PARSING SYSTEMC:
AN OPEN-SOURCE, EASY TO EXTEND PARSER

C. Brandolese	P. Di Felice	L. Pomante	D. Scarpazza
Politecnico di Milano	Università di L'Aquila	Università di L'Aquila	Politecnico di Milano
DEI	DIEI	DIEI	DEI
brandolese@elet.polimi.it	difelice@ing.univaq.it	pomante@ing.univaq.it	scarpazza@elet.polimi.it

1. ABSTRACT

SystemC is a de-facto standard for Register Transfer Language (RTL), behavioral and system-level modeling, but the lack of open-source parsers has represented, for a long time, a strong obstacle to the development of a large family of analysis, synthesis, estimation, and optimization tools. Only very recently there has been a variety of implementations of SystemC parsers. After a brief summary of previous solutions, the paper outlines our own contribute to fill the gap. The major merit of the new solution is the straightforwardness to be repeated and customized, when necessary.

2. KEYWORDS

SystemC, Parser, Open-source.

3. INTRODUCTION

SystemC is a *de facto* standard for modeling and designing at the register-transfer, behavioral and system levels. It is supported by the *OSCI* (*Open SystemC Initiative* 9) consortium, composed of more than 50 members among semiconductor companies, IP providers, embedded software developers, system houses, tool vendors and independent developers. Its unique features address many current and forecasted needs of the design community 9, by establishing common libraries, models and tools, which provide:

- a single description language for all the design elements enabling true hardware-software co-design, co-simulation and co-synthesis;
- an easy incorporation of software and hardware IP blocks;
- simulation capabilities at abstraction levels above the RTL, which is required for complex hardware designs.

Despite this promising scenario, for several years there have been no open and complete synthesis flows: the most known ones (e.g. *Synopsys SystemC Compiler* 9 and *Forte Design Systems Cynthesizer* 9, *Cadence NC-SystemC* 9) are not freely available and not even the front-ends (the parsers) of these flows are free. This lack has represented a severe obstacle to the development of a large spectrum of design methodologies and tools oriented to validation and application of source code metrics, analysis of source code quality, programmer's productivity, source-level optimization, automatic partitioning and design space exploration, etc.

Only very recently there has been a variety of implementations of SystemC parsers (9999). This paper too aims at giving a contribution to fill the gap. We focus on a significant subset of *SystemC 1.0* (the last version that is completely supported by a reference implementation and by appropriate synthesis tools), such as, the *Behavioral Synthesizable SystemC 1.0* 9 (*BSSC* for short) accepted by the *Synopsys SystemC Behavioral Compiler.*

Aim of this paper is the description of our own parser's development strategy. The material is organized as follows: Section 2 reports about the related work; Section 3 motivates and describes the strategy; Section 4 illustrates the design choices; Section 5 sketches an example of usage of the parser and the results of its application to a sample SystemC project; Section 6 ends the paper.

4. RELATED WORK

Table 1 collects all known implementations of open-source parsers of SystemC (to the best of our knowledge), ours included, and provides a short comparison among them according to eight basic features (namely: grammar, abstraction level, etc.).

Table 1. Comparison between open-source SystemC parsers

Feature/Tool	**PINAPA**	**PARSYC**	**KaSCPar**	**SystemPerl**	**OurTool**
Grammar	C++	Extended C++	Extended C++	Ad-hoc	Extended C
Abstraction Level	Full TLM	RTL	Partial TLM	N/A	Behavioural
Internal representation	Structural&AST (with limitations)	Structural&AST	Structural&AST (with limitations)	Only structural	Structural&AST
Parsing approach	Static and dynamic	Static	Static	Static	Static
Target applications	- Analisys - Model checking - Formal verification - Visualization	- Analisys - RTL synthesis (limited) - Visualization	- Analysis - Visualization	- Analisys (limited) - Visualization (limited)	- Analisys - Co-synthesis - Visualization
Development effort	HIGH/MEDIUM: - gcc dependent	MEDIUM: - Complex grammar - PCCS-based	HIGH/MEDIUM: - Complex grammar	LOW: - Simple info - Simple grammar	MEDIUM/LOW: - Simple grammar - Simple semantic actions
Customization effort	HIGH/MEDIUM: - Complex grammar - gcc dependent	MEDIUM: - Complex grammar - PCCS-based	HIGH/MEDIUM: - Complex grammar - gcc dependent	HIGH: - Intrinsic limitations	LOW: - Simple grammar - Simple semantic actions
Output format	Legacy	Legacy	XML	Legacy	Relational data

Below, a short description of each of the already known four different solutions is given.

PINAPA 9 is a tool based on the pure C++ grammar, so it is able to manage every kind of SystemC project, a feature especially suitable for developing analysis and verification tools working at high levels of abstraction (Transaction Level Modeling, TLM). It provides structural and syntactical information (*Abstract Syntax Tree*, AST) about the project, but such information are limited by the complex procedure necessary to extract them; in fact, other than a static analysis of the code, it has to be executed too in order to obtain the structural information lost during the translation into pure C++. To develop and customize PINAPA seems to be complex because it requires knowledge about gcc data structures and the ability to manipulate them. The output format is legacy.

PARSYC 9 is a tool based on a C++ grammar extended to manage also SystemC features. The authors claim that it is able to manage SystemC projects at high levels of abstraction, but current examples show only its RTL capabilities. It provides structural and syntactical information about the project (limited to RTL elements) obtained by a static analysis of the source code. PARSYC is suitable for project visualization and limited RTL synthesis. To develop and customize such a tool is less complex than the previous one; this time the management of the C++ grammar is required but, fortunately, such a task is supported by the *Purdue Compiler Construction Set*. The output format is legacy.

KaSCPar 9 is a tool based on a C++ grammar extended to manage also SystemC features and it is able to manage, with some limitations, SystemC projects at high levels of abstraction (TLM). It provides structural and syntactical information about the project (with some limitations) obtained by a static analysis of the source code. The tool is more suitable for project analysis and visualization but it has been designed to be adapted to different purposes. The effort needed to develop and customize KaSCPar seems to be quite high because it is required the management of the C++ grammar and it is required the knowledge about gcc data structures and the ability to manipulate them. The output format is XML.

SystemPerl 9 is a PERL library that provides also a SystemC parser. The actual applicability of this parser is limited because it provides only structural information. The design of SystemPerl is simple, while the extension is difficult because it is necessary to add all the missing features following, for example, one of the previously described approaches.

Last column of Table 1 summarizes the features of the parser presented in this work. Their comprehension requires the reading of next sections.

5. THE STRATEGY

SystemC design approach comprises a *simulation* and a *synthesis* flow.

As far as simulation is concerned, the tools are open-source; they include the SystemC class libraries (a set of macros and C++ classes implementing modules, signals and other hardware description language features added to the C++ language), and a lightweight simulation kernel, which schedules the concurrent processes composing the designs. In the simulation flow, user's models are treated as standard C++ sources, compiled via any C++ compiler and linked against a library of ready-made classes, to obtain a standard executable program. During preprocessing, in particular during the substitution of the macros provided in `systemc.h`, the user SystemC code is translated into plain C++; any hardware description oriented construct is lost (SystemC modules are rewritten as C++ classes, processes are rewritten as methods and so on). Since an actual C++ compiler is involved in the build process, any C++ compliant source code is a valid input for the simulation flow.

Things change significantly when we consider the synthesis flow: it is not any longer true that any C++ source is a valid input; instead, numerous constraints are set on the source code. Required coding guidelines are described in 99. The synthesis flow uses dedicated compilers that do not perform any macro substitution: instead, they interpret SystemC code in its purest semantics: modules, signals and processes have a direct impact on silicon. We would like to point out that virtually no C++ OO constructs are used, and also a number of C constructs are illegal (e.g., pointers, pointers to functions, etc.).

We modeled BSSC starting from a C grammar instead of a C++ one, because we found that BSSC is more "similar" to C than to C++. The practical benefits of such a choice impact on the global effort needed to implement the parser which is mainly due to the writing of the semantic rules/actions of the grammar: the number of C++ syntax rules is much higher than that of the C language (560 versus 213, see Table 2); furthermore, additional rules must be created *ex novo* to describe the specific constructs of SystemC.

Table 2 reports the actual size (number of rules and parser states) of the grammar of ANSI C 9, ISO C++ 9, and "BSSC from ANSI C". In order to extend C to BSSC 41 rules has been added to C.

Table 2: Number of rules and recognizer states for the considered grammars

Language	Rules	States
ANSI C	213	368
ISO C++	560	899
BSSC from C	254	451
BSSC from C++	595	955

We estimated that starting from C++, on the other hand, would have required to add 35 rules only, but it would have meant to write semantic actions for 595 rules. The difference in numbers (595 vs. 254) gives a rough idea of the difference in effort, but an additional, decisive factor is to be kept into account: a typical semantic action for C++ usually has to deal with OO concepts, and therefore demands much more effort than a typical C rule.

6. THE DESIGN

This section describes the design of the SystemC parser. The constituents of its architecture (Figure 1) are a lexical analyzer based on flex 9; a syntax analyzer based on bison 9; a set of semantic actions (implemented *ex novo* in C), which store the parsing results into a *PostgreSQL* database 9.

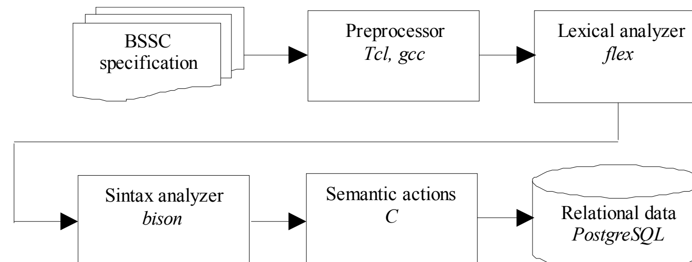

Figure 1: The structure of our SystemC front-end

6.1 The preprocessor

BSSC designs, as any C/C++ listing, can contain preprocessor directives, such as `#define` or `#include`; it is therefore clear that BSSC sources must undergo a standard preprocessing phase before being submitted to our parser. Anyway, an important detail must not be neglected: processing one of the above directive, namely `#include <systemc.h>`, which is present in all BSSC designs, implies including the definitions of macros SC_MODULE, SC_CTOR, SC_METHOD, SC_THREAD, SC_CTHREAD, etc., and expanding these macros means rewriting BSSC as standard C++ classes, structures and methods. This means that hardware-definition constructs are definitely lost, and that the parser – which is next in the tool chain – sees plain classes, structs and methods in place of them. As already said, this is acceptable for simulation (as far as the implemented classes, structures and methods behave consistently with expectations), but it is highly undesired for our purposes. Consequently, we prepared a simple preprocessing script, relying on `cpp` and `Tcl`, which prevents the expansion of `#include <systemc.h>`.

6.2 The lexical analyzer

The purpose of the lexical analyzer is to extract tokens from the BSSC specification. In order to recognize BSSC, the following additions were applied to the C lexical rules:

- rules recognizing C++ features not present in C, and truly required by BSSC (namely "//"-style one-line comments, the "::" class scope resolution operator and the *bool* type specifier keyword);
- rules recognizing SystemC hardware-description features like SC_MODULE, SC_METHOD, *sc_in_clk*, *sc_in*, *sc_out*, *sc_bit*, SC_CTOR, *sensitive*, *sensitive_pos*, *sensitive_neg* (and others, 20 in total).

The original C lexer 9 was composed of 90 rules, while our BSSC lexer comprises 113. The effort required to add the 23 new rules was negligible. Incidentally, for future applications, our lexer actually recognizes *all* SystemC tokens, not only BSSC.

6.3 The syntax analyzer

The grammar is the core of each parser, and the rules included in it directly affect the structure of the accepted language. In order to obtain a BSSC syntax analyzer, we designed a `yacc`-style BSSC grammar specification, starting from the ANSI C grammar 9 and applying the modifications described in the following.

A first set of rules was added or modified to support C++ features not present in C but present in BSSC:

- support for the *bool* type specifier adding the trivial rule *<type_specifier> ::= bool*;
- support for local variable declarations inside *for* constructs, as in *for(int i=0; i<N; i++) {...}*. To do that, the productions were extended as follows:

```
<iteration_statement> ::= WHILE ( <expression> ) <statement>
| DO <statement> WHILE ( <expression> ) ;
| FOR ( <expression_statement> <expression_statement> ) <statement>
| FOR ( <expression_statement> <expression_statement> <expression> ) <statement>
```

were changed as follows:

```
<iteration_statement> ::= WHILE ( <expression> ) <statement>
| DO <statement> WHILE ( <expression> ) ;
| FOR ( <simple_declaration> <expression> ) <statement>
| FOR ( <simple_declaration> <expression statement> <expression> ) <statement>

<simple_declaration> ::= <declaration_specifiers> <init_declarator_list> ;
| <declaration_specifiers> ;
| <init_declarator_list> ;
| ;
```

- mixture of declarations and statements inside a compound statement (a trivial modification in the right part of *<compound_statement>*);

- A second set of rules was created to recognize features peculiar to BSSC:
- support for BSSC types such as *sc_bit*, *sc_int<...>*, *sc_bv<...>*, etc. as built-in types, implementing parametric types without relying on C++ templates (6 new rules with left part *<type_specifier>* were introduced);
- support for SC_MODULE definitions and declarations constructs without requiring C++ classes (adding 2 new rules for left part *<external_declaration>* plus additional rules below);
- support for SC_CTOR module constructors without relying on C++ class methods (see the new rule for *<sysc_constructor>* below);
- support for method definition and without relying on C++ class methods (adding a new rule for left part *<direct_declarator>*);

The new grammar productions which best represent the impact of modifications just described are reported as follows:

```
<sysc_module_definition> ::= SC_MODULE ( <identifier> ) { <translation_unit> }
<sysc_module_instances> ;

<sysc_module_instances> ::= <init_declarator_list> <sysc_object_declaration> ::=
sc_in_clk <init_declarator_list> ;
| <sysc_object_type> <specifier_qualifier_list> <init_declarator_list> ;

<sysc_object_type> ::= sc_in | sc_out | sc_inout | sc_signal

<sysc_constructor> ::= SC_CTOR(<identifier>) { <sysc_constr_translation_unit> }

<sysc_constr_translation_unit> ::= <sysc_constr_declaration>
| <sysc_constr_translation_unit> <sysc_constr_declaration>

<sysc_constr_declaration> ::= <declaration>
| SC_CTHREAD ( <identifier> , <identifier> . <identifier> ( ) ) ;
| <sysc_sensitivity_edge> << <sysc_sensitivity_list> ;
| <sysc_sensitivity_edge> ( <identifier> , <postfix_expression> ) ;

<sysc_sensitivity_edge> ::= sensitive|sensitive_pos|sensitive_neg

<sysc_sensitivity_list> ::= <identifier>|<identifier> << <sysc_sensitivity_list>
```

6.4 Semantic actions

The semantic actions of a parser determine the way it behaves when each syntax construct is encountered; in essence, they determine the structure of the parser's output. In order to provide a generic front-end, actions are designed generate a generic output suitable to any cascaded application.

Our semantic actions store the structure of the given SystemC input and the "core" portion of the *parse tree* as entries of an SQL relational database, thus providing the next tools in the chain with powerful data access and high performance. Approaches based on custom APIs do not provide the above features.

The database we designed to model both structural and behavioral information contained in a BSSC source is made up of the following four tables:

project_files(OriginalFileName, PreProcName, DirtyFlag)

sc_modules(Position, ModuleName, OriginalFileName)

ports(ModuleName, PortName, PortType, PortWidth, PortSize, PortDirection, PreprocFile, OriginalFileName, StartLine, StartColumn)

syntaxtree(Module, Method, NodeId, ParentId, Species, Terminal, Type, PreprocF, FileName, StartL, StartC, EndL, EndC, Order)

Overall, the first three tables represent how modules are declared in source files to constitute projects; while the last one represents the *parse tree* of SystemC processes, methods and any other behavioral code.

The actual semantic rules rely on a simple database interaction API, composed of the following three functions:

```
void db_add_module(const char * module_name, coords_t start, coords_t end);

void db_add_port(const char * module_name, const char * port_name, const char *
port_type, const char * port_width, const char * port_size, const char *
port_direction, coords_t start);

void db_add_atom(const char * module_method_name, atom_t * atomp, int
parent_atom_id, int child_order);
```

Modularization and design-for-change principles were strongly enforced when designing the parser-database interface. Success is proven by the fact that semantic rules are composed of just 2.8 actual lines of C code on the average (semantic actions consist of 711 actual lines of C code, excluding comments and empty lines).

```
/* File: AES.cc */
#include <systemc.h>
#include "AES.h"
/*... other functions omitted ...*/

void AES::tran(){
    sc_bv<128> in,k;
    while(true){
            in=input.read();
            k=key.read();
            in=in^k;
            for (int i=0; i<9; i++){
                    in=SubByte(in);
                    in=Shift_Row(in);
                    in=MixColumn(in);
                    k=KeyExp(k, i);
                    in=in^k;
            }
        in=SubByte(in);
        in=Shift_Row(in);
        k=KeyExp(k, 9);
        in=in^k;
        output.write(in);
        wait();
    }
}
```

```
/* File: AES.h */
#include <systemc.h>
SC_MODULE(AES){
    static const sc_bv<8> rcon[10] =
    {"00001000", "00000100", "00000010", /* ... */ };
    static const sc_bv<8> Sbox_table[256] =
    {"01101100","11100011","11101110", /* ... */ };
    sc_in <sc_bv<128> > input;
    sc_in <sc_bv<128> > key;
    sc_out<sc_bv<128> > output;
    sc_in_clk clk;
    void tran();
    sc_bv<8> rcon_acc(int address);
    sc_bv<8> access(int address);
    sc_bv<8> SBox(sc_bv<8> in);
    sc_bv<128> Shift_Row(sc_bv<128> in);
    sc_bv<128> MixColumn(sc_bv<128> in);
    sc_bv<128> SubByte(sc_bv<128> in);
    sc_bv<32> SubByteKeyExp(sc_bv<32> in);
    sc_bv<32> RotByte(sc_bv<32> in);
    sc_bv<128> KeyExp(sc_bv<128> k, int n_round);
    SC_CTOR(AES) {
          SC_CTHREAD(tran,clk.pos());
    }
};
```

Figure 2. An excerpt of the example project

7. AN EXAMPLE OF USAGE OF THE SYSTEMC PARSER

In order to demonstrate the feasibility of our design strategy, we implemented a BSSC parser and validated it against a significant number of BSSC designs. To give an example of how our tool works, hereinafter we report about one of those tests, namely one implementing an AES encryption/decryption core.

The module interface and an excerpt of the implementation of the example project are shown in Figure 2 while Table 3 and Table 4 show tuples from `ports` and `syntaxtree` tables.

Table 3: Tuples in the `ports` table describing the structural interface of the AES sample module.

ModuleName	PortName	PortType	PortWidth	PortSize	PortDirection	PreprocFile	OriginalFileName	StartLine	StartColumn
AES	input	sc_bv	128	0	in	AES.i	AES.h	4	2
AES	key	sc_bv	128	0	in	AES.i	AES.h	5	2
AES	output	sc_bv	128	0	out	AES.i	AES.h	6	2
AES	clock	(clock)	0	0	in_clk	AES.i	AES.h	7	2

Table 4: Tuples in the `syntaxtree` table describing the parse tree of the code fragment of the AES module.

Module	Method	NodeId	ParentId	Species	Terminal	Type	PreprocF	FileName	StartL	StartC	EndL	EndC	Order
AES	tran	1850	0	Null		Compound	AES.i	AES.cc	205	2	221	10	0
AES	tran	1849	1850	Statement	while	Statement	AES.i	AES.cc	205	2	221	10	0
AES	tran	1742	1849	Terminal	true	Identifier	AES.i	AES.cc	205	8	205	11	0
AES	tran	1750	1849	Null		Compound	AES.i	AES.cc	206	4	221	10	1
AES	tran	1749	1750	Null	;	Compound	AES.i	AES.cc	206	4	206	19	0
AES	tran	1748	1749	Expression	=	Operator	AES.i	AES.cc	206	4	206	18	0
AES	tran	1743	1748	Terminal	in	Identifier	AES.i	AES.cc	206	4	206	5	0
AES	tran	1747	1748	Expression	()	Operator	AES.i	AES.cc	206	7	206	18	1
AES	tran	1746	1747	Expression	.	Operator	AES.i	AES.cc	206	7	206	16	0
AES	tran	1744	1746	Terminal	input	Identifier	AES.i	AES.cc	206	7	206	11	0
AES	tran	1745	1746	Terminal	read	Identifier	AES.i	AES.cc	206	13	206	16	1
AES	tran	1757	1750	Null	;	Compound	AES.i	AES.cc	207	4	207	16	1
AES	tran	1756	1757	Expression	=	Operator	AES.i	AES.cc	207	4	207	15	0
AES	tran	1751	1756	Terminal	k	Identifier	AES.i	AES.cc	207	4	207	4	0
AES	tran	1755	1756	Expression	()	Operator	AES.i	AES.cc	207	6	207	15	1
AES	tran	1754	1755	Expression	.	Operator	AES.i	AES.cc	207	6	207	13	0
AES	tran	1752	1754	Terminal	key	Identifier	AES.i	AES.cc	207	6	207	8	0
AES	tran	1753	1754	Terminal	read	Identifier	AES.i	AES.cc	207	10	207	13	1
AES	tran	1763	1750	Null	;	Compound	AES.i	AES.cc	208	4	208	11	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...

As far as the run time is concerned, the entire parsing process for the above example design (about 400 SystemC source code lines) takes 4 seconds on a Pentium III-class PC running Linux. The screenshot (taken from a visualization tool based on our parser) of the structural interface of the example module and the *parse tree* of the above implementation code fragment are represented in Figure 3.

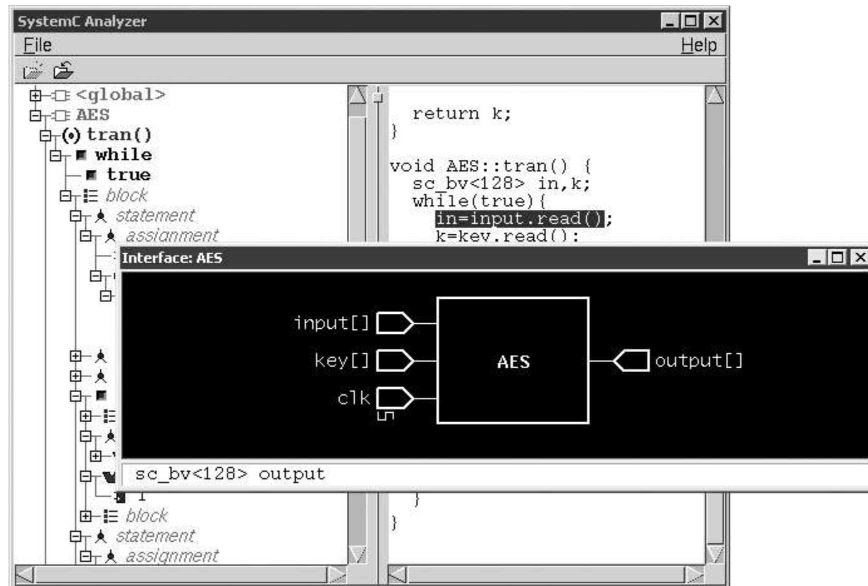

Figure 3: Visualization of the structural interface and the *parse tree* of the AES module

8. CONCLUSIONS

The paper reports about the design, development, and validation of a BSSC parser which meets the following global requirements:

- it relies on well-known, consolidated, open source tools, languages and technologies. In detail, we built our flow upon *lex* (in its *flex* implementation), *yacc* (in its *bison* implementation), and on the *PostgreSQL RDBMS*. The major merit of the new solution is the straightforwardness to be repeated and customized, when necessary;
- it is extensible: adding a new language keyword means just adding a line to the lexical definition of the language; adding a new language construct means just adding a BNF production to its syntax definition, and possibly a collection of simple semantic rules. The worst case is the addition of an entirely-new concept, which forces adding a table to the database and the semantic rules associated to its management.

In detail, the developed parser is based on a C grammar extended to manage BSSC features. It works at an abstraction level situated between TLM and RTL, so it allows to preserve an algorithmic view of the system and, at the same time, to keep the capability of exploiting HW/SW automatic synthesis opportunities (i.e., co-synthesis). It provides structural and syntactical information about the project (in connection with behavioural synthesizable elements) by means of a static code analysis. It is suitable as a front-end for analysis, visualization and co-synthesis tools. The efforts needed to develop and customize such a tool are low because of the adoption of the simple C grammar and a set of well-known open-source tools. The output format is standard (i.e., PostgreSQL tuples).

Future developments of this research will include the extension of the capability of the parser to manage also RTL SystemC code (which does not raise any conceptual difficulties with respect to BSSC) and the extension of the database schema in order to be able to represent higher-level concepts belonging to TLM specifications.

9. REFERENCES

[1] Synopsys, CoCentric SystemC Compiler Behavioral User and Modeling Guide, v2002.5.

[2] Synopsys, Describing Synthesizable RTL in SystemC, v1.2.

[3] J. Degener, ANSI C Yacc grammar, www.lysator.liu.se/c/ANSI-C-grammar-y.html.

[4] J. Degener, ANSI C grammar, Lex specification, www.lysator.liu.se/c/ANSI-C-grammar-l.html.

[5] E.D.Willink, Meta-compilation for C++, www.computing.surrey.ac.uk/research/dsrg/fog/FogThesis.html, 2001.

[6] The Free Software Foundation, Flex, www.gnu.org/manual/flex-2.5.4/flex.html

[7] The Free Software Foundation, Bison, www.gnu.org/manual/bison-1.35/bison.html

[8] PostgreSQL, www.postgresql.org/

[9] www.systemc.org

[10] www.forteds.com

[11] www.synopsys.com

[12] www.veripool.com

[13] greensocs.sourceforge.net/pinapa/

[14] www.fzi.de/KaSCPar.html

[15] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechlser. ParSyC: An efficient SystemC parser. In Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), pages 148–154, 2004.

[16] I. Yarom, G. Glasser. SystemC Opportunities in Chip Design Flow. IEEE ICECS 2004.

[17] www.cadence.com