SpecC/Synchronization Verification Tool (`S-VeT`)

Version 0.1

User's Manual

Thanyapat Sakunkonchak

Fujita Laboratory
The University of Tokyo

January 11, 2007.

Contents

1 Introduction **3**

2 Download and Installation **4**

3 Usage **5**
3.1 Generate `DESIGN.sir` 5
3.2 Options 5
3.2.1 Help option 5
3.2.2 Display `.sir` tree structure options 6
3.2.3 Verification options 6

4 Tutorial **7**
4.1 Display tree structure 7
4.2 Verification 9
4.2.1 Simple example 10
4.2.2 Verify simple example 12
4.2.3 Results explanation 13
4.3 Testbenches 14

5 Unsupported Syntaxes & Modeling Styles **15**

Acknowledgment

This work is supported by Semiconductor Technology Academic Research Center (STARC) [2] and Japan Society for the Promotion of Science (JSPS) [1].

Chapter 1

Introduction

`S-VeT` [60, 61, 62] is a tool that used to verify synchronization in SpecC descriptions [54, 34, 35, 33]. Implementation of `S-VeT` is similar to other well-known C verifiers like SLAM [4, 3, 5], BLAST [67, 41, 42, 40] and MAGIC [14]. However, in order to cope with non-deterministic behavior of statement execution in SpecC, execution of every statement is described with equality/inequality formulae and solved by using commercial ILP solver.

Chapter 2

Download and Installation

S-VeT has been successfully installed on Fedora Core 3, 4 and GCC 3.0-3.4. You can obtain the latest S-VeT binary distribution from

http://www.cad.t.u-tokyo.ac.jp/~thong/S-VeT/

In order to run S-VeT smoothly, you will need to install the following tools.

- NuSMV-2.2.3
  http://nusmv.irst.itc.it/
- CVC Lite
  http://www.cs.nyu.edu/acsys/cvcl/
- ILOG Cplex
  http://www.ilog.com/products/cplex/

Let say "/foo" is your working directory. Copy the following binaries into /foo or make them globally known (copy to /usr/bin or /usr/local/bin). Please make sure that all the binaries are having the same name as the followings (case-sensitive).

- NuSMV
- cvcl
- cplex
- S-VeT

Chapter 3

Usage

S-VeT is a command line based tool. S-VeT can performed various functions according given options in command line.

```
>> S-VeT [OPTION]... [DESIGN].sir ↵
```

3.1 Generate DESIGN.sir

The DESIGN to verify is given in the binary SpecC Internal Representation (.sir) file. S-VeT has been tested to work with .sir file generated by SpecC reference compiler version 2.2.0. This may not be working with the .sir file that generated by other versions of SpecC reference compiler.

To generate .sir file:

```
>> sir_gen DESIGN ↵ (no extension, just the DESIGN name)
```

3.2 Options

Mandatory arguments to long options are mandatory for short options too.

3.2.1 Help option

- `-h, --help`
  Display help and exit

3.2.2 Display `.sir` tree structure options

- `-b, --branch`
  Print the branches of the tree
- `-c, --child`
  List child behaviors in calling order
- `-f, --flatten`
  Flatten the tree (no indentation)
- `-l, --longlist`
  Long listing: list flags with each class
- `-t, --type`
  Print type information with each instance
- `-B, --channelonly`
  Exclude behaviors (list channels only)
- `-C, --behavioronly`
  Exclude channels (list behaviors only)

3.2.3 Verification options

- `-a, --abstraction`
  Apply abstraction & refinement to the design
- `-L, --showlist`
  Print lines & related variables list
- `-G, --showgraph`
  Print Control Flog Graph (CFG)

Chapter 4

Tutorial

In this chapter, we provide an easy and simple tutorial for the use of `S-VeT`. Examples which shown in this chapter (except for `ppp`) are provided together with `S-VeT` binary.

4.1 Display tree structure

This section shows how to use `S-VeT` to display the tree structure of various `.sir` files.

Example

If we would like to see what tree structure of the behaviors and channels of predicate abstraction example (`predabs1.sir`), double lock/unlock detection example (`lock_unlock.sir`), communication example (`communication.sir`), Point-to-Point Protocol design (`ppp.sir`), we can use the following options to view branch, longlist and type of each modules.

`predabs1.sir`

```
>> S-VeT -blt predabs1.sir ↵
>> B i o   behavior Main
```

`lock_unlock.sir`

```
>> S-VeT -blt lock_unlock.sir ↵
>> B i o   behavior Main
```

communication.sir

```
>> S-VeT -blt communication.sir ↵
>> B i o   behavior Main
>> B i l   |------ XBhvr1 b1
>> B i l   |------ XBhvr2 b2
>> C i l   \------ Chnl2 c2
>>
>> C i l   channel ip_packet_channel
```

ppp.sir

```
>> S-VeT -blt ppp.sir ↵
>> B i o   behavior Main
>> B i l   |------ rx_byte test_byte_read
>> B i l   |------ tx_byte test_byte_write
>> B i l   |------ gen_clk16 test_gen_clk16
>> B i l   |------ local_echo_read test_local_echo_read
>> B i l   |------ local_echo_write test_local_echo_write
>> B i l   |------ rx_ppp test_ppp_recv
>> B i l   |------ tx_ppp test_ppp_send
>> B i l   |------ rx_bit test_rx_bit
>> B i o   |------ rx_clock_wrapper test_rx_clock_wrapper
>> B i l   |       |------ rx_clk_gen rxd_clk_gen
>> B i l   |       \------ dummy trap_dummy
>> B i l   |------ tx_bit test_tx_bit
>> B i l   |------ tx_clk test_tx_clk
>> C i l   |------ byte_channel rx_byte_channel
>> C i l   |------ ip_packet_channel rx_ip_packet_channel
>> C i l   |------ ppp_packet_channel rx_ppp_packet_channel
>> C i l   |------ byte_channel tx_byte_channel
>> C i l   |------ ip_packet_channel tx_ip_packet_channel
>> C i l   \------ ppp_packet_channel tx_ppp_packet_channel
```

4.2 Verification

The goal of `S-VeT` is to verify the design written in SpecC descriptions. Verification can be divided into two methods.

- **Checking invariant of error label statement:** In other words, check whether the error label statement is reachable. This uses techniques in software verification called *predicate abstraction* and *counter-example-guided abstraction refinement (CEGAR).*
- **Validate execution order:** SpecC contains concurrency (for describing hardware part) and the execution order of statements among concurrent processes is in non-deterministic fashion. In order to validate the non-deterministic execution order, we represent execution of statements in SpecC with equalities/inequalities formulae. Since each timing for each statement is non-deterministic, we can validate the execution order of SpecC descriptions by validating this formulae. This can be done by using *integer linear programming (ILP)* technique.

In this version of `S-VeT`, we include only the first method. Second method will be added in latter version.

Further reading on *model checking*, *CEGAR* and all related researches can be found in [56, 26, 50, 15, 39, 30, 69, 63, 64, 52, 7, 8, 29, 16, 9, 10, 6, 44, 11, 13, 19, 12, 51, 18, 46, 57, 37, 31, 49, 70, 36, 38, 55, 17, 59, 58, 43, 32, 20, 25, 68, 21, 53, 71, 27, 48, 5, 33, 65, 60, 28, 61, 24, 47, 23, 22, 66, 41, 42, 40, 62, 45].

4.2.1 Simple example

Consider the following example for double lock/unlock checking written in SpecC. We would like to check whether, if the current state is at lock (or unlock), it will not visit lock (or unlock) state again. See state transition of this example in 4.1.

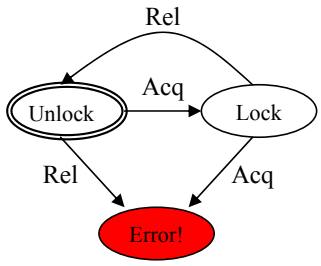

Figure 4.1: Double lock/unlock detection example

Error statements which we would like to check whether it is reachable are labeled by `ERROR_LABEL`#*number*. In order to validate whether any statement is reachable, the keyword `ERROR_LABEL` must be used. For #*number*, it can be any numerical number as long as there is no duplication for every error statement.

```
behavior Main() {
   int main() {
      bool temp, request, lock;
      int x, x_old;
      lock = false;
      x_old = -1;
      x = 0;

      do{
         /* lock */
         if(lock)
            ERROR_LABEL1:;
         else
            lock = true;

         x_old = x;
         if(request){
            request = false;
            /* unlock */
            if(!lock)
               ERROR_LABEL2:;
            else
               lock = false;

            x = x + 1;
         }
      }while(x != x_old);

      /* lock */
      if(!lock)
         ERROR_LABEL3:;
      else
         lock = false;

      return 0;
   }
};
```

Figure 4.2: Detail descriptions of Fig. 4.1 written in SpecC

4.2.2 Verify simple example

The following command is used to verify this example.

lock_unlock.sir

```
 >> S-VeT -a lock_unlock.sir ↵
 1) ************************************************************************
 2) * Program: SpecC/Synchronization Verification Tool (S-VeT)           *
 3) * Author:  Thanyapat Sakunkonchak                                    *
 4) * Affil:   Fujita Lab., University of Tokyo                          *
 5) * Sponsor: Semiconductor Technology Academic Research Center (STARC) *
 6) *          Japan Society for the Promotion of Science (JSPS)         *
 7) * Send bug-reports and/or questions to: thong@cad.t.u-tokyo.ac.jp    *
 8) ************************************************************************
 9) [DesignName X = lock_unlock]
10) Generate abstract program & syntax...  X.abs  X.ast
11) Generate pre-analysis list...  X.event
12) Generate list of all abstract details...  X.list
13) -- Bhvr:Iteration #0 --
14) CE-1.1 is infeasible (need refinement)
15) CE-1.2 is infeasible (need refinement)
16) CE-1.3 is infeasible (need refinement)
17) -- Bhvr:Iteration #1 --
18) CE-1.1 is infeasible (need refinement)
19) CE-1.2 is infeasible (need refinement)
20) -- Bhvr:Iteration #2 --
21) Properties satisfied.
22) ##### Statistics #####
23) Pre-processing (s):    0.035
24) BHVR Abs,MC,Ref(s):    1.320    0.538    0.969
25) Total run-time (s):    2.862
```

4.2.3 Results explanation

The results of running command "`S-VeT -a lock_unlock.sir` ↵" are shown with lines `1-25`. We can explain these results as follows.

- Line `1-8`: the header comments of our tool
- Line `9`: the design name
- Line `10`: the abstract program and abstract syntax are written in files `lock_unlock.abs` and `lock_unlock.ast`
- Line `11`: the list of events is stored in file `lock_unlock.event`
- Line `12`: details of abstraction store in `lock_unlock.list`
- Line `13-21`: CEGAR iterations
  - `Bhvr` denotes verification of behaviors while `Chnl` denotes verification of channels. Note that channels are verified separately prior to behaviors. This is to make sure that there is no deadlock when we use those channels. In this example we have only `Bhvr` because there is no channel in the design.
  - `Iteration #I` denotes the `I` iteration
  - `CE-X.Y` denotes the counterexamples. For `Bhvr`, `X` has the only value `1` (because we interpret all behaviors at once), where for `Chnl`, `X` equals to the number of all channels (we verify each channel separately). `Y` is the number of counterexamples according to that `X`.
  - For each iteration, these are following results.
    - `CE-X.Y is infeasible (need refinement)`
    - `CE-X.Y is feasible (real CE)`
    - `Properties satisfied`
  - For iteration `0`, there are 3 counterexamples and all are needed to be refined.
  - For iteration `1`, there are 2 counterexamples (one is disproved) and all are needed to be refined.

  - For iteration 2, no counterexample because all the properties are satisfied.

- Line 22-25: Runtime results
  - Line 23: Pre-processing time
  - Line 24: Model checking time
    * Abs - Time taken by abstraction process
    * MC - Time taken by model checker
    * Ref - Time taken by refinement process
  - Line 25: Total runtime

4.3 Testbenches

S-VeT comes with the following testbenches.

- predabs1
- predabs2
- predabs3
- lock_unlock

If you have more examples that you would like to provide us, you can mail them to thong@cad.t.u-tokyo.ac.jp.

Chapter 5

Unsupported Syntaxes & Modeling Styles

There are some SpecC syntaxes or some SpecC modeling (programming) styles that cannot be run with `S-VeT`.

- The use of `fsm{}` syntax
- The use of `pipe{}` syntax
- The use of **global** variables, i.e. declare variables outside behaviors/channels. To avoid this, all variables can be declared inside behaviors/channels or pass them through the ports of behaviors/channels.
- Complex/deep hierarchical design. This can be avioded by combine/flatten some modules.

We intend to cover these in the future version of `S-VeT`.

Bibliography

[1] Japan society for the promotion of science (JSPS). http://www.jsps.go.jp.

[2] Semiconductor technology academic research center (STARC). http://www.starc.jp.

[3] T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, Febuary 2000.

[4] T. Ball and S. K. Rajamani. *Boolean Programs: A Model and Process for Software Analysis.* Microsoft Research, http://research.microsoft.com/slam, .

[5] T. Ball and S. K. Rajamani. The SLAM toolkit. In *Proc. of the International Conference on Computer-Aided Verification (CAV'01)*, Paris, 2001.

[6] M. C. Browne and E. M. Clarke. SML: A high level language for the design and verification of finite state machines. In *IFIP WG10.2 Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs*. IFIP, 1987.

[7] M. C. Browne, E. M. Clarke, and D. L. Dill. Checking the correctness of sequential circuits. In *Proc. of the 1985 International Conference on Computer Design*. IEEE, 1985.

[8] M. C. Browne, E. M. Clarke, and D. L. Dill. Automatic circuit verification using temporal logic: Two new examples. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. Elsevier, 1986.

[9] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transaction on Computers*, 35(12), December 1986.

[10] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 35(8), August 1986.

[11] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proc. of the 1991 International Conference on VLSI*, August, 1991.

[12] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Long. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4), April 1993.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2), 1992.

[14] Carnegie Mellon University (CMU). Modular analysis of programs in c (MAGIC). http://www.cs.cmu.edu/~chaki/magic.

[15] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop*, LNCS 131, Yorktown Heights, NY, 1981. Springer-Verlag.

[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.

[17] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Conference on Correct Hardware Design and Verification Methods*, 1999.

[18] E. M. Clarke, O. Grumberg, and D. E. Dill. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5), September 1994.

[19] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the futurebus+ cache coherence protocol. In D. Agnew, L. J. M. Claesen, and R. Camposano, editors, *11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL '93*, North Holland, 1993. IFIP.

[20] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proc. of the International Conference on Computer-Aided Verification (CAV'00)*, Volume 1855 of LNCS. Springer-Verlag, 2000.

[21] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, January 2000.

[22] E. M. Clarke, H. Jain, and D. Kroening. Verification of SpecC using predicate abstraction. In *Second ACM-IEEE International Confernece on Formal Methods and Models for Codesign (MEMOCODE2004)*, 2004.

[23] E. M. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proc. of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.

[24] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. In *Proceeding of the Model Checking for Dependable Software-Intensive Systems Workshop*, San-Francisco, USA, 2003.

[25] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proc. of the 22nd International Conference on Software Engineering (ICSE2000)*, 2000.

[26] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium of Programming Language*, 1977.

[27] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Proceeding of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001)*, 2001.

[28] S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *4th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2002)*, 2002.

[29] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings*, Part E 133(5), 1986.

[30] E. A. Emerson. *Branching Time Temporal Logic and the Design of Correct Concurrent Programs*. PhD Thesis, Harvard University, 1981.

[31] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2), 1996.

[32] E. A. Emerson and R. J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Correct Hardware Design and Verification Methods (CHARME'1999)*, Volume 1703 of LNCS, 1999.

[33] M. Fujita and H. Nakamura. The standard SpecC language. In *International Symposium on Systems Synthesis (ISSS2001)*, Montreal, Canada, 2001. ACM.

[34] D. G. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publisher, March 2000.

[35] A. Gerstlauer, R. Doemer, J. Peng, and D. G. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publisher, May 2001.

[36] P. Godefroid. Model checking for programming languages using verisoft. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages*, Paris, 1997.

[37] P. Godefroid, D. Peled, and M. G. Staskauskas. Using partial order methods in the formal verification of industrial concurrent programs. In *International Symposium on Software Testing and Analysis (ISSTA'96)*, 1996.

[38] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proceeding of the International Conference on Computer Aided Verification (CAV'97)*, Volume 1254 of LNCS. Springer-Verlag, 1997.

[39] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[40] T. A. Henzinger, R. Jhala, R. Mujumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. of the 31st Annual Symposium on Principles of Programming Languages (POPL).* ACM Press, 2004.

[41] T. A. Henzinger, R. Jhala, R. Mujumdar, and G. Sutre. Lazy abstraction. In *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.

[42] T. A. Henzinger, R. Jhala, R. Mujumdar, and G. Sutre. Lazy abstraction. In *Proc. of the 10th SPIN Workshop on Model Checking Software (SPIN)*, Volume 2648 of LNCS. Springer-Verlag, 2003.

[43] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *Proc. of the 21st International Conference on Software Engineering (ICSE1999)*, 1999.

[44] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proc. of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46. ACM Press, 1988.

[45] H. Jain, F. Ivancic, A. Gupta, and M. K. Ganai. Localization and register sharing for predicate abstraction. In N. Halbwachs and L. Zuck, editors, *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, 2005.

[46] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[47] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *Proceeding of the International Conference on Computer Aided Verification (CAV'03)*, Volume 2725 of LNCS, Colorado, USA, 2003. Springer-Verlag.

[48] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, 2001.

[49] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based abstraction for CTL model checking. In *Proc. of the International Conference on Computer-Aided Verification (CAV'96)*. Springer-Verlag, 1996.

[50] Y. Malachi and S. S. Owicki. Temporal specifications of self-timed systems. In H. T. Kung, B. Sproull, and G. Steele, editors, *VLSI Systems and Computations*. Computer Science Press, 1981.

[51] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishing, 1993.

[52] B. Mishra and E. M. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science*, 38, 1985.

[53] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering and efficient SAT solver. In *Proc. of the Design Automation Conference (DAC'01)*, 2001.

[54] U. of California Irvine (UCI). The SpecC system. http://www.ics.uci.edu/∼specc.

[55] A. Pardo and G. D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference (DAC'98)*, 1998.

[56] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*. IEEE Computer Society Press, 1977.

[57] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Programming Languages (POPL'95)*, 1995.

[58] H. Saidi. Modular and incremental analysis of concurrent software systems. In *14th IEEE International Conference on Automated Software Engineering (ASE'99)*, 1999.

[59] H. Saidi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Proceeding of the International Conference on Computer Aided Verification (CAV'99)*, Volume 1633 of LNCS. Springer-Verlag, 1999.

[60] T. Sakunkonchak and M. Fujita. Verification of synchronization in SpecC description with the use of difference decision diagrams. In *Forum on specification & Design Languages (FDL'02)*, Marseille, France, 2002.

[61] T. Sakunkonchak, S. Komatsu, and M. Fujita. Verification of synchronization in SpecC descriptions with the use of difference decision diagrams. *IEICE Trans. on Special Section. on VLSI Design and CAD Algorithms*, E86-A(12), December 2003.

[62] T. Sakunkonchak, S. Komatsu, and M. Fujita. Synchronization verification in system-level design with ILP solvers. In *Third ACM-IEEE International Confernece on Formal Methods and Models for Codesign (MEMOCODE2005)*, Verona, Italy, 2005.

[63] A. P. Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD Thesis, Harvard University, 1983.

[64] A. P. Sistla and E. M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM*, 32(3), 1985.

[65] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A cooperating validity checker. In E. Brinksma and K. G. Larsen, editors, *14th International Conference on Computer Aided Verification (CAV'02)*, Lecture Notes in Computer Science, Copenhagen, Denmark, 2002. Springer-Verlag.

[66] K. Tanabe, S. Shunsuke, and M. Fujita. Program slicing for system level designs in SpecC. In *IASTED International Conference on Advances in Computer Science and Technology (ACST2004)*, 2004.

[67] University of California Berkeley. Berkeley abstraction software verification tool (BLAST) project. http://mtc.epfl.ch/software-tools/blast.

[68] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. of the 15th International Conference on Automated Software Engineering (ASE'2000)*, Grenoble, 2000.

[69] G. von Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers*, C-31(3), 1982.

[70] H. Zhang. SATO: An efficient propositional prover. In *Proc. of the Conference on Automated Deduction (CADE'97)*, 1997.

[71] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. of International Conference on Computer Aided Design (ICCAD'01)*, 2001.