Techniques to Evolve a C++ Based System Design Language

Robert Paško, and Serge Vernalde
Inter-University Microelectronics Center
Kapeldreef 57, B-3001 Leuven, Belgium
{pasko,vernalde}@imec.be

Patrick Schaumont
UCLA, Electrical Engineering Department
Los Angeles, CA 90095-1594
schaum@ee.ucla.edu

Abstract

Complex systems-on-chip present one of the most challenging design problems of today. To meet this challenge, new design languages capable to model such heterogeneous, dynamic systems are needed. For implementation of such a language, the use of an object oriented C++ class library has proven to be a promising approach, since new classes dealing with design- and platform-specific problems can be added in a conceptual and seamlessly reusable way.

This paper shows the development of such an extension aimed to provide a platform-independent high-level structured storage object through hiding of the low-level implementation details. It results in a completely virtualised, user-extendible component, suitable for use in heterogeneous systems.

1. Introduction

The design of a System on Chip (SoC) is one of the major conceptual challenges of today. It typically requires the modelling and integration of various SoftWare (SW), HardWare (HW) and Intellectual Property (IP) elements, so any SoC aspiring language or methodology must provide means to deal with these issues, preferably in a uniform SW-like way. This need has stimulated a significant amount of research aimed at the exploration of new languages/design methodologies to meet these goals, i.e. to be able to model SW and HW parts, handle the interfaces between them and provide for a seamless use of IP cores.

C++ based methodologies (SystemC [1, 2], Forte (prev. CynApps) [3], Ocapi [4, 5, 6]) tackle the complexity and diversity of the SoC designs by introducing the Object Oriented (OO) programming principles into the design process. These principles allow to separate interface from implementations, and provide for reuse at a higher level of abstraction using the OO concept of classes. Furthermore, the core set of classes implementing the basic semantic primitives can be easily extended according to the designer's needs. This *extendibility*, typical for OO languages, is one of the most interesting ideas behind the C++ based design. It allows to refine the methodology for a given task and platform, as well as to generalise and reuse the obtained results. Unfortunately, it implies that knowledge of C++ is necessary to certain extend. However, even without the bells and whistles of the advanced OO techniques, understanding of the basic OO principles is often sufficient for normal users to take advantage of these techniques.

In this paper, we present an extension of the C++ based design methodology *Ocapi-XL* with a set of classes providing the support for array-like data structures. We begin with several ad-hoc classes implementing a storage element on different platforms, and gradually show the conceptual refinement of these into a full fledged virtual array component. This component completely hides the low-level implementation details behind a clean high-level interface. We put a lot of emphasis onto a detailed explanation of the used programming techniques, so that the reader can follow and understand the role of the underlying C++ concepts in the design process.

The rest of the paper is structured as follows. In the next section, the essential ideas of C++ based design are discussed. In Section 3, our methodology *Ocapi-XL* is compared to the mainstream SystemC and short overview of *Ocapi-XL* is given. The initial ad-hoc array implementations are shown in Section 4. In Section 5, the refinement of the concepts using OO principles is discussed, and the resulting set of classes providing the generic array functionality is presented. Finally, the conclusions are drawn in Section 6.

2. C++ Based Design and Related Work

The essential idea of all C++ based design methodologies is to support the design of HW, or even SW via a library of the necessary semantic primitives. It is realised through C++ classes, and complemented with a suitable simulation engine. This makes the idea of C++ based methodologies quite similar to the concept of meta-languages, i.e. lan-

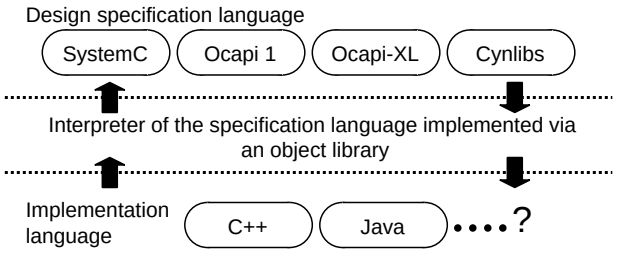


Figure 1. Meta-language concept in the C++ based design methodologies.

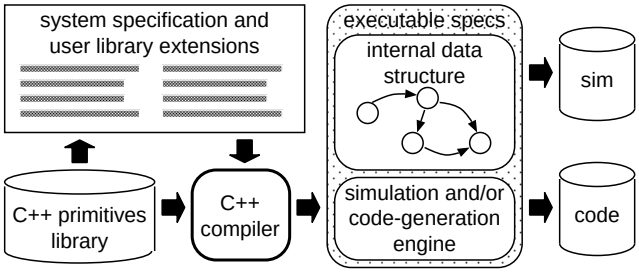


Figure 2. C++ based methodology design flow

guages not self-contained, but rather interpreted in terms of another programming language, as illustrated in Fig. 1. This strategy has some interesting advantages, like implementation simplicity and seamless extendibility. The extendibility is even more important in the SoC context, because the necessary semantic elements, as well as computational models, can vary from one application to another dramatically. However, when designing such an application in a C++ based methodology, the designer can devise his/her own extensions to deal with the problems at hand. This approach is well supported in the OO programming paradigm, since it allows to define and use the new primitives in the same way as the built-in data types and functions. Finally, the description is compiled using a standard C++ compiler, resulting in an executable specification, as shown in Fig. 2.

The advantages of the C++ based design languages with respect to the extendibility were considered recently by several authors. The methodology of choice in these works was SystemC[1], which appears to be more and more accepted by the design community. In [7] and [8], the support for high-level communications was added, in the first case to allow the clock-true simulation of a multi-processor platform, and in the second to facilitate *interface-based design* paradigm. In [9], a methodology for smooth interfacing of a third party SW was discussed. All these papers concentrated almost exclusively on the ensuing application of the library extensions. This paper targets the C++ issues related to the implementation of the extension itself, since in our opinion, this sort of information was generally missing.

In several recent designs made with *Ocapi-XL* [5], we have repeatedly faced problems which were solvable through an extension of the core library. Among the most compelling ones was the support of an indexed storage element, not present in the base library. However, the introduction of such a simple high-level concept was complicated by the variability of the target platforms, each one requiring a different implementation. For example, a register bank was needed in a design of a small RISC core for a Field Programmable Gate Array (FPGA), C-like arrays were required in the design of an embedded SW medium access (MAC) layer for a wireless network [10], or finally, on- and off-chip SRAMs were used for storage in an FPGA based embedded camera design [11]. Such diversity and need for reuse in different contexts is typical for SoC design, and the presented approach can be can be considered rather characteristic for C++ based methodologies.

[1]The relation between *Ocapi-XL* and SystemC will be briefly outlined in the next section.

3. Ocapi-XL Overview

Ocapi-XL is a good example of a C++ based design methodology. It was specifically designed for modelling of heterogeneous HW/SW systems and it uses a unified approach to HW and SW code [5] to achieve that objective. More specifically, it provides parallelism at a process level, uses high-level communication primitives like messages or semaphores, and offers several options for incorporation of C/C++ legacy code for simulation purposes. This features put it approximately on the same level of abstraction as SystemC v2.0, which embodies similar concepts. Thus *Ocapi-XL* can be used for executable specifications of a system in a similar way as it is done in SystemC (e.g. in [12]).

The crucial difference between *Ocapi-XL* and SystemC is that *Ocapi-XL* unifies the simulation and code generation under the same set of objects. Thus, a single executable *Ocapi-XL* program can perform simulation as well as HDL or SW code generation. This allows a gradual refinement of the original high-level C/C++ code inside of the same environment. In order to make the following chapters understandable, the supported modelling/code-generation primitives must be introduced more in detail.

The basic quantum of computation is an instruction. In HW, multiple instructions can be executed in a single clock-cycle. The instructions can be divided into data and control ones. The data instructions provide the necessary collection of arithmetic, logic and assignment operations for the *Ocapi-XL* data type `Int`, which stands for a simple integer. For control, the conditional execution statements, as well as

```
/*--- variables definitions ---*/
 Int a,b,c;
/*--- starting a new process named P ---*/
procHLHW P("P");
 a = cc(0);   //assigning constant 0 to a
 //cc(x) is a macro converting integer to Int
label L;      //label definition for branching
 c = a+cc(1); //arithmetic operation
 //control statement equivalent
 //to a C statement => b = (c == 1) ? a : b
 b = (c == cc(1)).sel(a,b);
 sync_();     //control returns to the scheduler
 jumpif(L,a); //conditional branch to label L
 a = a & b;   //logic operation
 jump(L);     //unconditional branch
```

Figure 3. *Ocapi-XL* **code example.**

looping and branching instructions are provided.

To support parallel execution, *Ocapi-XL* provides the notion of a process as the basic level of parallelism and hierarchy. Two essential types of processes are distinguished: software (`procHLSW`), and hardware (`procHLHW`). The switching between running and waiting processes is done on a blocking operation or via the special instruction `sync_()`, which represents a clock-edge in HW and a process switch in SW. Some examples of the data, control and process primitives are shown in Fig. 3.

Communication between processes is implemented via three basic communication primitives: *messages* implementing blocking *read* and non-blocking *write*; *shared* variables with non-blocking access; and binary *semaphores* with blocking *grab* and non-blocking *release* operations. The blocking operations can be used for inter-process synchronisation as well.

To increase the flexibility, a direct interface to C++ is implemented via a so-called Foreign Language Interface (FLI). It allows to run any snippet of C++ code during an *Ocapi-XL* simulation by overloading the originally empty `run` method of the FLI class with the intended code (Fig. 4). Another way to incorporate a C/C++ code into simulation is to use a special type of the process, which runs a C thread inside. However, this technique is not used in the rest of the paper, so it will not be discussed more in detail.

Finally, the *Ocapi-XL* description compiles into executable code. In addition, it supports code-generation to other languages: VHDL/Verilog for hardware and C for software. The FLI's are appearing in the generated code as function signatures/calls in the C code and black-box entities with appropriate ports in VHDL/Verilog.

```
/*--- FLI supplying the values of sin(x) ---*/
class sin_fli : public fli {
 public:
  //virtual run method is overloaded with
  //the required behaviour
  void run() { int x = var[1]; var[2] = sin(x); }
  //var[..] are provided FLI variables used
  //to import/export data to/from an FLI object
};
/*--- usage of the above defined fli ---*/
sin_fli SIN;  //defining the fli object
Int a,b;
//fliIn and fliOut define the port direction
call(SIN,fliIn(a),fliOut(b)); //<==> b = sin(a)
```

Figure 4. *Ocapi-XL* **FLI syntax and semantics.**

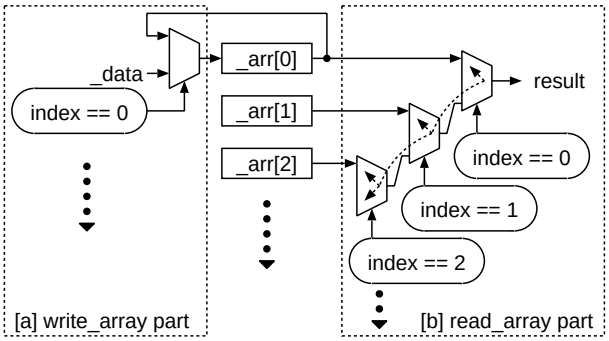


Figure 5. Hardware `RegArr` **structure.**

4. Initial Ad-hoc Array Designs

First, we present various ad-hoc schemes implementing the indexed storage element in different contexts.

4.1. Register Array Implementation

The first type of array structure supports modeling of the register bank of a RISC core. We developed a class `RegArr` that provides such array access and automatically refines to registers and multiplexers after the code generation, as shown in Fig. 5.

In order to get the proper code for synthesis, the *read* and *write* operations must be implemented themselves using *Ocapi-XL* instructions. The key is the recursive method `match_index`, which internally unfolds into the multiplexer tree, as shown in Fig. 5[b]. Similarly, the `for` loop in the `write_array` method will generate the structure in Fig. 5[a]. The corresponding class definition is shown in Fig. 6 [2].

[2]For readability, references and consts will be avoided, if not strictly required. e.g. the purists' `read_array` definition is: `const Int& RegArr::read_array(const Int& index) const;`

```
/*--- Definition of the initial class ---*/
class RegArr {
  vector<Int> _arr;
  Int match_index(Int _index, int _i);
 public:
  RegArr(int _N) : _arr(N) {}
  /* the read and write methods */
  Int read_array(Int _index);
  void write_array(Int _index, Int data);
};
/*--- generating the reading multiplexers ---*/
Int RegArr::match_index(Int _index, int _i) {
  //if _arr[i] is the last element, finish
  //the recursion and directly return it
  if (_i == _arr.size()-1) { return _arr[i]; }
  //otherwise check for the index match
  //and recurse further down the array
  return (_index == cc(_i)).sel(
            _arr[i],
            match_index(_index,i+1));
}
/*--- reads array by calling match method ---*/
Int RegArr::read_array(Int _index) {
  return match_index(_index,0);
}
/*--- write_array uses similar mechanism ----*/
void RegArr::write_array(Int _index, Int _data) {
 for(int i=0; i<_arr.size(); i++) {
  _arr[i] = (_index != cc(i)).sel(
              _arr[i],
              _data);
} }
```

Figure 6. Initial `RegArr` implementation.

4.2. C-like Array Implementation

The second type of array structure was used in a HIPERLAN II MAC layer [10], where a SW implementation was the primary target. We based our implementation here on the FLI extension mechanism, since we wanted to use the C compilers' native implementation of an array. We still want however seamless integration of this C array with the *Ocapi-XL* computation model. This leads to a new implementation of a generic array class `GenArr` based on the FLI mechanism, shown in Fig. 7.

The functionality in Fig. 7 is achieved through two FLI calls for read and write, which are sharing the same C++ vector object. This implementation results in fast simulation, as well as proper code-generation. Since the FLI's are generated as C-function calls, it is sufficient to provide two externally defined functions, or macros, implementing the C-style array access.

4.3. RAM Implementation

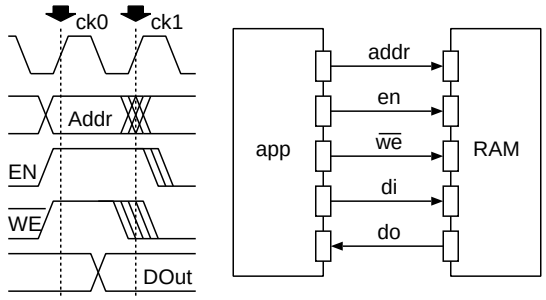
A third refinement of an array was introduced to facilitate the use of SRAM modules connected to an FPGA based

```
/*--- class providing reading through FLI ---*/
class fliREAD : public fli {
  //reference to an outside array, since it must
  //be shared for reading and writing
  vector<int>& arr;
 public:
  fliREAD(vector<int>& _arr) : arr(_arr) {}
  //run reads the array at the supplied address
  void run {
    int addr = var[0];
    var[1] = arr[addr];
  }
};
/*--- class providing writing through FLI ---*/
class fliWRITE i: public fli {
  vector<int>& arr;
 public:
  fliWRITE(vector<int>& _arr) : arr(_arr) {}
  //writes the data at the supplied address
  void run {
    int addr = var[0];
    int data = var[1];
    arr[addr] = data;
  }
};
/*--- class GenArr implementation ---*/
class GenArr {
  vector<int> arr;
  fliREAD     RD;
  fliWRITE    WR;
  Int         result;
 public:
  GenArr(int _N) : arr(_N), rd(arr), wr(arr) {}
  //read and write just call the FLI's
  Int read_array(Int _index) {
    call(RD, fliIn(_index), fliOut(result));
    return result;
  }
  void write_array(Int _index, Int _data) {
    call(WR, fliIn(_index), fliIn(_data));
} };
```

Figure 7. `GenArr` implementation using FLIs.

networked camera [11]. An external RAM Array can be considered as a mixture of the two previous approaches. The memory itself is an external element, thus FLI is the proper modelling technique. However, the reading and writing operations have to comply to a certain protocol, which must be implemented in *Ocapi-XL* code. Let us assume the protocol for reading, as shown in Fig. 8[a], i.e. the address and control signals must be asserted before the first clock cycle, while the data can be read in with the second. In addition to this behaviour, an interface as in Fig. 8[b] must be generated.

A possible implementation of `RAMArr` class is shown in Fig. 9. The functionality is provided inside the `fliRAM` class. It is not necessary to separate the *read* and *write* operations into two FLI's, since $\overline{WE}$ can be used to choose the proper one. The `read_array` method describes the access



[a] RAM read access timing diagram

[b] generated interface

Figure 8. RAM access protocol and interface.

protocol, i.e. two successive calls of the FLI, first with the control signals asserted, and second to read the data. The write operation was implemented in a similar way. This code, with minor modifications, was used to simulate wide range of RAMs, starting from external RAM modules to the internal Xilinx-Virtex SelectRAM and BlockRAM structures.

5. Virtualising the Concept

All the above defined classes represent the low-level implementations of a high-level concept: *array*. They are, unfortunately, too application specific to provide a good library component. In order to provide a re-usable, virtual *array* component, several requirements have to be taken into account. First, a uniform high-level user-friendly interface should be devised. Second, the structure must provide efficient low-level implementations. Finally, it has to be seamlessly extendible. The first goal can be achieved by a separation of the interface and implementation.

5.1. Separation of Interface and Implementation

The separation of interface and implementation can be realised in C++ through a combination of the *inheritance* and *polymorphism* mechanisms. The first one provides the means for indicating commonalities between objects by defining a parent-child relationship, while the second ensures that the proper code will be executed when parent or child class is used. In our problem, the inheritance hierarchy, as shown in Fig. 10, can be used for the separation of interface and implementation. The parent `AbsArr` class's sole purpose is to define the appropriate high-level interface for later implementations. The methods `read_array` and `write_array` are the obvious candidates. Every one of the descendent classes provides the specific implementation of the abstract array object: `RegArr`

```
/*--- fliRAM interface ---*/
class fliRAM : public fli {
  vector<int> arr; //C++ array to hold the data
  //to achieve the pipeling effect at the output
  int         lastdata;
 public:
  fliRAM(int _N) : arr(_N), lastdata(0) {}
  virtual void run() {
    int addr = var[0]; //address
    int en   = var[1]; //enable signal
    int we   = var[2]; //write enable signal
    int di   = var[3]; //data in
    if (en) {
      if (we == 1) { // read operation
        lastdata = arr[addr];
      } else {       // write operation
        arr[addr] = di;
        lastdata = di; // write through
    } }
    var[4] = lastdata;
} };
/*--- class RAMArr implementation ---*/
class RAMArr : public AbsArr {
  fliRAM  RAM;
  // the intermediate signals
  Int addr, en, we, di, do;
 public:
  RAMArr(int _N) : RAM(_N);
  Int read_array(Int _index);
  void write_array(Int _index, Int _data);
};
/*--- RAMArr reading procedure ---*/
Int RAMArr::read_array(Int _index) {
  addr = _index;   // _index is the address
  en = we = cc(1); // enable and read set to 1
  di = cc(0);      // input data is 0 for read
  // calling the fli
  call(fliIn(addr), fliIn(en), fliIn(we),
       fliIn(di), fliOut(do));
  sync_(); //---- the clock cycle boundary ----
  // second call provides the data
  en = cc(1);     // en is 1 in the 2nd cycle
  we = cc(0);     // we is 0 in the 2nd cycle
  call(fliIn(addr), fliIn(en), fliIn(we),
       fliIn(di), fliOut(do));
  return do;
}
```

Figure 9. `RAMArr` **implementation using FLI.**

is the previously described register array, `RAMArr` provides the RAM interface and `GenArr` is the FLI based array model. Polymorphism ensures that proper implementation of `read_array` or `write_array` methods will be invoked whenever the `AbsArr` will be substituted by one of its descendent classes.

The C++ declaration defining the inheritance tree from Fig. 10 is shown in Fig. 11. The `AbsArr` parent class contains just two *pure virtual*, i.e. empty, method declarations. The descendant classes can be implemented as in the previous section. The inheritance relationship to `AbsArr`

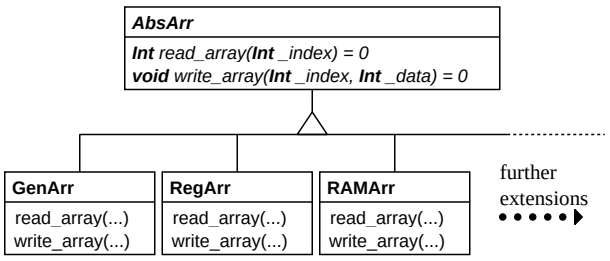


Figure 10. The inheritance hierarchy of array classes.

```
/*--- Abstract parent class ---*/
class AbsArr {
 public:
  //pure virtual methods to be redefined later
  Int read_array(Int _index) = 0;
  void write_array(Int _index, Int _data) = 0;
};
/*--- Register Array class ---*/
class RegArr : public AbsArr {
 //same code as in the previous section
};
/*--- generic array class ---*/
class GenArr : public AbsArr {
 //same code as in the previous section
};
/*--- RAM based array class ---*/
class RAMArrr : public AbsArr {
 //same code as in the previous section
};
```

Figure 11. Hierarchy definition in C++

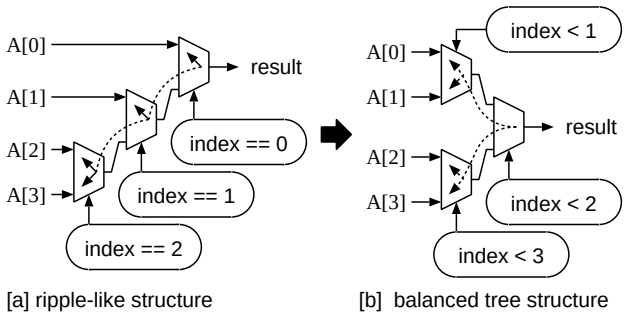


Figure 12. `RegArr` **reading implemented via a balanced tree.**

```
// searches for an element between l and h
Int RegArr::bsearch(Int _index, int l, int h) {
  //the base level is reached
  if (l == h) { return _arr[li]; }
  else {
    //split the array into two halves
    //and continute the search further down
    int half = floor((l + h)/2) + 1);
    return (_index < cc(half)).sel(
      bsearch(_index,l,half-1),
      bsearch(_index,half,h));
} }
/*--- read_array searches the whole array ---*/
Int RegArr::read_array(Int _index) {
  return bsearch(_index,0,_arr.size()-1);
}
```

Figure 13. Binary search implementation.

must be indicated only in the header definition, as shown in Fig. 11.

This set of classes provides the user an array interface complemented with an easily extendible set of implementations. Since the component is intended for frequent reuse, every detail has to be handled with care. This includes for example the issues of efficiency and user-friendly syntax.

5.2. Refinements of the Scheme

Since the designed classes can be reused in various scenarios and with different back-end tools, the structure should be as efficient as possible. For example, the multiplexer chain providing the reading functionality is generated in a ripple-like configuration, making the overall delay function of the array length n. A preferable structure in this scenario is a balanced tree, making the total delay a function of $\log(n)$, as shown in Fig. 12.

A simple binary search strategy can achieve this our goal. The basic idea for implementation of the binary search is to start from the right side of the multiplexer tree and instead of comparing the specific index, just decide in which half of the array is the indexed element placed (see Fig. 12[b]). With this information, the whole procedure can be recursively applied on both halves, till the base level is reached, as shown in the C++ implementation depicted in Fig. 13. Thus the only necessary modification inside of the `RegArr` class is the replacement of the original `match_index` method with the new `bsearch`, as shown in Fig. 13.

5.3. Adding Syntactic Consistency

Finally, there are two eventual objections concerning the usage of the presented set of classes. First, in order to be able to switch between various implementations, it is necessary to work with pointers or references to the `AbsArr` array, which is not very comfortable. Second, the array access methods `read_array` and `write_array` are lacking the simplicity and elegance of the C++ bracket operator `[]` used normally for an array access.

The remedy for the first problem is relatively easy and straightforward. One additional array class, the `IntArr`,

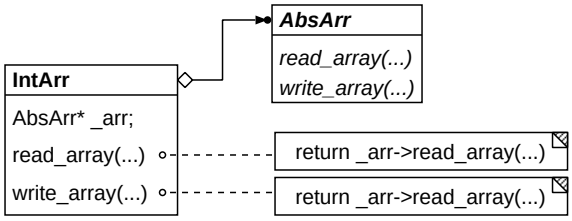


Figure 14. `IntArr` **wrapper class.**

```
/*--- overloading operator[] ---*/
class foo {
  int* arr;
  //operator[] just returns a reference to
  //an appropriate memory location
  int& operator[](int i) { return arr[i]; }
};
//...
foo  A(10);
A[5] = 5;          //writing operation
int i = 2 + A[7]; //reading operation
/*--- same approach for IntArr ??? ---*/
IntArr A(10); Int x;
A[5] = x;  //lhs -> write_array(..) to be called
x = A[4];  //rhs -> read_array(..) to be called
```

Figure 15. Overloading of the `operator[]`**.**

is needed to provide a wrapper around the `AbsArr` and the descendant classes. The appropriate array type to create can be specified as a parameter to the `IntArr` class constructor, and the methods `read_array` and `write_array` will simply call the polymorphic methods of the `AbsArr`, as shown in Fig. 14.

The second problem, however, requires some imaginative C++ programming techniques to employ. Since C++ allows overloading of the `operator[]`, the solution seems to be similar to an example given in almost every C++ text, shown in Fig. 15. This scheme of Fig. 15 works if the object returned by the overloaded operator can be used in reading as well as writing context, as in the case of integers. In our array implementation however, two fundamentally different operations are to be performed, i.e. it is necessary to distinguish, whether the `operator[]` is called for reading or writing.

This distinction between reading and writing can not be done at the time when the `operator[]` is called. Thus, the evaluation of the expression must somehow be delayed. This can be achieved by an intermediate *proxy* object, which does not perform the array access immediately, but rather stores the necessary information to do it later. Thorough discussion of the implementation of the *lazy evaluation scheme* in C++ is given in [13].

The decision, in which context the array access was made, can be based on the reasoning indicated in Fig. 16.

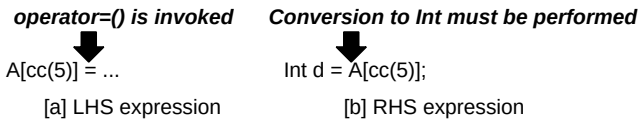


Figure 16. Distinction between read and write.

```
/*--- the Proxy class definition ---*/
class ProxyInt {
  // the necessary info to store, so array
  // access can be performed later,
  AbsArr* a;
  Int&    i;
 public:
  ProxyInt(AbsArr* _a, Int& _i) : a(_a), i(_i) {}
  //operator= used => left hand-side expression
  ProxyInt& operator=(Int& _d) {
    a->write_array(i,_d);
    return *this;
  }
  //type conversion used => right hand-side
  operator Int() {
    return a->read_array(i);
  } };
/*--- final and complete IntArr class ---*/
class IntArr {
  AbsArr* arr;
 public:
  enum IntType {GEN, REG, RAM}; // and others ..
  IntArr(int N, IntType=Gen) {
    arr = 0;
    if (IntType == GEN) arr = new GenArr(_N);
    if (IntType == REG) arr = new RegArr(_N);
    if (IntType == RAM) arr = new RAMArr(_N);
  }
  ProxyInt operator[](Int& _i) {
    return ProxyInt(arr,_i);
  } };
```

Figure 17. Final version of the `IntArr` **class.**

If it was used at the left hand-side of an expression, assignment operator will be eventually invoked. So the code for array write can be put inside of a method overloading that operator of the *proxy* object. Similarly, an automatic type conversion will be invoked in the right hand-side scenario, so that is the place for the array read code. The C++ implementation of the `IntArr` wrapper class providing access via the `operator[]` is given in Fig. 17 [3].

The last example of the proxy classes requires considerable knowledge of the C++ language. However, the proxy class is completely hidden from the designer. He/she is confronted only with the clean and simple interface of the `IntArr` class, while the underlying code can be pro-

[3]The definition of the `ProxyInt` is in reality a bit more complex (the interested reader should look at [13] and the proxy pattern in [14]), but the presented code shows the essential idea.

```
//10 element register array
IntArr  A(10,IntArr::REG);
//interface to a 1000 element RAM
IntArr  B(1000,IntArr::RAM);
Int     adr, dt, dt1;
//...
adr = cc(5);
dt = A[adr]; // reads dt from a register array
// writes to a memory addressed by dt with
B[dt] = dt1; // the proper access protocol
```

Figure 18. Example of `IntArr` definition and use.

grammed by a skilled C++ expert designing the array module. An example of code using the presented extensions is shown in Fig. 18. It demonstrates the high-level simplicity of the concept.

6. Conclusions

C++ based design methodologies are among the most appealing candidates for the design of complex SoC applications. One of the most important properties is the seamless extendibility, which is especially useful in the highly heterogeneous SoC environments.

We have demonstrated, how virtualisation of an extension component can be realised in C++ based methodologies, thanks to the OOP features. The resulting set of classes provides the user with an extendible *array* type of container featuring an easy-to-use high-level interface hiding many complex low-level implementation details. The use of such component results in smaller, more intelligible code, with the possibility to extend it whenever necessary.

References

[1] Guido Arnout, "SystemC Standard", *'Proc. of ASP-DAC 2000*, Yokohama, Japan, pp.573-577, January 2000.

[2] SystemC [online]. http://www.systemc.org

[3] Forte (prev. CynApps) [online]. http://www.ForteDS.com/

[4] P. Schaumont, et al., "A Programming Environment for the Design of Complex High-Speed ASIC's", *Proc. DAC 1998*, San Francisco, CA, June 1998.

[5] G.Vanmeerbeeck, et al., "Hardware/Software Partitioning fo Embedded Systems in OCAPI-XL", *Proc. of CODES 2001*, Copenhagen, Denmark, April 2001.

[6] Ocapi [online]. http://www.imec.be/ocapi

[7] Patrice Gerin, et al., "Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures", *Proc. of ASP-DAC 2001*, Yokohama, Japan, February 2001.

[8] Robert Siegmund, and Dietmar Müller, "SystemC ^{SV}: An Extension of SystemC for Mixed Multi-Level Communication Modelling and Interface-Based System Design", *Proc. of DATE 2001*, Munich Germany, March 2001.

[9] Luc Charest, et al., "A Methodology for Interfacing Open Source SystemC with a Third Party Spftware", *Proc. of DATE 2001*, Munich Germany, March 2001.

[10] V. Nema, et al., "Optimised MAC Policy for a Dedicated Short Range Communication System", *Proc. of IEEE International Conference on 3rd Generation Wireless and Beyond*, San Francisco, CA, June 2001.

[11] R.Cmar, et al., "Platform Design Approach for Reconfigurable Network Appliances", *Proc. of CICC 2001*, San Diego, CA, May 2001.

[12] Ghassan Fayad, and Karim Khordoc, "An Object-Oriented Refinement Methodology through the Design of a Settop-Box", *Proc. of 2000 Canadian Conference on Electrical and Computer Engineering*, Halifax, Canada, March 2000.

[13] S. Meyers, "More Effective C++", Addison-Wesley, 1996.

[14] E. Gamma, et al., "Design Patterns", Addison-Wesley, 1999.