# ParSyC: An Efficient SystemC Parser[*]

Görschwin Fey    Daniel Große    Tim Cassens    Christian Genz    Tim Warode    Rolf Drechsler

Institute of Computer Science, University of Bremen
28359 Bremen, Germany
{fey,drechsle}@informatik.uni-bremen.de

**Abstract— Due to the ever increasing complexity of circuits and systems new methodologies for system design are mandatory. Languages that enable modeling at higher levels of abstraction but also allow for a concise hardware description offer a promising way into this direction. One such language is SystemC.**

**In this paper we propose the SystemC parser *ParSyC*, that allows to convert given SystemC source code into an internal intermediate representation. By this a formal model is retrieved from a SystemC description. *ParSyC* can serve as a generic front end for different purposes as e.g. verification or visualization. To demonstrate the efficiency *ParSyC* is applied to synthesis of Register Transfer Level (RTL) descriptions of different sizes and different types of designs.**

## I. Introduction

New design methodologies and design flows are needed to cope with the increasing complexity of today's circuits and systems. This applies to all stages of circuit design from system level modeling and verification down to layout.

One focus of research in this area is the use of new hardware description languages. Traditionally the system level description is done in a programming language like C or C++, while dedicated hardware description languages like VHDL and Verilog are used on the RTL. This leads to a decoupling of the behavioral description and the synthesizable description. Recently developed languages allow for higher degrees of abstraction, but additionally the refinement for synthesis is possible within the language. One of these new languages is *SystemC* [8, 6]. Basis of SystemC is C++. Therefore all features of C++ are available. The additional SystemC library provides all concepts that are needed to model hardware, as e.g. timing or concurrency. By this, hardware modeling with SystemC can be easily done.

Research on the conceptual side and on algorithms for SystemC is more difficult. For areas like high-level synthesis, verification or power-estimation a formal understanding of the given design is necessary before any subsequent processing can be carried out. Most recent publications either focused on special features of SystemC or C/C++, like synthesis of fixed point numeric operations [2], polymorphism [14] or pointers [7], or considered the design methodology [10]. Few works (e.g. [11]) have been published that rely on the formal model of an arbitrary SystemC design. One reason for this is the high effort to syntactically and semantically *understand* the SystemC description. For this purpose SystemC has to be parsed.

In this paper a Parser for SystemC implemented as the tool *ParSyC* is presented. The parser covers SystemC and to a certain extent C++. The *Purdue Compiler Construction Tool Set* (PCCTS) [12] was used to build *ParSyC*. This parser produces an easy-to-process representation of a SystemC design in form of an *intermediate representation*. The description is generic, i.e. any further processing can start from this representation, regardless of the application to visualization [5], formal verification [4] or other purposes. As an example the application to synthesis of RTL descriptions is explained and the efficiency is underlined by experiments. Some of the advantages yielded by this approach are easy extendability, adaptivity and efficiency of the SystemC front-end.

The paper is structured as follows: The basic concepts of SystemC are discussed in the following section. The methodology to create *ParSyC* and the exemplary application to synthesis are explained in Section III. Advantages of the approach are discussed in Section IV. The experimental results are given in Section V. Conclusions are presented in Section VI.

## II. SystemC

The main concepts of SystemC are briefly reviewed in the following to make the paper self-contained. SystemC is a system description language that enables modeling at different levels of abstraction. Constructs known from traditional hardware description languages are also provided. By this, any task between design exploration at the high-level and synthesis at the low-level can be carried out within the same environment. Features to aid modeling at different levels of abstraction are included in SystemC. For example the concept of channels allows to abstract from details of communication between modules. Therefore modeling at the transactional level can be done. This in turn enables fast design space exploration and partitioning of the design before working on the details of protocols or modules.

[*]This work was supported in part by DFG grant DR 287/8-1.

In practice SystemC comes as a library that provides classes to model hardware in C++. For example a module in hardware is modeled using the class `sc_module` provided by SystemC. All features of C++ are also available in SystemC. This includes dynamic memory allocation, multiple inheritance as well as any type of complex operations on data of an arbitrary type. Any SystemC-design can be simulated by compiling it with an ordinary C++-compiler into an executable specification. But to focus on other aspects of circuit design a formal model of the design is needed.

Deriving a formal model from a SystemC description is hard: A parser that handles SystemC – and for this C++ – is necessary. But developing a parser for a complex language comes at a high effort. Moreover the parser should be generic in order to aid not only a single purpose but to be applicable for different areas as well, e.g. synthesis, formal verification and visualization.

### *A. Synthesis*

In order to allow for concise modeling of hardware several constructs are excluded from the synthesizable subset of SystemC [16]. For example SystemC provides classes to easily model buffers for arbitrary data using the class `sc_fifo`. An instance of type `sc_fifo` can have an arbitrary size and can work without explicit timing. Therefore there is no general way for synthesis. In principle this could be solved by providing a standard realization of the class. But in order to retrieve a good - e.g. small and/or fast - solution after synthesis, several decisions are necessary. Therefore it is left to the hardware designer to replace this class by a synthesizable description. Also the concept of dynamic memory allocation is hardly synthesizable in an efficient way and therefore excluded from the synthesizable subset.

For a better understanding synthesis of RTL descriptions is used to demonstrate the features of *ParSyC*. Due to this application the SystemC input is restricted, but as a generic front-end *ParSyC* can handle other types of SystemC descriptions as well.

## III. SYSTEMC PARSER

In this section the methodology to build a parser and the special features used for parsing SystemC are explained. The synthesis of RTL descriptions is carried out using *ParSyC* as a front-end.

The methodology for parsing and compiling has been studied intensively (see e.g. [1]). Often the Unix-tools *lex* and *yacc* are used to create parsers. But more recent tools provide easier and more powerful interfaces for this purpose. Therefore the tool PCCTS was used to create *ParSyC*. For details on the advantages of PCCTS see [13, 12]. Specialized for SystemC the parser was built as follows:

```
sc_in<sc_bv<8> >  uSEQ_BUS;
sc_out<bool>      LSB_CNTR;
sc_uint<8>        counter;
sc_signal<bool>   DONE, LDDIST, COUNT;
```

(a) Datatypes

```
(0) void robot_controller::counter_proc()
(1) {
(2)   if (LDDIST.read()) {
(3)     counter = uSEQ_BUS.read();
(4)   } else if (COUNT.read()) {
(5)     counter = counter - 1;
(6)   }
(7)   DONE.write(counter == 0);
(8)   LSB_CNTR.write(counter[0]);
(9) }
```

(b) Process

Fig. 1. The process `counter_proc` of the robot controller from [6]

- A preprocessor is used to account for directives and to filter out header-files that are not part of the design, like system-header-files.
- A lexical analyzer splits the input into a sequence of *tokens*. These are given as regular expressions that define keywords, identifiers etc. of SystemC descriptions. Besides C++ keywords also essential keywords of SystemC are added, e.g. `SC_MODULE` or `sc_int`.
- A syntactitcal analyzer checks if the sequence of tokens conforms to the *grammar* that describes the syntax of SystemC. Terminals in this grammar are the tokens.

PCCTS creates the lexical and syntactical analyzer from tokens and grammar, respectively. Together they are referred to as the parser. The result of parsing a SystemC description is an *Abstract Syntax Tree* (AST). At this stage no semantic checks have been performed as e.g. for type conflicts. The AST is constructed using a single node type, that can have a pointer to the list of children and a pointer to one sibling. Additional tags at each node are used to store the type of a statement, the string for an identifier and other necessary information. This structure is explained by the following example.

**Example 1** *Consider the code fragment in Figure 1. This is one process of the robot controller introduced in [6]. Figure 2 shows a part of the AST for this process. Missing parts of the AST are indicated by triangles. In the AST produced by PCCTS each node points to the next sibling and to the list of children. The node in the upper left represents the* `if`*-statement from line (2) of the code. The condition is stored as a child of this node. The* `then`*-part and the* `else`*-part of the statement are siblings of the child.*

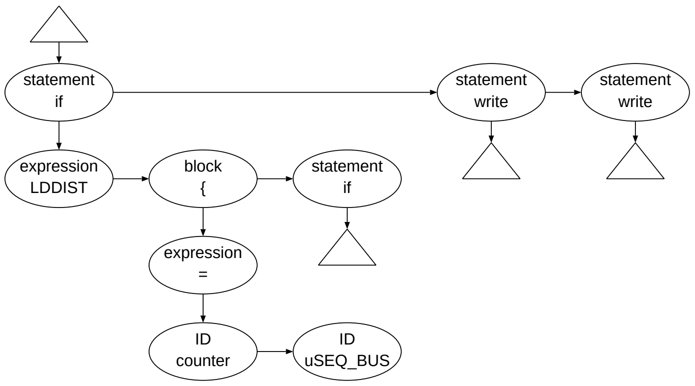

Fig. 2. AST for Example 1

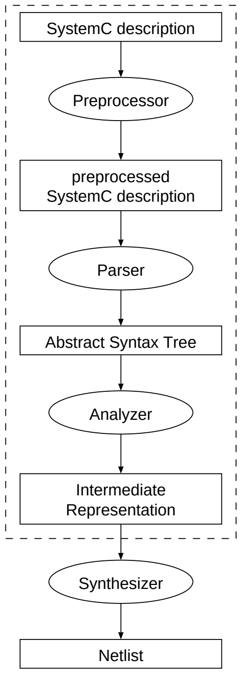

Fig. 3. The overall synthesis procedure

The overall procedure when applying the parser for synthesis is shown in Figure 3. The dashed box indicates steps that are application independent, i.e. the corresponding tasks have to be executed for other applications as visualization or formal verification as well. The whole process can be divided in several steps.

- After preprocessing the parser is used to build the AST from the SystemC description of a design.

- The AST is traversed to build an *intermediate representation* of the design. All nodes in an AST have the same type, any additional information is contained in tags attached to theses nodes. Therefore different cases have to be handled at each node while traversing the AST. By transforming the AST into the intermediate representation the information is made explicit in the new representation for further processing. The intermediate representation is built using classes to represent building blocks of the design, like e.g. modules, statements or blocks of statements. During this traversal semantic consistency checks are carried out. This includes checking for correct typing of operands, consistency of declarations and definitions, etc. (Up to this stage the parser is not restricted to synthesis and all processing is application-independent.)

- The intermediate representation serves as the starting point for the originally intended application. At this point handling the design is much easier, because it is represented as a formal model within the class structure of the intermediate representation. The classes used to assemble the intermediate representation correspond to constructs in the SystemC-code. Each component is “self-aware”, i.e. it knows about its own semantic in the original description. Further processing of the design is done by adding application-specific features to the classes used for storing the intermediate representation. In case of synthesis a recursive traversal is necessary. Each class is extended by functions for the synthesis of substructures to generate a gate-level description of the design.

**Example 2** *Again consider the AST shown in Figure 2. This is transformed into the intermediate representation shown in Figure 4. The structure looks similar to that of the AST, but in the AST only one type of node was used. Now dedicated classes hold different types of constructs. The differentiation of these classes relies on inheritance in C++. Therefore synthesis can recursively descend through the intermediate representation.*

As usual in synthesis, RTL descriptions in SystemC are restricted to a subset of possible C++ and SystemC constructs [16]. C++ features like dynamic memory allocation, pointers, recursions or loops with variable bounds are not allowed to prevent difficulties already known from high-level-synthesis. In the

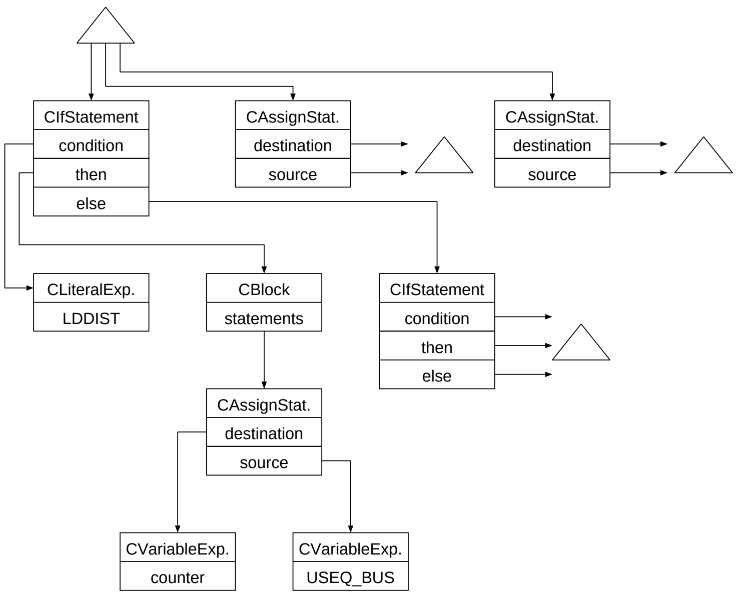

Fig. 4. Intermediate representation

same way some SystemC constructs are excluded from systhesis as they have no direct correspondence on the RTL, e.g. as shown for `sc_fifo` in Section II.A. Thus, for simplicity SystemC channels were excluded from synthesis. For channels that obey certain restrictions synthesis can be extended by providing a library of RTL realizations.

Supported are all other constructs that are known from traditional hardware description languages. This comprises different operators for SystemC-datatypes, hierarchical modeling or concurrent processes in a module. Additionally, the `new`-operator is allowed for instatiation of submodules to allow for a compact description of scalable designs.

Outcome of the synthesis process is a gate-level description in BLIF-format as used by SIS [15]. Switching the output format to VHDL or Verilog on the RTL is easily possible. Focus of this work is the parsing of SystemC and retrieving a formal model from the description, therefore no optimizations are applied during synthesis.

## IV. Discussion

The presented approach to create a formal model from a SystemC description has several advantages:

- *Extendability.* SystemC is still evolving. The parser can easily be extended to cope with future developments by changing the underlying grammar and extending the classes for the intermediate representation. The necessary changes are straightforward in most cases.
- *Adaptivity.* Here, *ParSyC* is only exemplary applied to synthesis, but several other applications are also of interest. When starting with a new application that should work on SystemC-designs the intermediate representation directly serves as a first model of the design. Only application specific extensions are necessary to allow for further processing.
- *Decoupling.* The complex process of parsing should be hidden from the application. The use of *ParSyC* as the front-end to “understand” a given SystemC description allows the intended application to concentrate on algorithms and efficiency for the intended purpose.
- *Efficiency.* A fast front-end is necessary to cope with large designs. The efficiency of the front-end is guaranteed by the compiler-generator PCCTS. The subsequent application can directly start processing the intermediate representation that is given as a C++-class structure. Experiments are presented in the next section to underline the efficiency of *ParSyC*.
- *Compactness.* The parser should be compact to allow for an easy understanding during later use and extension. The parser itself has only ≈1000 lines of code (loc) which includes the grammar and necessary modifications beyond PCCTS to create the AST. The code for analyzing and the classes for the intermediate representation consists of ≈4000 loc. For synthesis ≈2500 loc are needed. The complete tool for synthesis including error handling, messaging etc. has ≈9000 loc. Comments and blank lines in the source are not included in these numbers.

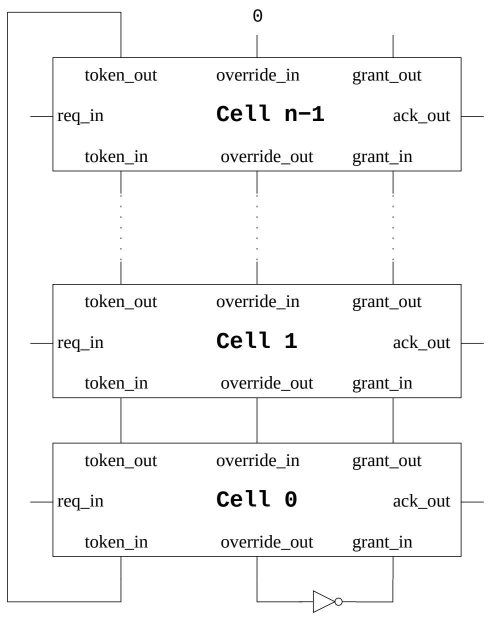

Fig. 5. Arbiter: Block-level diagram

## V. EXPERIMENTS

All experiments have been carried out on a Pentium IV with Hyperthreading at 3GHz and 1GB RAM running Linux. *ParSyC* has been implemented using C++.

A control dominated design and a data dominated design are considered in the first two experiments, respectively. Large SystemC-descriptions are created from ISCAS89-circuits to demonstrate the efficiency of *ParSyC* in the third experiment.

### *A. Control Dominated Design*

The scalable arbiter introduced in [9] has been frequently used in works related to property checking. Therefore a SystemC-description on the RTL was created and synthesized. The top level view of the arbiter is shown in Figure 5. This design handles the access of `NUMC` devices to a shared resource. Device `i` can signal a request on line `req_in[i]` and may access the resource, if the arbiter sets `ack_out[i]`. The arbiter uses priority scheduling, but also guarantees that no device waits forever (for details we refer to [9]). Figure 6 shows the SystemC-description of the top-level module `scalable`. For each of the `n` devices a corresponding arbiter cell is instantiated and the cells are interconnected using a `for`-loop. Results for the synthesis with different numbers of arbiter cells are shown in Table I. Given are the size of the netlist output and the CPU times needed. The netlist is built from 2-input gates. The number of gates that is contained in the flattened netlist, while the hierarchical description generated by the synthesis tool always contains 190 gates: 188 gates per arbiter cell plus one additional buffer and one inverter. The same holds for the number of latches. The flattened netlist contains two latches per arbiter cell while the hierarchical netlist only contains two latches in total. Note, that the arbiter cells are described at RTL and synthesis is carried out

```
#include "RTLCell.h"
#include "Inverter.cc"
#define NUMC 2

SC_MODULE(scalable) {
  // Declaration of inputs, outputs
  // and internal signals is omitted
  // due to lack of space

  Inverter *inv;
  RTLCell *cells[NUMC];
  SC_CTOR(scalable) {
    for (int i = 0; i < NUMC; ++i) {
      // Create cell i
      cells[i]= new RTLCell("cells");
      if (i==0) {
        // Connect cell 0
        cells[i]->TICK(clk);
        ...
        cells[i]->ove_out(override_out);
      } else {
        if (i==(NUMC-1)) {
          // Connect cell NUMC-1
          ...
        } else {
          // Connect cell i
          ...
        }
      }
    }
    inv= new Inverter("Inverter");
    inv->in( override_out );
    inv->out( grant_in );
  }
};
```

Fig. 6. Arbiter: Top-level module `scalable`

without applying optimizations. Therefore the gate level representation of the cells can not be optimal.

The times needed for parsing $t_p$, analyzing $t_a$, synthesis $t_s$ and the total time $t_t$ are shown in the respective columns. As can be seen scaling the arbiter does not influence the time for parsing because only the constant `NUMC` in the source code is changed. The time for analyzing increases moderately since type checks for the different cells have to be carried out. During synthesis the `for`-loop has to be unrolled and therefore scaling influences the synthesis time. The total time is dominated by the time needed for synthesis and includes overhead like reading the template for the output format, parsing the command line etc.

Even synthesizing a design that corresponds to a flattend netlist with 200k gates takes less than one CPU second.

TABLE I
ARBITER: SYNTHESIS RESULTS

| NUMC | in | out | latches | gates | $t_p$ | $t_a$ | $t_s$ | $t_t$ |
|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 5 | 10 | 942 | 0.01 | <0.01 | 0.01 | 0.04 |
| 10 | 11 | 10 | 20 | 1882 | <0.01 | 0.01 | <0.01 | 0.01 |
| 50 | 51 | 50 | 100 | 9402 | <0.01 | <0.01 | 0.02 | 0.04 |
| 100 | 101 | 100 | 200 | 18802 | <0.01 | 0.01 | 0.02 | 0.04 |
| 500 | 501 | 500 | 1000 | 94002 | <0.01 | 0.03 | 0.08 | 0.11 |
| 1000 | 1001 | 1000 | 2000 | 188002 | <0.01 | 0.06 | 0.16 | 0.24 |

## B. Data Dominated Design

The second design is a FIR-filter of scalable width. Scalable are the number of coefficients and the bit-width of data. A block-level diagram of the FIR-filter is shown in Figure 7. Incoming data is stored in a shift register (`d[0],...,d[n-1]`), coefficients (`c[0],...,c[n-1]`) are stored in a register array. The result is provided at the output `dout`. The SystemC description contains one process to create the shift-register and another process that carries out the calculations. The coefficients are provided by an array of constants. Synthesis results for different bit-widths and numbers of coefficients are given in Table II. In case of the arbiter additional checks for submodules were necessary during analysis of the `for`-loop. This is not the case for the FIR-filter, where no submodules are created, therefore scaling does not influence the time needed for analysis. But for the FIR-filter the time for synthesis increases faster compared to the arbiter when the design is expanded. This is due to the description of the multiplication as a simple equation in SystemC:

```
for (int i=0; i < n; i++) {
   tmp= c[i] * d[i].read();
   out= out+tmp;
}
```

Instead of instantiating modules, the operations are directly generated into the netlist. Therefore bit-width and number of coefficients have a direct influence on the synthesis time and the size of the output.

Again a large design of 500k gates has been parsed and analyzed very fast. The synthesis, which inlcudes writing the output to the hard disk, only took about 5 seconds.

## C. Large SystemC Descriptions

The first two experiments showed the influence of scaling different types of designs. In the following the influence of a large SystemC description is investigated. Circuits from the ISCAS89 benchmark set are considered. Starting from the netlist *Binary Decision Diagrams* (BDD) [3] were built for each circuit. While building the BDD no reordering techniques were applied for size reductions. For each output and next state the BDD was dumped

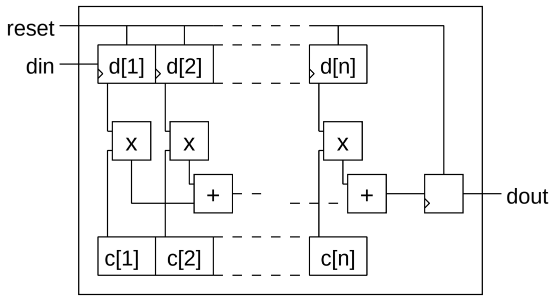

Fig. 7. FIR-filter: Block-level diagram

TABLE II
FIR-FILTER: SYNTHESIS RESULTS

| width | coeff | in | out | latches | gates | $t_p$ | $t_a$ | $t_s$ | $t_t$ |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 3 | 4 | 8 | 159 | <0.01 | <0.01 | <0.01 | <0.01 |
| 2 | 4 | 3 | 4 | 12 | 301 | <0.01 | <0.01 | <0.01 | <0.01 |
| 2 | 8 | 3 | 4 | 20 | 585 | <0.01 | <0.01 | 0.02 | 0.03 |
| 4 | 2 | 5 | 8 | 16 | 611 | <0.01 | <0.01 | 0.01 | 0.01 |
| 4 | 4 | 5 | 8 | 24 | 1189 | <0.01 | <0.01 | 0.02 | 0.04 |
| 4 | 8 | 5 | 8 | 40 | 2345 | <0.01 | <0.01 | 0.03 | 0.03 |
| 8 | 2 | 9 | 16 | 32 | 2283 | <0.01 | <0.01 | 0.03 | 0.04 |
| 8 | 4 | 9 | 16 | 48 | 4501 | <0.01 | <0.01 | 0.05 | 0.05 |
| 8 | 8 | 9 | 16 | 80 | 8937 | <0.01 | <0.01 | 0.10 | 0.12 |
| 16 | 2 | 17 | 32 | 64 | 8699 | 0.01 | <0.01 | 0.07 | 0.08 |
| 16 | 4 | 17 | 32 | 96 | 17269 | <0.01 | <0.01 | 0.14 | 0.15 |
| 16 | 8 | 17 | 32 | 160 | 34409 | <0.01 | <0.01 | 0.30 | 0.31 |
| 32 | 2 | 33 | 64 | 128 | 33819 | <0.01 | <0.01 | 0.29 | 0.29 |
| 32 | 4 | 33 | 64 | 192 | 67381 | <0.01 | <0.01 | 0.63 | 0.64 |
| 32 | 8 | 33 | 64 | 320 | 134505 | <0.01 | 0.01 | 1.22 | 1.23 |
| 64 | 2 | 65 | 128 | 256 | 133211 | <0.01 | <0.01 | 1.23 | 1.23 |
| 64 | 4 | 65 | 128 | 384 | 265909 | <0.01 | <0.01 | 2.34 | 2.35 |
| 64 | 8 | 65 | 128 | 640 | 531305 | <0.01 | <0.01 | 5.34 | 5.34 |

into an `if-then-else`-structure, which was embedded in a SystemC-module. This module was synthesized. The results are shown in Table III. Given are the name of the circuit, the lines of code *#loc* and the number of characters *#char* in the SystemC code. The circuits are ordered by increasing *#loc*. As can be seen the time for parsing increases with the size of the source code, but is small even for large designs of several 100000 *#loc*. The time needed for analysis increases faster due to the semantical checks and the translation into the intermediate representation that are carried out at this stage. The largest amount of time is due to synthesis were the intermediate structure is traversed and the netlist is written.

As a reference the time needed to compile the SystemC code using *g++* (version 3.3.2, optimizations turned off, no linking is done) is given in column $t_{g++}$. Compiling the SystemC description using *g++* means to create an executable description of the design for simulation. Therefore this is comparable to synthesis which creates the hardware description of the design. The total

TABLE III
ISCAS 89: SYNTHESIS RESULTS

| circuit | #loc | #char | $t_p$ | $t_a$ | $t_s$ | $t_t$ | $t_{g++}$ |
|---|---|---|---|---|---|---|---|
| s27 | 184 | 3129 | <0.01 | <0.01 | 0.01 | 0.02 | 2.26 |
| s298 | 1269 | 20798 | 0.01 | 0.02 | 0.07 | 0.12 | 2.26 |
| s382 | 2704 | 47343 | 0.02 | 0.05 | 0.16 | 0.26 | 2.44 |
| s400 | 2704 | 47343 | 0.03 | 0.04 | 0.16 | 0.26 | 2.41 |
| s386 | 4260 | 69331 | 0.04 | 0.07 | 0.24 | 0.39 | 2.57 |
| s526 | 3332 | 52999 | 0.03 | 0.05 | 0.19 | 0.31 | 2.56 |
| s344 | 5103 | 86055 | 0.04 | 0.06 | 0.29 | 0.45 | 2.70 |
| s349 | 5103 | 86055 | 0.05 | 0.09 | 0.29 | 0.49 | 2.73 |
| s444 | 6264 | 97100 | 0.06 | 0.11 | 0.39 | 0.63 | 2.78 |
| s641 | 54849 | 847546 | 0.48 | 1.27 | 4.16 | 6.47 | 8.27 |
| s713 | 54849 | 847546 | 0.50 | 1.29 | 4.24 | 6.58 | 8.52 |
| s1488 | 60605 | 981692 | 0.57 | 1.15 | 3.61 | 5.95 | 8.81 |
| s1494 | 60605 | 981692 | 0.55 | 1.17 | 3.61 | 5.96 | 8.84 |
| s1196 | 247884 | 3817191 | 2.27 | 5.57 | 16.53 | 26.82 | 30.88 |
| s1238 | 247884 | 3817191 | 2.33 | 5.62 | 16.58 | 27.01 | 31.28 |
| s820 | 402546 | 6130213 | 3.80 | 10.53 | 25.36 | 43.77 | 42.68 |
| s832 | 402546 | 6130213 | 3.75 | 10.57 | 25.69 | 43.99 | 43.12 |

runtime needed for synthesis is comparable to the time needed by *g++*, even for the largest files.

The experiments have shown, that *ParSyC* is an efficient front-end for SystemC. For this purpose designs have been considered that are large in terms of the number of gates and in terms of the size of the SystemC-description. The performance of *ParSyC* is comparable to an efficient and widely used compiler as g++.

## VI. CONCLUSIONS

*ParSyC* has been introduced as a front-end to construct a formal model from a SystemC description. The formal model is given by an intermediate representation that can serve as a starting point for different applications in the design flow, like verification, visualization and others. This allows to hide the complexity of parsing a SystemC description from the intended application. As an example the synthesis of RTL descriptions has been shown. Several experiments underline the efficiency of the tool.

## ACKNOWLEDGMENT

We would like to thank *FunTaskIC*, a project run by students of the University of Bremen. *FunTaskIC* was where the main implementation work has been done and where a lot of important discussions took place.

We would also like to thank Terrence J. Parr and the PCCTS development team for providing the Purdue Compiler Construction Tool Set as public domain software.

## REFERENCES

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers - Principles, Techniques and Tools*. Pearson Higher Education, 1985.

[2] A.G. Braun, J.B. Freuer, J. Gerlach, and W. Rosenstiel. Automated conversion of SystemC fixed-point data types for hardware synthesis. In *VLSI of System-on-Chip*, pages 55–60, 2003.

[3] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[4] D. Große and R. Drechsler. Formal verification of LTL formulas for SystemC designs. In *IEEE International Symposium on Circuits and Systems*, pages V:245–V:248, 2003.

[5] D. Große, R. Drechsler, L. Linhard, and G. Angst. Efficient automatic visualization of SystemC designs. In *Forum on Specification and Design Languages*, pages 646–657, 2003.

[6] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[7] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conf.*, pages 368–371, 2003.

[8] S. Liao, S. Tjiang, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the scenic design environment. In *Design Automation Conf.*, pages 70–75, 1997.

[9] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[10] W. Müller, W. Rosenstiel, and J. Ruf, editors. *SystemC Methodologies and Applications*. Kluwer Academic Publishers, 2003.

[11] W. Müller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of SystemC. In *Design, Automation and Test in Europe*, pages 64–70, 2001.

[12] T. Parr. *Language Translation using PCCTS and C++: A Reference Guide*. Automata Publishing Co., 1997.

[13] T.J. Parr and R.W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software – Practice and Experience*, 25(7):789–810, 1995.

[14] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosenstiel. Object-oriented modeling and synthesis of SystemC specifications. In *ASP Design Automation Conf.*, pages 238–243, 2004.

[15] E. Sentovich, K. Singh, L. Lavagno, Ch. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, University of Berkeley, 1992.

[16] Synopsys. *Describing Synthesizable RTL in SystemC$^{TM}$, Vers. 1.1*. Synopsys Inc., 2002. Available at http://www.synopsys.com.