SystemC Object-Oriented Extensions and Synthesis Features

Eike Grimpe[1], Bernd Timmermann, Tiemo Fandrey, Ramon Biniasch, Frank Oppenheimer
OFFIS Research Institute
Escherweg 2, 26121 Oldenburg - Germany
[1]Phone: +49 441 97 22-2 28 - Fax: +49 441 97 22-2 82
E-mail: Eike.Grimpe@offis.de

Abstract

In this paper we present an overview about the object-oriented hardware description language SystemC-Plus that is based on SystemC. First we discuss the chances lying in the application of object-oriented techniques for designing hardware. Afterwards we give an introduction into SystemC-Plus' basic features and depict their use by means of some descriptive examples. And finally we will give a brief overview of the synthesis framework that is necessary to make hardware starting from object-oriented models.

1 Introduction

Managing the steadily increasing complexity of embedded systems is today one of the most serious challenges in the area of electronic design automation (EDA). While the technological possibilities and together with them the requirements on systems grow, the designer's productivity does not keep pace, widening the so-called design gap. One major problem in the existing design process is the step from the initial specification to the first implementation. This is in particular a problem for hardware, since system specification often starts with a C/C++ description of the system, mostly referred to as "golden model". While several compilers and development environments exist, helping to bring the software parts of a system to a specific target architecture, the step from the initial specification to an implementation in hardware is still a huge and complicated one. This step does not only require to manually transform the initial C/C++ hardware description into a behavioral equivalent hardware description in an HDL like VHDL or Verilog, but it means also to deal with the different characteristics of hardware and software.

Since SystemC [14, 13, 9] is based on C/C++ it is no longer necessary to translate an initial C/C++ description into another language, but though the design languages are the same, the semantics may strongly differ. The effort of transforming a plain C/C++ description into a behavioral equivalent and synthesisable SystemC description nearly stays the same as when translating into VHDL. The main reason for that and one of the most important disadvantages of SystemC is, that it does not provide a higher level of abstraction for modelling as used from VHDL or Verilog when targeting automated synthesis. All high level language constructs like non-primitive channels that are offered by SystemC must be refined manually, down to a cycle accurate hardware model that uses signals for communication. This is not just a specific problem of the langauge itself but also a result of existing synthesis techniques and tools. SystemC-Plus [1, 5] and the accompanying synthesis framework are lined up, to overcome this problem, and to provide the designers with the possibility to use powerful object-oriented techniques for the development of hardware combined with automatic hardware synthesis.

2 Perspectives

One may argue why to use object-oriented techniques for modelling hardware. But why do programmers today mainly use programming languages like C++, Java^{TM}and other object-oriented languages and not still FORTRAN or even assembler? The answer is, that object-orientation is a powerful instrument for mastering and implementing complex systems. Of course, there are differences between the natures of software and hardware, but many problems are similar and thus solutions to solve them can be adopted. The differences may require for specific flavors of object-oriented modelling, however, object-orientation can not also be very helpful in mastering complex software but also in mastering complex hardware and even combined hardware/software systems.

Since the general advantages that are offered by object-orientation - and not to forget also its disadvantages - have been already discussed in full detail for software applications, we do not want to repeat these discussions in this work. But what we want to point out from these considerations is, that classes are a first class concept for reuse, that classes together with inheritance are a fine concept for modelling new data types and for organizing and encapsulating data, and finally that polymorphism based on a class and inheritance concept is a powerful instrument to make an implementation flexible and easy at the same time. These observations mainly motivate our researches on the field of object-oriented hardware design.

3 SystemC-Plus

SystemC-Plus was developed within the framework of the European Commission's IST FP5 project ODETTE [8]. It is completely based on SystemC, but its main focus lies on synthesis. The SystemC synthesis subset that is accepted by synthesis tools like the CoCentric® SystemC Compiler [11, 12] can be seen as a subset of SystemC-Plus. Based on this subset object-oriented constructs accompanied by modelling guidelines are added. But only such object-oriented constructs are introduced, for which we have a clear synthesis semantics. Section 4 will give a brief overview of our synthesis concepts.

Because of focussing on synthesis, SystemC-Plus does not just simply mean to combine C++ with SystemC. Of course, the basic class concept of C++ is adopted by SystemC-Plus, but its polymorphism mechanism could not be adopted one-to-one. The problem of C++' polymorphism concept is, that it only works by means of pointers, and pointers are, despite from a very few exceptions, banned from SystemC-Plus, as well as dynamic memory allocation and de-allocation. The reason is the static nature of hardware and the problem of efficient pointer synthesis, which we regard not to be solved. For the same reason also other C++ constructs were restricted in use or even banned, because of not finding an appropriate way to synthesize them.

Building a SystemC-Plus simulation model is as easy as for SystemC. The SystemC-Plus description does only have to be compiled with a common more or less ANSI C++ compliant compiler like the gcc, that is available on a variety of platforms. The resulting executable can then directly be run for simulation. Including the SystemC class library is also necessary for SystemC-Plus since it only augments SystemC by additional features. Some of these features, like polymorphic (see 3.1.2) and global objects (see 3.1.3) additionally require to include the so-called OOHWLib (Object-Oriented HardWare Library), that was also developed within the framework of ODETTE. Figure 1 illustrates how simulation and synthesis start from an object-oriented SystemC-Plus description.

3.1 Basic Features

SystemC-Plus is mainly characterized by supporting the following high level modelling constructs for synthesis:

- classes and objects, also including the concept of inheritance
- polymorphic objects
- global objects, for modelling shared resources and communication
- templates, for implementing highly parameterizable components

We will discuss each of this features in more detail in the following sections and demonstrate their use by means of simple but descriptive examples.

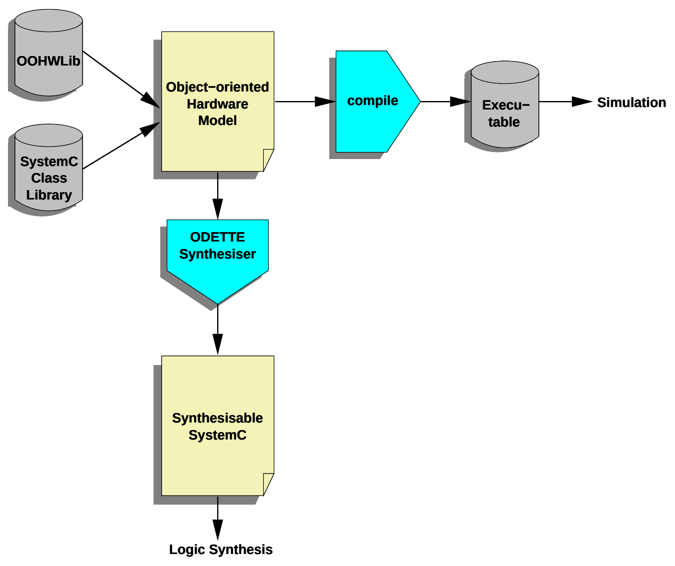

Figure 1: Synthesis and simulation starting from SystemC-Plus

3.1.1 Classes and Objects

As used from C++, SystemC-Plus allows the user to model new data types by means of classes. Classes are a first class concept for modelling reusable components. Take for example the complete [1] class declaration of a simple counter in the following:

Listing 1: Sample implementation of a counter class

```
class Counter{

public:
  Counter( const unsigned int lowerBound = 0,
      const unsigned int upperBound = 255,
      const unsigned int initialValue = 0 ) {
    m_counter = initialValue;
    m_lowerBound = lowerBound;
    m_upperBound = upperBound;
    m_overflow = false;
  }

  bool overflow() {
    return( m_overflow );
  }

  void clearOverflow() {
    m_overflow = false;
  }

  void setLowerBound(
      const unsigned int lowerBound ) {
    m_lowerBound = lowerBound;
  }

  void setUpperBound(
      const unsigned int upperBound ) {
    m_upperBound = upperBound;
  }
  void reset() {
    m_counter = m_lowerBound;
    m_overflow = false;
  }

  unsigned int getValue() {
    return( m_counter );
  }

  void count(
      const unsigned int stepSize = 1 ) {
    unsigned int diff =
      m_upperBound - m_counter;
    if ( diff < stepSize ) // counter overflow
    {
      m_counter =
        m_lowerBound + ( stepSize - diff ) - 1;
      m_overflow = true;
    } else {
      m_counter += stepSize;
    }
  }

private:
  unsigned int m_lowerBound;
  unsigned int m_upperBound;
  unsigned int m_counter;
  bool overflow;
};
```

[1] Note that the implementation is not a very safe one, because it would be possible to set the counter to a value lower than the specified lower bound. Such cases may be handled by exceptions, which must be switched off for synthesis, since we are not performing synthesis of exception handling.

It is not only possible to easily reuse the counter in a variety of different systems, but its implementation also satisfies varying flavors of handling the counter and also different requirements, like a flexible step size and counting range. Though the above example may not be very impressive for a C++ programmer, the possibility to use it for modelling hardware is impressive. Instances of class *Counter* shown above could be used in every SystemC-Plus model for direct synthesis.

One thing, that must be clearly stated is, that objects in contrast to modules are always passive. That means they must be triggered from outside to be active because they lack an own thread of control. If the counter modelled in the above example should count events, its *count()* method must be triggered within a process, that is sensitive to these events. Active components can only be modelled by means of modules. However, objects can ease the modelling of the functionality that is inherent in a module as illustrated by Listing 2. In the listing an instance of the *Counter* class is used to count clock-events. After each 256 events - according to the default constructor arguments - an overflow is indicated and an alarm signal is set for one clock cycle. Assume *countProcess()* to be declared as synchronous process - an *SC_CTHREAD* - in some SystemC module.

Listing 2: Using class Counter

```
void countProcess() {
  Counter cntr;
  if ( reset == true ) {
    cntr.reset();
    alarm.write( false );
  }
  wait();
  while ( true ) {
    cntr.count();
    if ( cntr.overflow() ) {
      alarm.write( true );
      cntr.clearOverflow();
    } else {
      alarm.write( false );
    }
  wait();
  }
}
```

3.1.2 Polymorphism

Polymorphism in SystemC-Plus works by means of so-called polymorphic objects. A polymorphic object can change its class type dynamically during runtime. The main difference between the native C++ polymorphism mechanism and SystemC-Plus' polymorphic objects is, that the latter ones provide their own state space. That means, that an assignment to a polymorphic object does not simply mean to bend a pointer from one object to another, but to assign a real copy of the source object to the target object. It was necessary to introduce this special flavor of polymorphism, since we have a clear synthesis semantics for it, but not for the C++ mechanism. The main problem in the case of the native C++ polymorphism mechanism is the handling of pointers, which we do not claim to be efficiently synthesisable as already mentioned previously.

Polymorphism is an extremely powerful object-oriented feature. Its usefulness can be best demonstrated by means of a example. First, assume the simple class hierarchy, shown in the following Figure, that models arithmetic operations.

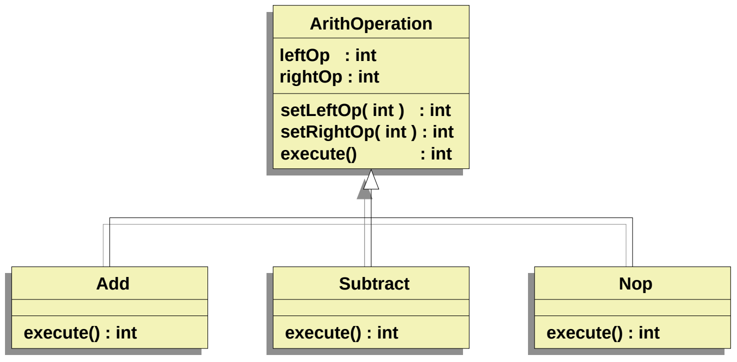

Figure 2: Arithmetic operations modelled as class hierarchy

This way of modelling arithmetic operation in an object oriented way together with polymorphism now allows to implement very easy a very flexible arithmetic unit, as demonstrated next:

Listing 3: A polymorphic arithmetic unit

```
SC_MODULE( ArithmeticUnit ) {
  sc_in_clk  clock;
  sc_in< bool > reset;
  sc_in< PolyObject< ArithOperation > > operation;
  sc_in< int > leftOp;
  sc_in< int > rightOp;
  sc_out< int > result;

  void executeOp() {
    PolyObject< ArithOperation > op;
    if ( reset == true ) {
      result.write ( 0 );
    }
    wait();
    while ( true ) {
      op = operation.read();
      op->setLeftOp( leftOp.read() );
      op->setRightOp( rightOp.read() );
      result.write( op->execute() )
      wait();
    }
  }

  SC_CTOR( ArimethicUnit ) {
    SC_CTHREAD( executeOp, clock.pos() );
    watching( reset.delayed() == true );
  }
};
```

The *execute()* operation in the above listing is dynamically dispatched, that means it is determined at runtime, which implementation of the function is actually invoked on a call. It depends on the actual class type of the polymorphic object that is passed through the input-port *operation*, which implementation is taken. If an instance of class *Add* is sent, its *execute()* method is invoked, if an instance of *Sub* arrives, its *execute()* method is invoked. It does not matter which kind of arithmetic operation is sent, as long as it provides an appropriate implementation of *execute()*.

It is also very easy to add new functionality to *ArithmeticUnit* without even having the need for touching its implementation. Imagine, that a multiplication should be added to the set of arithmetic operations. The only thing to be done is to derive a new class, call it *Multiply*, from *ArithOperation* and to overload its *execute()* function accordingly. Now, *ArithmeticUnit* is able to also handle multiplication operations without having touched its implementation. Clearly spoken, with this mechanism it is possible to introduce new functionality into a module from outside and, in the ideal case, without having knowledge of the implementation details of the module itself.

And finally, though polymorphism is a feature that can make synthesis quite complex, it also offers some interesting potential for optimization. If a synthesis tool can statically determine, which set of classes and especially which set of different method implementations are actually used with a polymorphic object, it can prevent the other method implementations from synthesis. Take again a look to the arithmetic operations example above. If a synthesis tool can determine, that only instances of class *Add* are sent through the *operation* port, it will not synthesize any circuits for the other operations. That means the arithmetic unit is automatically optimally tailored for specific requirements just by usage, without having the need for doing manual adaptations. But this kind of optimization depends on statical analysis and is limited, if the functionality of systems parts depends on the environment of a system, that is not taken into account for synthesis.

A polymorphic object in SystemC-Plus is simply declared as follows:

PolyObject<RootClass> polyObj;

This can also be seen in lines 4 and 10 in Listing 3. The assignment rules are the same as used from C++, for example instances of the root class and instances of all classes derived from the root class can be assigned to a polymorphic object, provided, that an appropriate assignment operator is available. The only requirement on the root class and all objects that are assigned to a polymorphic object is, that their class declarations include a polymorphic specifier:

Listing 4: The polymorphic specifier

```
class ArithOperation {
  POLYMORPHIC( ArithOperation );
   ...
};

class Add : public ArithOperation {
  POLYMORPHIC( Add );
   ...
};
```

Members of a polymorphic object are always accessed by means of the −> operator, indicating that the access is dynamically dispatched. But, as used from C++, only those functions are actually dynamically dispatched, which are declared virtual.

3.1.3 Global Objects and Communication

Communication and data exchange between concurrent components in hardware is usually based on signals. That does not only mean that data is exchanged via busses or serial links, it does also mean that every activity is triggered by activating a certain signal or by assigning a certain bit pattern to a bunch of signals. This kind of interface is not very handy, because just a bit pattern usually does not speak for itself. A method based interface does provide a lot of advantages. For example an interface that consists of the methods *add()*, *subtract()*, *multiply()*, and *divide()* is much clearer than the bit patterns "00", "01", "10" and "11". In the first case a designer can figure out what kind of operation he is actually invoking just by having a look at the code. In the latter case an assignment of two bits to two signals does not explain anything. And if an interface does not only consist of two signals but of a few dozens, it becomes even more complicated. Also the risk is higher, that a wrong operation is triggered only because of accidently toggling or permuting some bits.

Though a method interface would be preferable, it is not just possible to let concurrent components communicate through normal C++ objects, because these objects do not provide mechanisms for handling concurrent accesses. Just accessing an instance of a class that may be declared as a member of a module from within different concurrent processes at the same time, would not produce any observable conflict but would also show absolutely improper behavior in the sense of hardware. For that reason, SystemC-Plus introduces its so-called global objects. Global objects are similar to Ada95's [2] protected objects which are mainly used to communicate between concurrent threads. Global object posses built-in mechanisms for arbitrating concurrent accesses and guarantee for mutual exclusive access. They also have a certain notion of time, in particular of the communication protocol, wherefore their behavior is simulated cycle accurate during a SystemC-Plus simulation run. As a tribute to keeping automated synthesis, the applied communication protocol is fixed, but the scheduling can be determined by the designer within certain borders.

The following listing illustrates the application of global objects by means of a very simple producer/consumer example. The consumer generates somehow a random number each clock cycle and puts it into a bounded buffer that is shared between the producer and the consumer process. A consumer process periodically gets an element out of the buffer. Mutual exclusive access is guaranteed by the global object.

Listing 5: Using global objects

```
 1  SC_MODULE( ProdCons ) {
 2
 3    GlobalObject< RoundRobin, FiFoBuffer< int, 8 > > sharedBuffer;
 4
 5     sc_in_clk          clock;
 6     sc_in< bool >      reset;
 7    sc_out< int >       output;
 8
 9    void producer() {
10      sharedBuffer.subscribe();
11      if (reset == true) {
12        sharedBuffer.reset();
13      }
14      wait();
15      while ( true ) {
16        GLOBAL_PROCEDURE_CALL( sharedBuffer, put( randomNumber() ) );
17        wait();
18      }
19    }
20
21    void consumer() {
22      int val;
23      sharedBuffer.subscribe();
24      if ( reset == true ) {
25        output.write( 0 );
26      }
27      wait();
28      while ( true ) {
29        GLOBAL_FUNCTION_CALL( sharedBuffer, get(), val );
30        output.write( val );
31        wait();
32      }
33    }
34
35    //Constructor
36    SC_CTOR( ProdCons ) {
37      SC_CTHREAD( producer, clock.pos() );
38      watching(reset.delayed() == true);
39
40      SC_CTHREAD( consumer, clock.pos() );
41      watching(reset.delayed() == true);
42    }
43  };
```

A global object requires two template parameters to be specified at declaration. The first one must denote a scheduler, which determines the scheduling strategy that is applied on scheduling concurrent accesses, and the second one may be any synthesisable user-defined class for implementing the actual functionality of the global object. Declaration of a global object is shown in line 3 of the above example. The used scheduler, *RoundRobin*, is one that is already provided by the OOHWLib and which implements a round robin scheduling strategy. A user may also define his own schedulers by deriving them from a special base class and with respect to some coding guidelines for their implementation. The second argument passed in the example is assumed to be some kind of parameterizable FIFO buffer, which can store up to eight elements of type int. A designer may freely pass his own classes instead.

One requirement that must be satisfied by all user-defined classes that are passed to a global object is, that each of its member functions being called from outside the object - in the example above these are the functions *put()* and *get()* - must be declared as so-called guarded methods. A guarded method declaration associates a certain guard condition with a method. Only if the associated guard condition is true, a caller of that method will be regarded for scheduling, otherwise it is blocked. This mechanism allows atomic test and access for global objects. For example the guard condition of the *get()* method should check, if there are still elements left in the buffer, the guard condition of the *put()* method should check, if the buffer is not already full. Every client that calls a method on a global object, but does not succeed, because it calls a function whose guard condition is actually false or because it looses arbitration, is blocked at the call site, until it finally succeeds. Note, that a client may be even blocked forever.

Global objects can be bound to other global objects of the same type across module boundaries, analogue to a port binding. It is therefore not only possible to connect processes within the same module by global objects, but also processes that are located in different modules, spread over the whole system hierarchy. This feature allows in principle to equip a module with a method interface instead of a signal based interface, since it is possible to make its functionality accessible through a global object.

But the modelling comfort offered by global objects comes along with some limitations. For instance global objects must only be accessed from within *SC_CTHREADs*, driven by the same clock net. Furthermore an access to a global object does always need at least three clock cycles as a tribute that has to be paid to the fixed communication protocol and to automatic synthesis.

3.1.4 Templates

Templates are not directly an object-oriented feature. VHDL, for example, provides the possibility to use generic parameters for implementing parameterizable entities. But C++' template mechanism combined with object-orientation offers much more modelling power, especially because C++ allows to pass types and in particular class types as template arguments and not only scalar values, expressions respectively, as in VHDL.

Since classes may comprise a significant amount of functionality, the behavior of a module or even a system may be completely changed just by passing varying classes for instantiation. One obvious application is to provide different implementations of a functionality for different purposes. Taking for example the arithmetic operations from above, two different *Multiply* classes may be provided for use. One class that implements a fast multiplication algorithm that would result in a larger circuit after synthesis and another implementation, that implements a slower multiplication algorithm, but which would need less area. Dependent on specific needs, one of these implementations may be chosen and passed as a template argument to a module that is accordingly modelled as a template.

Templates also serve for modelling flexible data containers as demonstrated in line 3 of Listing 5. The buffer being passed as argument to the global object declaration is modelled as a template, whose size and element type can be arbitrary chosen just by passing different template arguments.

4 Synthesis

SystemC-Plus main focus lies on providing synthesisable object-oriented features. Since there are already sophisticated synthesis techniques existing, starting from RT or behavioral level, the synthesis we are proposing translates an object-oriented input description into a description that can be processed by existing logic synthesis tools without any problem. Therefore we are mainly concentrating on removing the object-oriented constructs of the initial description by replacing them by behavioral equivalent constructs on RT or behavioral level. The synthesis step we propose will not directly produce a gate net list and can be seen as high level synthesis.

The basic idea of our synthesis [10, 4] approach is the translation of objects into bit vectors, which can be easily processed by existing logic synthesis tools, like the CoCentric® SystemC Compiler. As shown in Figure 3, each data member of an object is individually encoded as a bit vector first and then the concatenation of all encoded data members forms the state space of a synthesized object. Each data member can still be identified as slice of the bit vector being synthesized for an object, wherefore accesses to these members can be translated into accesses to the appropriate slice. Likewise all member functions must be translated into a form, that operates on the synthesized bit vector instead of the original object.

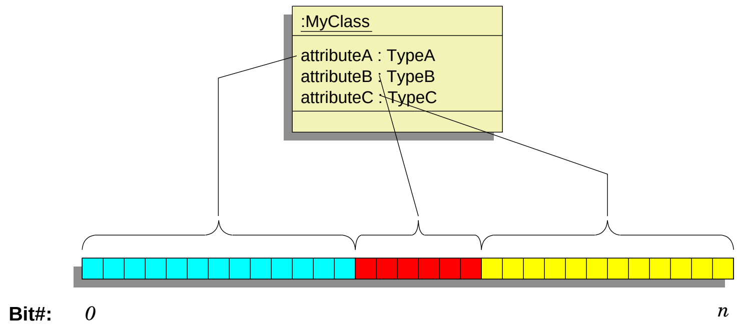

Figure 3: Transforming an object into a bit vector

If a polymorphic object or an object that is assigned to a polymorphic object is synthesized, the resulting vector also is augmented by a tag, which is used to identify the actual class membership of an object at runtime. By means of this tag, the appropriate implementation for a dynamically dispatched function, i.e. declared virtual, is chosen during a function call.

For a global object and its clients, the structure being outlined in Figure 4 is synthesized. Calls to a global object are transformed into a handshake protocol based on signal communication. The channels in the figure comprise the necessary handshake signals and also busses for arguments and return values. An arbiter that schedules concurrent accesses to the former global object - the server in Figure 4 - is also automatically synthesized. The scheduling functionality is generated according to the scheduler that was specified at a global object declaration. The functionality of the server is synthesized from the user defined class that was passed at the global object declaration.

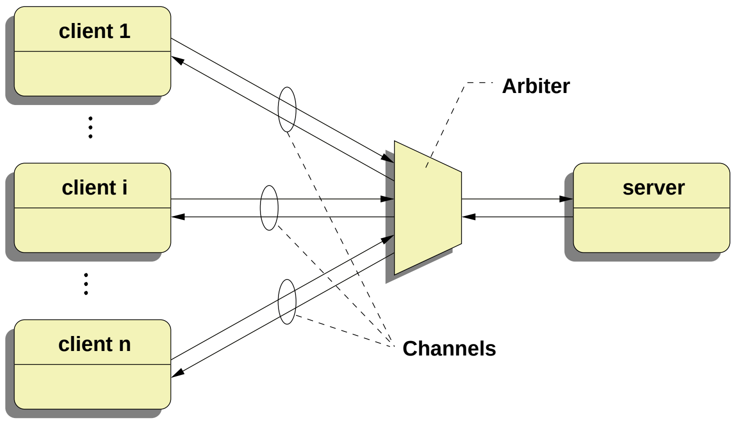

Figure 4: Synthesized client/server structure

As mentioned before, SystemC-Plus is focussing on synthesis. That means not every kind of C++ code, for example classes from the STL, can just be taken and included in a SystemC-Plus model. There are some restrictions as tribute to the statical nature of hardware. The most striking restrictions may be the ban on using pointers in general and the ban of dynamic memory allocation/deallocation. The first restriction was introduced because we claim pointers not to be efficiently synthesisable. The second restriction follows from the first one and also from the fact, that circuits can not simply be newly created or destroyed at runtime in hardware. Possibly reconfigurable hardware does provide a new perspective for the future but this is out of the scope of our actual work. All the different restrictions that have to be regarded for modelling with SystemC-Plus are summed up in an LRM [1], that is not standardized but available to the public [8]. Additional restrictions may have to be considered dependent on the back-end synthesis tool, that is targeted.

Another topic for synthesis of object-oriented models is optimization. Object-oriented models tend towards a certain overhead regarding area and timing when not applying sophisticated optimization techniques. But there is also certain potential for optimizations and for reducing the produced overhead significantly. But even if a certain overhead will remain, and the synthesized designs may be not quite as efficient in resource usage as a design that was completely handcrafted, we claim this to be acceptable, if the productivity of the design process can be significantly increased by applying object-oriented techniques. We will not go into further details here, since optimization techniques are out of the scope of this paper.

4.1 ODETTE Synthesis Tool

To proof, that the synthesis techniques outlined in the previous chapter are not just of theoretical nature, a synthesis tool that performs the proposed synthesis of object-oriented constructs is actually under development within the framework of the ODETTE project. The final prototype is scheduled for the end of this year. The prototype is expected to already support all features listed in the previous sections. It should also be able to apply various optimization techniques specific to object-oriented modelling. The input of the synthesis tool is a SystemC-Plus model, the generated output is a SystemC description, that can be processed by the CoCentric® SystemC Compiler. Prototypes that will first not support all features listed above are scheduled for the near future. The synthesis tool and design methodologies based on SystemC-Plus will be evaluated by industrial partners from the automotive and telecommunication area.

5 Conclusions

In this work we have shown, that object-orientation offers some very interesting new perspectives for designing hardware and that it has the potential to increase the productivity in hardware design. We have also shown, that automatic synthesis of object-oriented constructs can actually be performed.

References

[1] P. J. Ashenden, R. Biniasch, T. Fandrey, E. Grimpe, A. Schubert, T. Schubert, *Input Language Subset Specification (formal)*, ODETTE Deliverable, 2001, http://odette.offis.de

[2] J. Barnes, *Programming in Ada95, 2nd Edition*, Addison-Wesley, 1999

[3] M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[4] E. Grimpe and F. Oppenheimer, *Aspects of Object-Oriented Hardware Modelling with SystemC-Plus*, Proceedings of the FDL'01, Lyon, France, September 2001.

[5] E. Grimpe and F. Oppenheimer, *Object Oriented High Level Synthesis Based on SystemC*, Proceedings of the ICECS 2001, Malta, September 2001.

[6] IEEE Std 1076, 2000 Edition, *IEEE Standard VHDL Language Reference Manual*, IEEE, 2000

[7] IEEE Std 1076.6, *Standard For VHDL Register Transfer Level Synthesis*, http://www.eda.org/siwg/

[8] ODETTE - Object-oriented co-DEsign and functional Test TEchniques, IST project of the Commission of the European Communities, http://offis.odette.de

[9] Open SystemC Initiative, http://www.SystemC.org.

[10] M. Radetzki, *Synthesis of Digital Circuits from Object-Oriented Specifications*, Dissertation at University of Oldenburg, 2000

[11] Synopsys, Inc., *CoCentric® SystemC Compiler Behavioral Modeling Guide*, 2001

[12] Synopsys, Inc., *CoCentric® SystemC Compiler RTL User and Modeling Guide*, 2001

[13] Various contributors, *Functional Specification for SystemC 2.0 - Final - Version 2.0-M*, 2001.

[14] Various contributors, *SystemC Version 2.0 Users's Guide*, 2001.