

Object Oriented Hardware Synthesis and Verification

T. Kuhn, T. Oppold,
C. Schulz-Key,
M. Winterholer,
W. Rosenstiel
Univ. of Tuebingen, Sand 13
72076 Tuebingen, Germany
oase,rosen@informatik.
uni-tuebingen.de

M. Edwards
Cisco Systems, Inc.
7025 Kit Creek Road
RTP, NC 27709
jme@cisco.com

Y. Kashai
Verisity Design, Inc.
2041 Landings Drive
Mountain View, CA 94043
yaron@verisity.com

ABSTRACT

The synthesis of hardware from object oriented specifications is presented. Our approach utilizes the \mathcal{E} language that has been proven to be highly efficient for the verification of hardware. The \mathcal{E} language is similar to Java and provides additional constructs for specification and verification of hardware. We describe an automated design flow for the synthesis of object oriented descriptions that tightly integrates simulation based verification. The usability of our approach is demonstrated by real-world examples.

Keywords

Object oriented hardware modeling, verification, high-level synthesis.

1. INTRODUCTION

There is a growing gap between the number of gates that can be implemented on a single chip and the number of gates that can be designed by one person in one day. Additionally, short product life cycles, time to market, and changing standards require shorter design times. Therefore, new design methodologies must be applied. By specifying on a higher level of abstraction the productivity can be increased enormously. Commercial tools for synthesizing algorithmic descriptions are available already and gain more and more acceptance. Newer approaches raise the abstraction level even more by synthesizing object oriented descriptions. The object oriented paradigm can be applied in order to cope with the complexity of system-level designs. This paradigm also simplifies re-use of IP.

Since hardware description languages like VHDL and Verilog are more suitable for RT level design than for higher levels of abstraction, software languages like C/C++ or Java are recently deployed for synthesis. These languages are already widely used for specifications that can be executed. Furthermore, hardware/software co-design can be simplified by using a single language for both domains.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.
Copyright 2001 ACM 1-58113-418-5/01/0010 ...\$5.00.

Verification typically consumes over half of the design effort. The languages currently used for synthesis provide only insufficient support for verification. Therefore languages like Verisity's \mathcal{E} (www.verisity.com) or Synopsys' Vera (www.open-vera.com) hardware verification language are deployed within a dedicated environment to improve the verification task. The \mathcal{E} language is similar to Java enhanced by various verification and hardware related constructs. \mathcal{E} provides all basic object oriented constructs, supports *multi-threading*, and it can be executed in the SpecmanTM environment. Therefore it is also well suited for specification.

Synthesis on the system-level requires support for multi-threaded descriptions. Due to the complexity of such designs, we want to support verification appropriately in our approach. The powerful verification features of \mathcal{E} and its capability for object oriented specification, encouraged us to investigate the applicability of the \mathcal{E} language for synthesis. By making the synthesis of \mathcal{E} descriptions possible, we allow to use one single language for specification, verification, and synthesis.

The remainder of the paper is organized as follows: In Section 2, we will present an overview of related work. Section 3 describes object oriented synthesis and details of the synthesis from \mathcal{E} descriptions. In Section 4, we present verification within the Specman environment and the integration of this environment in our design flow. Section 5 presents examples and discusses the results. Section 6 concludes this paper and gives an outlook on further work.

2. PREVIOUS WORK

In the past, various approaches have been made to specify hardware and hardware/software systems on the basis of objects. Object oriented VHDL [1, 2] enhances VHDL with object oriented concepts. SystemC [3] is based on C++ class libraries and is intended for design at the system/algorithmic and register-transfer levels. Recently, successful efforts have been made to specify [4, 5, 6] and to synthesize [7, 8] from Java.

For validation, simulation and formal verification have a long tradition and various commercial tools are provided by major EDA companies, e. g., Cadence, Mentor Graphics, and Synopsys.

3. OBJECT ORIENTED SYNTHESIS

The object oriented design paradigm has already gained broad acceptance in software engineering for the design of large systems. Object oriented synthesis allows hardware modeling on an abstraction level above the behavioral level. Designers can express their ideas in a very natural way by thinking of classes rather than of data and procedures. Encapsulating things that belong together and

having data and behavior within one entity also simplifies re-use.

We developed an object oriented analysis system (OOAS) that allows to synthesize object oriented hardware descriptions. Our OOAS can be used to read in descriptions written in different languages. Currently we support Java, SystemC, and \mathcal{E} as input languages. In this paper we describe the synthesis from \mathcal{E} .

There is already a variety of good synthesis tools on the market. These tools provide an automated design flow for RT and algorithmic descriptions, but they do not support object oriented concepts like object references, inheritance, or polymorphism. We utilize the algorithms deployed by these tools for the synthesis of object oriented descriptions by transforming such descriptions into equivalent descriptions on a lower level of abstraction that can be handled by commercial tools. The formats that we generate for those descriptions are VHDL, Verilog, and SystemC (figure 1). For the sake of simplicity this paper refers only to the output of Verilog.

3.1 Synthesis from \mathcal{E}

Using one single object oriented language for specification, verification, and synthesis simplifies the design process. There is a variety of languages that could be used as such a unified language, each of them having advantages and disadvantages. Most of the languages do not support verification adequately. Verification of multi-threaded descriptions is mandatory for hardware design and particularly challenging. The \mathcal{E} language is therefore especially suited as a unified language for hardware design. Today \mathcal{E} is not only used for verifying hardware descriptions written in VHDL or Verilog, it is already used for writing a *golden model* of the hardware in an object oriented high-level language. This model is then manually translated into a synthesizable HDL description. The advantage of \mathcal{E} over software languages like C/C++ or Java is that hardware related constructs like bit accurate data types, clocks, etc. are part of the language.

For the transformation from an object oriented description in \mathcal{E} into an equivalent description in Verilog, the meta data that describe the object oriented structure is generated. The number and structure of objects have to be static in the realization in hardware. This should be considered in the description and is checked by pre-allocation during the analysis. Objects have to be allocated for variables and parameters of methods which reference sets of objects. The identification of such sets of static references is done during an alias analysis, which is part of the OOAS.

In \mathcal{E} concurrent threads of control are described by so called *Time Consuming Methods* (TCMs). TCMs are similar to Threads in Java. TCMs are triggered by events. Events can be defined by temporal expressions that are part of the \mathcal{E} language. TCMs have to be handled separately from the regular methods (non-TCMs) dur-

ing the transformation process. Conflicts resulting from concurrent access to variables by different TCMs must be detected by a concurrency analysis and then be resolved. We now describe the control data flow analysis that is performed for the sequential case and the concurrency analysis that is performed for the parallel case.

3.2 Control Data Flow Analysis

The control data flow analysis applies techniques known from software compiler design [9] to the synthesis of hardware.

After the lexical and semantical analysis done by the scanner and parser for \mathcal{E} , a static control and data flow analysis is performed on the set of syntax trees for each \mathcal{E} object. The result of the analysis is a control flow graph (CFG) where the data flow information is stored in a scope table within each node of the graph. Because of the combination of data and control flow information, no separate data flow graph has to be generated and the analysis traverses the syntax tree only once. The CFG has two different types of nodes. The control flow nodes divide the CFG in multiple sub-trees if a node has multiple children. This is the case for nodes representing, e.g., *while* loops or *if* statements. The second type of nodes in the CFG are the data flow nodes, which do not change the control flow, like arithmetic operations or assignments.

The main problem of the transformation is the usage of references. Deciding which object is accessed when a method is called on a variable can only be done at runtime. Objects may have several references, or aliases at the same time, so it is hard to tell which statements affect which object.

The analysis determines a set of possible objects for each variable within the scope of a statement. Each node of the CFG has one scope table, where all variables in the scope of the node are stored together with a set of objects that may be referenced by this variable. The scope tables are built together with the whole CFG. When a new node is created, the scope table of the parent node is cloned and the set of references ('reference set') of each variable changed by the statement is updated.

A reference set $R_s(a)$ is the maximal set of all references on objects, which can be hidden by the alias a after the execution of statement s under consideration of the type of the variable and the preceding control flow. All reference sets $R_s(a)$ of variables accessible in a CFG node n , build the scope table $S(n)$.

Figure 2 shows an example of a hardware component written in \mathcal{E} . Classes are described using the keyword *struct*. A struct declares data fields and methods. Inheritance is supported by the keyword *like*. The *init* method is similar to the constructor in Java. Instances of a struct are made by using the keyword *new*. A detailed description of the \mathcal{E} language can be found in [10]. The constructs that we use for the specification of hardware components are described in [11].

The analysis for the TCM *runHWO* of the example code results in the CFG shown in figure 3. At node 5 in the CFG, the scope table includes three reference sets $R_5(x)$, $R_5(y)$ and $R_5(z)$. Each reference set has been initialized in the *init* method, where three objects of the struct *S* have been instantiated. The objects *S_1*, *S_2* and *S_3* are labeled with subsequent numbers.

For each type of statement, a different algorithm is implemented to update the scope table of a node in the CFG. The algorithm for loop statements, used for the determination of the scope table for node number 7 in figure 3 is described in algorithm 1.

Table 1 shows the evolution of the reference sets for each iteration of the algorithm. The body of the *while* loop has to be re-analyzed four times to get the final reference sets for the *while* loop node (without determining the exact number of iterations during execution).

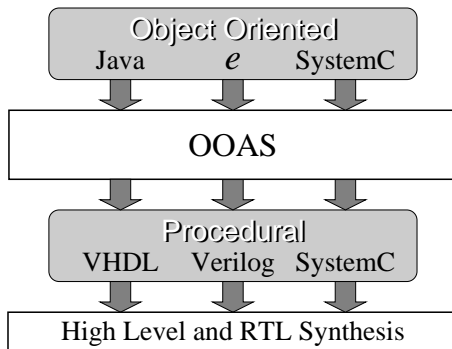


Figure 1: Languages supported by OOAS

```

struct Test like HWO {
  x : S; y : S; z : S;

  init() is {
    x = new S;
    y = new S;
    z = new S;
  };

  runHWO() @clk is {
    var i : uint;
    i = 0;
    while (i < 10)
    {
      z = x;
      x = y;
      i = i + 1;
    };
    z.foo();
  };
};

```

```

struct S {
  foo() is {
    ...
  };
};

```

Figure 2: Struct *Test* and struct *S*

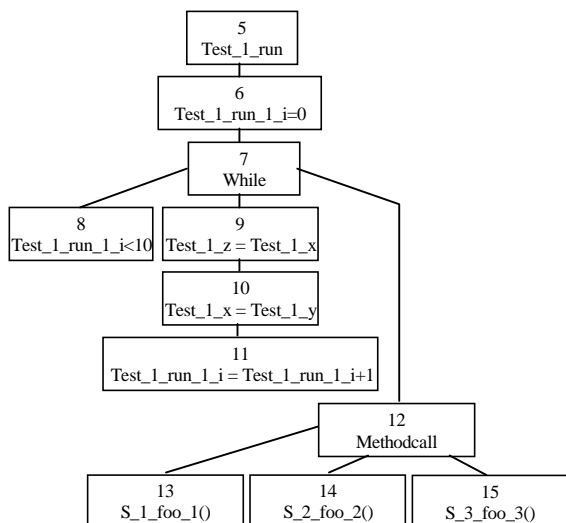


Figure 3: CFG for the TCM *runHWO*

The analysis terminates because there are no changes in the reference sets of step three and four. The merged reference sets in the last column of table 1 are the resulting sets contained in the scope table $S(7)$ of node 7.

```

Input: CFG node  $n$ , statement  $s$  of  $n$ , scope table  $S(p)$ 
where  $p$  is the predecessor node of  $n$ 
Output: Updated scope table  $S(n)$ 
begin
   $S(n) \leftarrow$  copy of  $S(p)$ 
   $S(tmp) \leftarrow$  copy of  $S(p)$ 
  repeat
     $S(n) \leftarrow S(n) \cup S(tmp)$ 
     $S(tmp) \leftarrow$  analyze the block statements of the loop
  until  $S(n) = S(tmp)$ 
end

```

Algorithm 1: Analysis of loops

Table 1: Reference sets for the while loop

	1.	2.	3.	4.	result
$R_7(x)$	{S_1}	{S_2}	{S_2}	{S_2}	{S_1, S_2}
$R_7(y)$	{S_2}	{S_2}	{S_2}	{S_2}	{S_2}
$R_7(z)$	{S_3}	{S_1}	{S_2}	{S_2}	{S_3, S_1, S_2}

The scope table $S(7)$ is the input for the analysis of node 12. The method call $z.foo()$ uses variable z where $R_{12(z)}$ contains the objects S_3 , S_1 and S_2 . On which object the method $foo()$ will be called depends on how often the *while* loop has been iterated at runtime. The analysis splits the CFG into three sub-trees, one for each object that may be referenced by the variable z .

For each method call, a new node in the CFG is created. The body of the method builds a sub-tree of the node. When the method is a TCM, the new CFG node builds an independent CFG with the method call node as root node. A copy of the reference sets of the actual scope is the initial scope of the new CFG. The analysis always terminates because of the constant number of objects in the whole system. The number of objects in a reference set can not exceed the maximum number of objects instantiated in the whole system. The constant number of objects instantiated in the \mathcal{E} program is guaranteed since the instantiation of objects in loops is only allowed if the number of iterations can be determined at compile time. The instantiation of objects is not allowed within cyclic or recursive method calls.

When the analysis terminates, the CFG is used as input for the concurrency analysis, which is described in the next section.

3.3 Concurrency Analysis

During the concurrency analysis, the reference sets and the CFGs of the data and control flow analysis are used for creating a set of variables which are accessed by different TCMs. A variable can be accessed by a TCM in two different ways:

- **write access:** The variable is on the left hand side of an assignment and becomes an alias for another object.
- **read access:** The variable is on the right hand side of an assignment or is passed to another method as parameter or a method is called on the variable.

If a node in the CFG accesses a variable the following cases have to be handled by the analysis:

1. There is no other TCM which reads or writes the variable. The variable is a normal variable that does not require any special treatment.
2. At most one TCM has write access to the variable and multiple TCMs have read access. This variable has to be declared in a global scope to enable multiple TCMs to access the referenced object.
3. More than one TCM writes to the variable. This case is reported to the user of the system. An arbiter to resolve the access must then be inserted. This arbiter is an adapted variation of the arbiter described in [12].

For the variables which are accessed by multiple TCMs (case 2) the CFG is extended by a set of global variables called *global*. To build up this set, the CFG is traversed to build a set of variables read by the TCM t called $read_t$ and a set of variables written by t , called $write_t$. When a read access occurs in a node of the TCM t ,

the variable v is added to the set $read_t$. If v is in the set $write_m$ or $read_m$, where m is an already analyzed TCM, v becomes element of $global$. In case of a write access, v is added to the set $write_t$. If v is already element of the set $write_m$ an error state is reached. When v is member of $read_m$ v becomes element of the set $global$. The actual TCMs are stored on a stack because a TCM can start another TCM. The actual TCM where a read or write access is executed, is always the top element of a stack called $tcmStack$. The described method that creates the global set of shared variables is shown in algorithm 2.

```

Input: node {root node of the CFG}
Output: global {set of global references}
begin
  tcmStack  $\leftarrow$  empty {stack with actual analyzed TCMs}
  tcmSet  $\leftarrow$  empty {set of already analyzed TCMs}
  repeat
    switch node
      case start of TCM t
        tcmSet  $\leftarrow$  tcmSet  $\cup$  tcmStack.top
        tcmStack.push(t)
      case end of TCM t
        tcmSet  $\leftarrow$  tcmSet  $\cup$  tcmStack.pop
      case statement reads variable v
        TCM t  $\leftarrow$  tcmStack.top
        read_t  $\leftarrow$  read_t  $\cup$  v
        for all TCM m in tcmSet do
          if  $v \in read_m$  or  $write_m$  then
            global  $\leftarrow$  global  $\cup$  v
          end if
        end for
      case statement writes variable v
        TCM t  $\leftarrow$  tcmStack.top
        write_t  $\leftarrow$  write_t  $\cup$  v
        for all TCM m in tcmSet do
          if  $v \in read_m$  then
            global  $\leftarrow$  global  $\cup$  v
          end if
          if  $v \in write_m$  then
            report write access to the user
          end if
        end for
      end switch
    node  $\leftarrow$  node.next
  until node is empty
end

```

Algorithm 2: Concurrency analysis

After the complete concurrency analysis, the set $global$ is added to the root node of the CFG. The CFG is traversed by the Verilog code generator that uses the scope information to generate the appropriate Verilog constructs. The TCMs that are used in \mathcal{E} to specify multiple threads of control are transformed into Verilog *always* blocks. Each *always* block is initially idle and starts its actual execution at the same time as the corresponding TCM is started in the \mathcal{E} specification. In order to obtain this behavior and to ensure a correct reset behavior of the circuit, a skeleton is built implicitly around the actual description. Within that skeleton, the Verilog statements are generated according to the information in the CFG.

4. VERIFICATION WITH \mathcal{E} AND SPECMAN

Closely related to \mathcal{E} is the verification environment Specman, both developed by Verisity Design. Simulation based verification requires the introduction of stimuli to the Device Under Test (DUT) being simulated, as well as the collection of DUT responses for

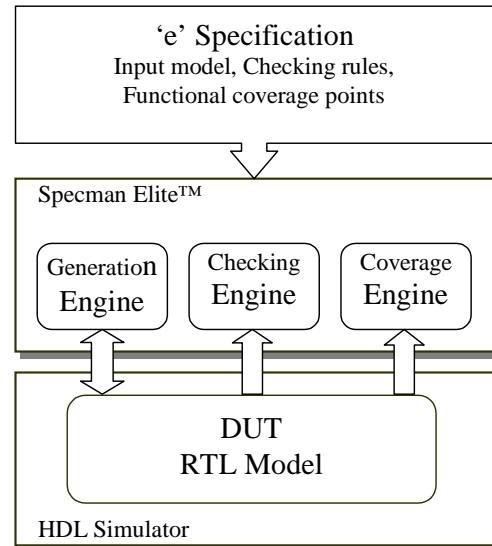


Figure 4: A typical \mathcal{E} driven testbench

the purpose of checking and coverage analysis (figure 4). Specman and the \mathcal{E} language provide powerful support for these verification tasks. The supported verification methodology can be applied to DUTs implemented in \mathcal{E} , those implemented in HDLs such as Verilog and VHDL as well as numerous other modeling languages.

We use this verification environment in three different phases of the design process depicted in figure 5.

First, the design flow starts with the specification of the hardware module in \mathcal{E} . This system specification may contain algorithmic descriptions as well as descriptions at the register-transfer level. The correctness of the specification can be verified by executing the \mathcal{E} code in the Specman environment.

Second, existing Verilog IP can be included for verification by attaching a simulator. The results of this first step in the verification process can be used to refine the test bench, also written in \mathcal{E} , around the hardware module.

Third, the \mathcal{E} specification of the hardware module is processed by our object oriented analysis system. Depending on the description style used in the specification (algorithmic or register-transfer level) the generated code is either behavioral Verilog, suitable for high-level synthesis with Synopsys Behavioral Compiler™/Design Compiler™, or a Verilog description at the register-transfer level that can be synthesized by Design Compiler. Additionally, the OOAS generates Verilog code that comprises several enhancements for simulation. The generated Verilog can be verified by attaching a simulator the same way as for the simulation of Verilog IP. The same environment is used for the verification of the results of the commercial tools.

All steps of the verification task are supported by Specman's sophisticated verification methodologies and the test bench can be re-used easily.

4.1 Input Modeling and Generation

In \mathcal{E} , input stimuli are modeled as a hierarchy of objects with inter relating constraints. By defining object types the user specifies the universe of data elements or an alphabet for an input sequence. Constraints can both remove parts of the alphabet and restrict the composition of members of the alphabet into sequences. Constraints may depend on the state of the system.

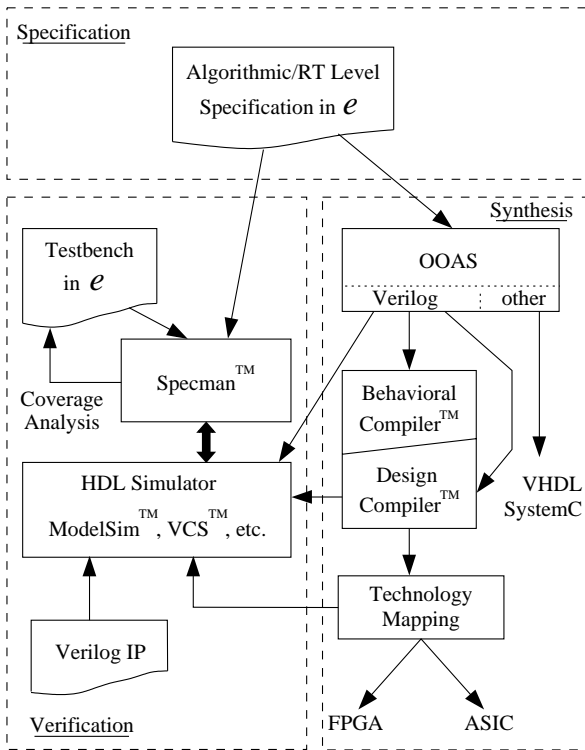


Figure 5: Design flow

In contrast to deterministic test inputs, the \mathcal{E} input model normally contains degrees of freedom. The Specman test generation engine uncovers these degrees of freedom and creates input sequences using a controlled pseudo-random process.

Constraints are conjunctive by nature, hence one can direct the generated input sequence by addition of constraints. These additions can be made per feature to be tested, or as a way to avoid areas of known defects and work in progress.

4.2 Driving and Checking

A directed random input generation methodology requires automated checking, since the input model does not predict a unique response. The \mathcal{E} language offers three major features in support of driving and checking: Events, TCMs, and a complete temporal language. The temporal language uses events and state formulae as atomic entities. Temporal and logical operators are used for expressing protocol rules the DUT must adhere to. Specman features a temporal engine that interprets temporal expressions as runtime checkers.

4.3 Functional Coverage

The generation process ensures non-zero probability for any legal input sequence. However, given the huge input space and state space of modern devices, a practical approach would control the distribution of input sequences in view of the accumulated coverage.

The \mathcal{E} language provides a way to define functional coverage metrics. Functional coverage points are user defined combinations of states, or sequences of states that have some architectural or micro-architectural significance. Because of its sequential nature, functional coverage is a more rigorous metric than code coverage.

The accumulated functional coverage and its breakdown to architectural and micro-architectural features provide status informa-

tion about the verification effort. This information is used for steering the process and to eventually certify that the DUT has a high probability of being functional.

4.4 Hardware/software Co-Verification

A key aspect of verifying SoC designs, which typically have one or more processors on board, is verifying the embedded software together with the hardware. This will flush out integration errors, beside hardware only and software only defects. In order to apply the same methodology to the integrated system it is crucial that generation, coverage, and checking are applied to the software part, as well as to the hardware part. This will facilitate tests, checks, and coverage metrics that capture hardware and software dependencies.

This requirement is achieved by the combination of Specman with a hardware/software co-verification tool, which is running the software components. Such deep integration exists for the Mentor Graphics Seamless™ tool. This integration provides Specman with the capability of reading and writing to any variable and memory location. This is the support required for the application of generated stimuli to the software parts, checking based in part on the state of the software and collecting functional coverage while taking into account the state of the software along with the hardware components.

5. EXAMPLES AND RESULTS

We used our object oriented analysis system for synthesizing several examples. One example is a part of an ATM header translator (AHT) another one is the Rana interface. The Rana interface represents a typical ‘receive’ direction, FIFO buffering scheme between three proprietary data communication interfaces. The primary incoming data interface is being buffered into two distinct FIFOs that are then being transmitted to two identical transmit interfaces. Thus the data stream is being de-multiplexed from the incoming data stream into two separate data streams. On all three interfaces handshake flow control is implemented. The Rana interface was developed by Cisco Systems [13] and is presently successfully in production.

The results of the synthesis are depicted in table 2. For each module, the number of non-comment lines of code (ncloc) of both the \mathcal{E} specification and the resulting Verilog code, and the time needed by the OOAS for the translation from \mathcal{E} to Verilog are given. The execution time of the OOAS, as well as the time spent by Behavioral Compiler and Design Compiler for the synthesis (BC/DC), were measured on a 360MHz Sun Ultra-5 with 256 MBytes RAM. The clock rate for which the module was synthesized and the area of the resulting circuit are also given in the table. The area is composed of the cell area and the net interconnect area. It is estimated by Design Compiler and given in units of the used library. We used the lca300k library, where one unit is the area of one basic cell of the LCA.

The AHT module contains almost no object oriented constructs and therefore the translation of the \mathcal{E} specification into Verilog results in a description of nearly equal size. In contrast to that, there is a great number of objects involved in the Rana module. One of these objects is a FIFO buffer used to store other objects. We have synthesized the Rana module with three different sizes for the FIFO buffer. The depth of the FIFO (i.e., the number of objects that can be stored in the buffer) is appended to the module names in table 2. While the \mathcal{E} specifications differ only in the declaration of the FIFO depth and therefore have the same size, the resulting Verilog code is growing considerably. The reason for that is, that polymorphic method calls have to be resolved to enable the transformation from an object oriented specification into procedural code.

Table 2: Results of the synthesized examples

Module	ncloc \mathcal{E}	ncloc Verilog	OO AS	BC/ DC	Clk	Area
AHT	134	132	11 sec	92 sec	40 MHz	1426
Rana3	945	1535	48 sec	32 min	20 MHz	17239
Rana8	945	1920	55 sec	1h 24m	20 MHz	24707
Rana16	945	2536	68 sec	5h 36m	20 MHz	36707

The control data flow analysis therefore determines statically for each method call on an object all possible references the variable can hold during run-time and a case statement with branches for all possible values is inserted in the Verilog code. Since the number of possible objects grows with the depth of the FIFO, the size of the generated Verilog code also increases with the FIFO depth. This effect is even more drastic as we use the technique of inlining method calls in the calling method.

6. CONCLUSIONS AND FUTURE WORK

This paper presented results from the DFG project OASE and a cooperation between Cisco Systems, Verisity Design, and the Computer Engineering Department from the University of Tuebingen. Our approach provides an automated design flow for synthesis of object oriented specifications. Verification is tightly integrated in the design flow and all the design tasks can be carried out using the object oriented language \mathcal{E} . This contribution shows that synthesis from high-level models written in \mathcal{E} is practical and that the presented analysis system is capable of dealing with real-world applications.

At present, no extensions have been made to the \mathcal{E} language in order to distinguish hardware and software objects or to define the interfaces between them. Instead inheritance was used to augment object semantics as needed. We consider adding proper constructs to the \mathcal{E} language to simplify the specification task for the designer.

Further research will be done in the area of optimization of multiple concurrent tasks within the same hardware object and hierarchies of hardware objects with no formal interface between them.

So far we have deployed our synthesis system to improve and speed up the implementation flow. When applied to the verification flow, some components of the test environment may be subject to synthesis which may facilitate test bench acceleration and post silicon validation.

We also investigate the application of object oriented concepts for hardware/software co-design partitioning[14].

7. ACKNOWLEDGMENTS

This work is funded by CISCO and DFG project No. RO 1030/8-1.

8. REFERENCES

- [1] S. Swamy, A. Molin, and B. Covnot. OO-VHDL: Object-oriented extensions to VHDL. IEEE Computer, 1995.
- [2] M. Radetzki, W. Putzke-Röming, and W. Nebel. Objective VHDL: The object-oriented approach to hardware reuse. In *Advances in Information Technologies: The Business Challenge*, 1997.
- [3] www.systemc.org.
- [4] R. Helaihel and K. Olukotun. Java as a specification language for hardware-software systems. In *IEEE/ACM International Conference on Computer-Aided Design*, 1997.
- [5] J. S. Young, J. MacDonald, M. Shilman, P. H. Tabbara, and A. R. Newton. Design and specification of embedded systems in Java using successive formal refinement. In *Proceedings of the Design Automation Conference (DAC'1998)*, 1998.
- [6] T. Kuhn, W. Rosenstiel, and U. Kebschull. Description and simulation of hardware/software systems with Java. In *Proceedings of the Design Automation Conference (DAC'1999)*, 1999.
- [7] LavaLogic. Forge-J: Fast Java to Verilog-HDL Compiler. <http://www.lavalogic.com>, 1999.
- [8] T. Kuhn and W. Rosenstiel. Java based object oriented hardware specification and synthesis. In *Proceedings of ASP-DAC*, 2000.
- [9] R. Wilson and M. Lam. Efficient context sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 1995.
- [10] Y. Hollander, M. Morley, and A. Noy. The e language: A fresh separation of concerns. In *Proceedings of TOOLS-38*, 2001.
- [11] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai. A framework for object oriented hardware specification, verification, and synthesis. In *Proceedings of the Design Automation Conference (DAC'2001)*, 2001.
- [12] J. Madsen and J.P. Brage. Modeling shared variables in VHDL. *Transactions on the ACM*, 1994.
- [13] www.cisco.com.
- [14] C. Schulz-Key, T. Kuhn, and W. Rosenstiel. A framework for system-level partitioning of object-oriented specifications. In *Proceedings of the tenth workshop on Synthesis and System Integration of Mixed Technologies (SASIMI'2001)*, Nara, Japan, 2001.