A Framework for Object Oriented Hardware Specification, Verification, and Synthesis*

T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel
University of Tuebingen
Sand 13, Germany
Phone: +49 (7071) 29 78977
{kuhn, oppold, winterho, rosen}
@informatik.uni-tuebingen.de

Marc Edwards
Cisco Systems, Inc.
7025 Kit Creek Road
RTP, NC 27709
Phone: +1 (919) 392 2233
jme@cisco.com

Yaron Kashai
Verisity Design, Inc.
2041 Landings Drive
Mountain View, CA 94043
Phone: +1 (408) 934 6855
yaron@verisity.com

ABSTRACT
We describe two things. First, we present a uniform framework for object oriented specification and verification of hardware. For this purpose the object oriented language 'e' is introduced along with a powerful run-time environment that enables the designer to perform the verification task. Second, we present an object oriented synthesis that enhances 'e' and its dedicated run-time environment into a framework for specification, verification, and synthesis. The usability of our approach is demonstrated by real-world examples.

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

1. INTRODUCTION
The ever increasing complexity of hardware systems along with the growing importance of hardware/software systems and FPGA-based reconfigurable systems impose great demands on the hardware designers. This challenge must be met by up-to-date design techniques. An adequate technique has to feature primarily the support of three problem domains: specification, verification, and synthesis. In current approaches, these problems are handled independently and there is barely any approach to embrace these problem domains within an uniform framework.

Hardware description languages like VHDL and Verilog are widely used for specification. These HDLs are indeed qualified for specification at the register-transfer level, but at the algorithmic level they are rather unsuitable and, due to the lack of clear semantics, verification of VHDL and Verilog at the behavioral level is impractical. Furthermore, these HDLs were designed as hardware description languages and therefore do not adequately address hardware/software codesign.

Recently languages like C++ and Java were employed to take advantage of their object oriented concepts. These concepts enable the designer to specify a system at a higher level of abstraction and to thereby increase productivity, readability, and reusability. However, all these languages suffer from insufficient concepts for verification and therefore do not allow coverage of all three problem domains mentioned above.

This shortcoming is solved by the 'e' language presented in this paper. 'e' is an object oriented language that was designed in the style of Java and enhanced by various constructs and concepts for verification. These concepts are supported by the dedicated run-time environment Specman™ [1]. Thus 'e' and Specman provide an efficient verification platform for hardware design. The 'e' language allows designers to make use of the abstract design techniques previously available only in software languages. Furthermore, it provides constructs for register-transfer level specification. Therefore we consider 'e' as a highly qualified language for specification.

The paper is organized as follows: In Section 2, we present an overview of some related work. In Section 3, we first introduce the design flow of the framework and give a short summary of specification and verification with 'e.' The remainder of that section presents our synthesis approach. Section 4 presents examples and discusses the results. Section 5 concludes the paper.

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 [2,3] enhances VHDL with object oriented concepts. SystemC [4] and CynLib [5] are based on C++ class libraries and are intended for design at the system/ algorithmic and register-transfer levels. Recently, successful efforts have been made to specify [6,7,8] and to synthesize [9] from Java.

Validation, simulation and formal verification have a long tradition and various commercial tools are in the market (e.g. Mentor Graphics, Cadence, Synopsys etc.). With the increasing complexity of today's hardware and hardware/software systems with millions of gates on a single chip, traditional simulation often reaches its limits due to the exploding number of test cases.

In contrast to simulation, formal verification [11,12] guarantees 100% test coverage. However, formal proofs often suffer from combinatorial explosion of the search space (e.g., model checking or Boolean equivalence checking). Formal verification was mostly academic for many years, but is now on the verge of being an

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

* This work was funded by DFG project No. Ro 1030/8-1 and CISCO.

integrated part of industrial design flows. In order to take advantage of both methods, several attempts to combine simulation with formal verification have been proposed [10].

3. THE FRAMEWORK

The design flow (figure 1) starts with the specification of the hardware module in 'e.' This specification can be either an **algorithmic** description or a description at the **register-transfer level**. The correctness of the specification can be verified by executing the 'e' code in the Specman environment. The results of this first step in the verification process can be used to refine the test bench, also written in 'e,' around the hardware module.

The 'e' specification of the hardware module is processed by our 'e' Synthesis System (eSS). The output of eSS is an intermediate format that contains no more object oriented constructs and is therefore ready to be further processed by standard EDA tools. One output format is synthesizable Verilog. 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™, or a Verilog description at the register-transfer level that can be synthesized by Design Compiler. Additionally, eSS generates Verilog code that comprises several enhancements for simulation. Other formats like VHDL or C/C++ are also generated by eSS and can be further processed by standard tools.

During all phases of the synthesis process the results can be verified by simulation in the Specman environment. An HDL simulator can be attached to Specman; while the same test bench as in step one of the verification process can be loaded into Specman, the generated **Verilog** code is now executed by the simulator. All steps of the verification process are supported by Specman's sophisticated verification methodologies, and the object oriented concepts of the 'e' language allow a high degree of reusability of the design and the test bench.

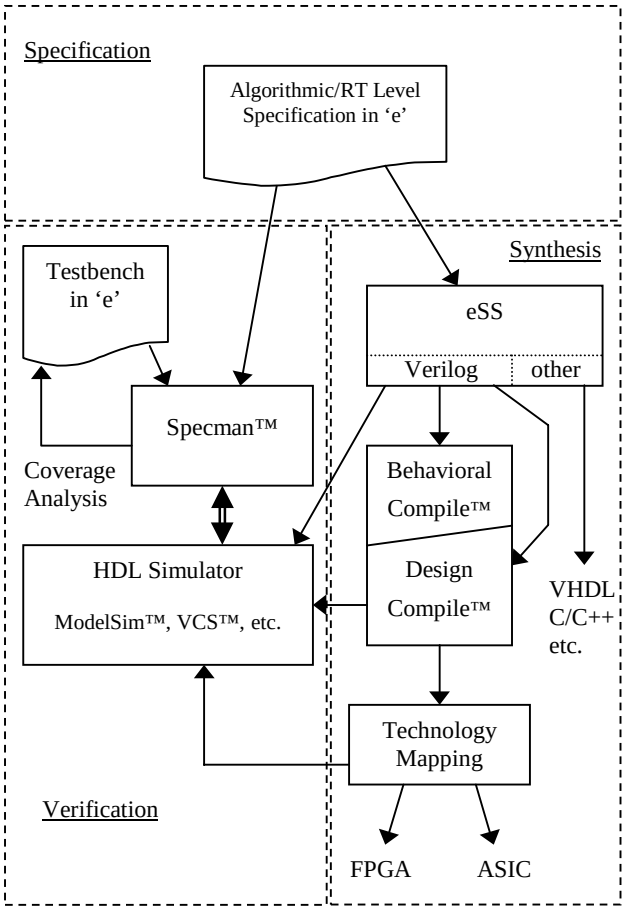

Figure 1: Design flow

3.1 Specification with 'e'

The main benefit of specifying with 'e' is that all common object oriented concepts like data abstraction and inheritance can be used to keep the specification understandable, manageable, and reusable while the syntax of 'e' allows the **imperative** description of hardware specific concepts like word lengths of variables, in/out-ports, clocks etc. as well as the **declarative** description of constraints and type extensions. We only give a short insight to syntax of 'e'. For a detailed description please refer to [1,15].

The main language construct are classes. Classes are expressed using the keyword *struct* and can be subclasses using *like*:

```
struct SubTest like Test {
    // data field declarations
    // method declarations
};
```

Instances of a struct are made by using the keyword *new*. Objects can be accessed by typed references.

```
t : Test; t = new Test; t.foo();
```

Scalar data types, bit slices etc. are also supported as well as the commom arithmetical, logical, and boolean operators:

```
x : uint(bits: 4); x = 5; x = 0b1010;
b : bool; b = FALSE;
```

Time Consuming Methods (TCM) are concurrent threads of control that execute over multiple simulated time units. They are used to specify concurrency as well as driving information for the Device Under Test (DUT). In contrast, non-TCMs are executed in one simulation unit. TCMs are invoked using the *start* command. An example for a struct 'S' with a TCM is shown below in figure 2, where the definition of the clock 'clk' indicates that the method 'tcm' is a TCM The client code in the lower box of figure 2 calls the TCM 'tcm' on an instance of the struct 'S.'

Events can be emitted upon change of state either in the DUT or the verification environment. Events are the basic synchronization primitives in 'e.' Events can be used for asynchronous or synchronous communication. They can define simulator clocks which drive TCMs. Only when the defined event occurs, the TCM is executed by the scheduler.

Wait statements are avialable to synchronize write operations on data field members which are used as out-ports or for inter-TCM communication.

The 'e' language also features constraints which allow precise data generation and system definition. The object oriented and constraint driven definition of a system provides a valuable resource for the verification of a system under test. In the code sequence below, the declarative keyword *keep* defines a constraint for the data fields 'a' and 'b.'

```
a : int; b : int; keep a + b < 32;
```

The sum of the generated data for the variables is always kept lower than 32. Two legal values for the variables 'a' and 'b' are '-256' and '+128.'

```
struct S {
    event clk is @ sys.HWO.clk;

    tcm() @ clk is { ... };
    nontcm() is { ... };
};

s : S;
s = new S;
start s.tcm();
wait[2];
```

Figure 2: struct with one TCM and one non-TCM

We have enhanced e with the following constructs for HW design:

The programmer of a hardware component uses the struct HWO (HardWare Object) that is predefined by the synthesis framework. The HWO struct is used as superclass for each hardware component of the system. An example is shown in figure 3.

```
struct Test like HWO {
    n : uint(bits: 4);
    init() is {  n = 0b1101;  };
    runHWO() @ clk is {
        n = n & 0b0100;
    };
};
```

Figure 3: Example for a hardware component

The hardware component has to implement two methods: an *init* method that is like a constructor and is used for the initialization of data field members of the struct, and a 'runHWO' method that is used to describe the functionality. Components communicate through ports. In/out-ports of HWOs are implemented as data field members of the HWO struct. Access to a port is realized by setter and getter methods for each port.

3.2 Verification with 'e' and Specman

Simulation based verification requires the introduction of stimuli to the DUT being simulated, as well as the collection of DUT responses for the purpose of checking and coverage analysis (figure 4). Specman and the 'e' language provide powerful support for these verification tasks. The supported verification methodology can be applied to DUTs implemented in 'e,' those implemented in HDLs such as Verilog and VHDL as well as numerous other modeling languages. Many SoC designs involve both hardware and software components, therefore the integrated hardware/software environment will be the target of verification.

The 'e' and Specman based methodology can effectively be applied to hardware/software co-verification [13], with the optional addition of a processor modeling system such as Mentor Graphics' Seamless™

3.2.1 Input Modeling and Generation

In 'e,' input stimuli are modeled as a hierarchy of objects with interrelating 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. They 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.

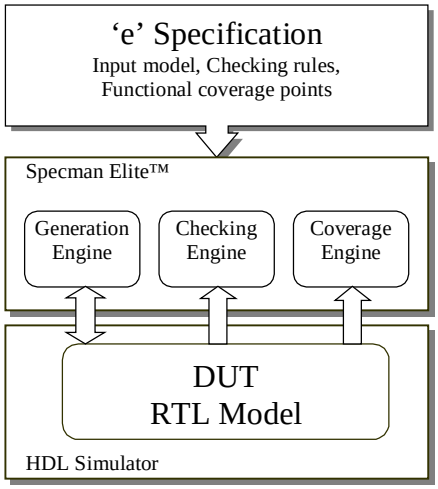

Figure 4: A typical 'e' driven test bench

3.2.2 Driving and Checking

A directed random input generation methodology requires automated checking, since the input model does not predict a unique response. The '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 to express protocol rules the DUT must adhere to. Specman features a temporal engine that interprets temporal expressions.

3.2.3 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 '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 information about the verification effort. This information is used to steer the process and to eventually certify that the DUT has a high probability of being functional.

3.2.4 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 (figure 5). 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 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.

3.3 Synthesis from 'e'

For synthesis, the object oriented system description in 'e' has to be transformed into an equivalent description in Verilog. The analysis steps and the resulting data structures are shown in figure 6. The last step of the transformation is the output of intermediate formats, which is based on the results of the control data flow analysis and the concurrency analysis.

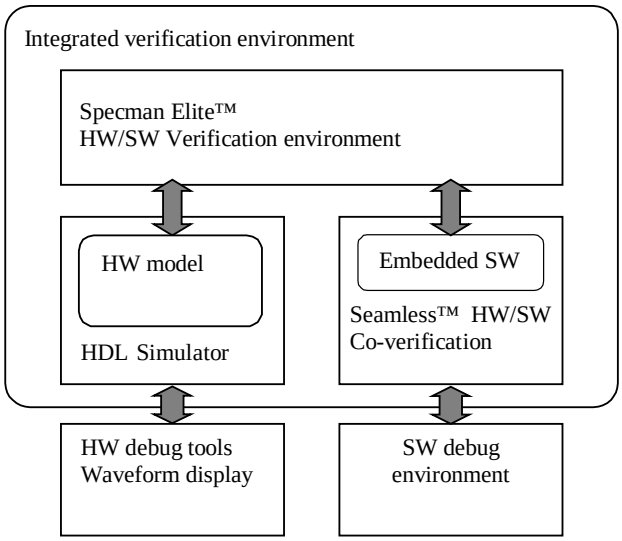

Figure 5: Hardware/software co-verification environment

3.3.1 Control Data Flow Analysis

After the lexical and semantical analysis done by the scanner and parser for 'e,' a static control and data flow analysis is performed on the set of syntax trees for each 'e' struct. 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 (each node has multiple children). For example nodes for *while* loops or *if* statements. The second type of nodes in the CFG are the data flow nodes, which don't 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.

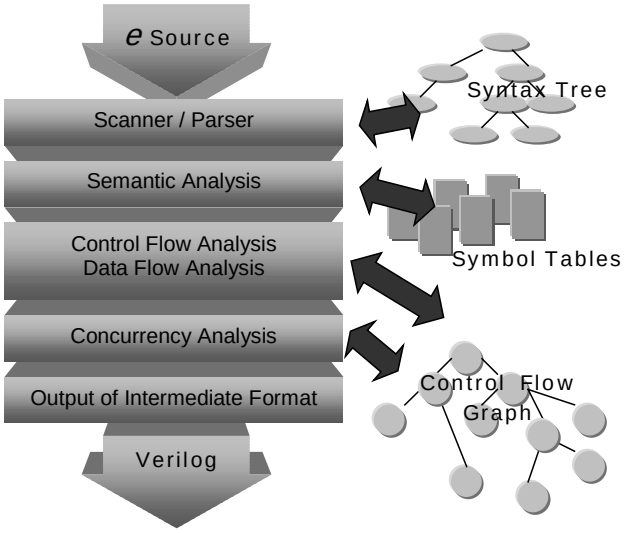

Figure 6: The transformation process from 'e' to Verilog

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 $S(n)$.

The analysis for the TCM 'runHWO' of the example 'e' code in figure 7 results in the CFG shown in figure 8. 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.

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). 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.

Table 1: Reference sets for the whole 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 the number of iteration within the while

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. This is guaranteed since the instantiation of objects is only allowed in within loops with fixed number of iterations.

```
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 7: HWO struct 'Test' and struct 'S'

3.3.2 Concurrency Analysis

During the concurrency analysis, the reference sets and the CFGs of the data and control flow analysis are used to create 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 [14].

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*.

After the analysis has determined the CFG, the actual synthesis can be carried out. For that purpose the CFG is tranformed into a general format that can be processed by all standard tools. Implemented general formats are Verilog, VHDL and SystemC.

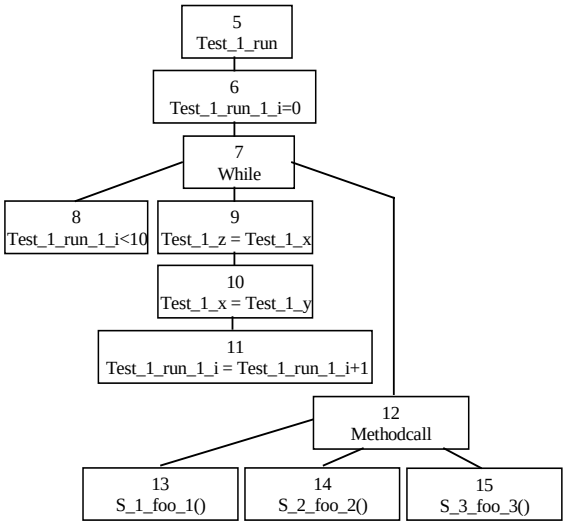

Figure 8: CFG for the TCM 'runHWO'

4. EXAMPLES AND RESULTS

We used our framework to specify, verify, and synthesize 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 and is presently successfully in production.

The results of the synthesis are depicted in table 2. 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 depth of the FIFO buffer. While the '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. For each module, the number of non-comment lines of code (ncloc) of both the 'e' specification and the resulting Verilog code, and the time needed by eSS for the translation from 'e' to Verilog are given. The execution time of eSS, as well as the time spent by Behavioral Compiler and Design Compiler for the synthesis (BC/DC), were measured on a 360 MHz 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.

Table 2. Results of the synthesized examples

Module	ncloc 'e'	ncloc Verilog	ESS	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

5. CONCLUSIONS AND FUTURE WORK

This paper presented results from the DFG project OASE and a cooperation between Cisco Systems, Verisity, and the Computer Engineering Department from the University of Tübingen. The developed framework allows the designer to specify, verify, and synthesize hardware within a uniform environment. All these tasks can be carried out by the designer by using the object oriented language 'e.' This contribution shows that synthesis from high-level models written in 'e' is practical and that the presented framework is capable of dealing with real-world applications. So far we have deployed our framework and the 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.

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

6. REFERENCES

[1] http://www.verisity.com

[2] S. Swamy, A. Molin, and B. Covnot. OO-VHDL: Object-Oriented Extensions to VHDL. *IEEE Computer*, 1995.

[3] 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.

[4] http://www.systemc.org

[5] http://www.cynlib.com

[6] R. Helaihel and K. Olukotun. Java as a Specification Language for Hardware-Software Systems. In *Proceedings of ICCAD*, 1997.

[7] 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 DAC*, 1998.

[8] T. Kuhn, W. Rosenstiel, and U. Kebschull. Description and Simulation of Hardware/Software Systems with Java. In *Proceedings of DAC*, 1999.

[9] T. Kuhn and W. Rosenstiel. Java Based Object Oriented Hardware Specification and Synthesis. In *Proceedings of ASP-DAC*, 2000.

[10] J. Ruf, D. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation Based Validation of FLTL Formulas in Executable System Descriptions. In *Proceedings of the Third Forum On Design Languages*, Tuebingen, 2000.

[11] W.C. Carter, W.H. Joyner Jr., and D. Brand. Symbolic simulation for correct machine design. In *16th ACM/IEEE Design Automation Conference (DAC)*, 1979.

[12] J.J. Joyce and C.-J.H. Seger. Linking BDD based symbolic evaluation to interactive theorem proving. In *Proceedings of DAC*, 1993.

[13] G. Mosenson. Practical Approaches to SOC Verification. In *Proceedings of DATE User Forum*, 2000.

[14] J. Madsen and J.P. Brage. Modeling Shared Variables in VHDL. In *Transactions on the ACM*, 1994.

[15] Y. Hollander, M. Morley, and A. Noy. The E Language: A Fresh Separation Of Concerns. *Proceedings of TOOLS-38*, 2001