. Gajski.** VHDL+/SpecC Comparisons – A Case Study. *Technical Report ICS*-98-23, *Department of Information and Computer Science, University of California*, 1998.

[22] **A. Bunker, S. A. McKee, G. Gopalakrishnan.** An Overview of Formal Hardware Specification Languages. *Proc. Grace Hopper Celebration of Women in Computing (GHC*2002*), October* 9-12 2002, *British Columbia, Canada*.

[23] **E.A. Bezerra.** Selecting a Hardware Description Language for the Design of an On-board Scientific Instrument Processing Module. 2*nd U.K. ACM SIGDA Workshop on Electronic Design Automation*, 16-17 *September* 2002, *Bournemouth University, UK*.

[24] **G. Bonanome.** Hardware Description Languages Compared: Verilog and SystemC. *Technical Report, Department of Computer Science, Columbia University, New York*, USA, 2001.

[25] **N. Agliada, A. Fin, F. Fummi, M. Martignano, G. Pravdelli.** On the Reuse of VHDL Modules into SystemC Designs. *Forum on Design Languages (FDL* 2001*),* 3-7 *September* 2001, *Lyon, France*.

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

[27] **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. Kluwer Academic Publishers*, 2000.

[28] **P.J. Ashenden.** The Designer's Guide to VHDL. *Morgan Kaufmann Publishers, San Francisco*, 1996.

[29] **K.C. Chang.** Digital Design and Modeling with VHDL and Synthesis. *IEEE Computer Society Pr*ess, 1997.

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

[31] Synopsys Inc., CoWare Inc., Frontier Design, Inc. *SystemC User's Guide, 2000. Available at: http://www.systemc.org*

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

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