Scalable Hybrid Verification for Embedded Software

Jörg Behrend, Djones Lettnin, Patrick Heckeler,
Jürgen Ruf, Thomas Kropf and Wolfgang Rosenstiel
Wilhelm-Schickard-Institute for Computer Science
Department of Computer Engineering
University of Tübingen
Sand 13
72076 Tübingen, Germany
E-mail: {behrend,lettnin,heckeler,ruf,kropf,rosenstiel}@informatik.uni-tuebingen.de

***Abstract*—The verification of embedded software has become an important subject over the last years. However, neither stand-alone verification approaches, like simulation-based or formal verification, nor state-of-the-art hybrid/semiformal verification approaches are able to verify large and complex embedded software with hardware dependencies. This work presents a new scalable and extendable hybrid verification approach for the verification of temporal properties in embedded software with hardware dependencies using for the first time a new mixed bottom-up/top-down algorithm. Therefore, new algorithms and methodologies like static parameter assignment and counter-example guided simulation are proposed in order to combine simulation-based and formal verification in a new way. We have successfully applied this hybrid approach to embedded software applications: Motorola's Powerstone Benchmark suite and a complex industrial embedded automotive software. The results show that our approach scales better than stand-alone software model checkers to reach deep state spaces. The whole approach is best suited for fast falsification.**

I. Introduction

Embedded software (ESW) is omnipresent in our daily life. It is responsible for controlling car functions, telecommunication products, electrical appliances, robots, medical devices and aircrafts to mention only a few examples. It plays a key role in overcoming the time-to-market pressure and providing new functionalities, like reduction of fuel emissions, improvement of security and comfort. Therefore, a high number of users are dependent on the services provided by embedded software.

The increasing processing power, the decreasing power consumption, the decrease in prices and the structural size minimization of microcontrollers and microprocessors make it possible to move more functionalities from hardware to embedded software which ends up in million lines of code and a fast increasing complexity [1].

These functionalities are mostly implemented using the C language at different levels of implementation. Software applications, which have in most cases no direct access to the hardware (HW), provide high level operations to the user. On the other hand, drivers or firmwares, which are mostly hardware-dependent (low level) software, have direct access to the hardware based on pointers, interrupts or inline assembly [2]. Most of the fatal errors occur in hardware-dependent software (e.g., low level drivers) and not in application software. Some examples of severe coding errors at low level drivers are finite-state machine errors, timing errors, stack/memory overflow errors and inconsistence errors (e.g., in non-volatile memory) [3]. Therefore, the verification of hardware-dependent embedded software is of fundamental importance.

The most commonly used approaches to verify embedded software (ESW) are based on simulation or formal verification (FV) approaches. Testing, co-debugging and/or co-simulation techniques result in a tremendous effort to create test vectors. Furthermore, critical corner case scenarios might remain unnoticed. Assertion-based verification (ABV) methodology captures a design's intended behavior in temporal properties and monitors the properties during system simulation. This methodology has been successfully used at lower levels of hardware designs, which are not suitable for software. ESW has no timing reference and contains more complex data structures (e.g., integers, pointers) requiring a new mechanism to apply an assertion-based methodology. In order to verify temporal properties in ESW, formal verification techniques are efficient, but only up to medium sized software systems. For more complex software designs, formal verification using model checking often suffers from the state space explosion problem. Therefore, abstraction techniques (e.g., predicate abstraction [4]) are applied to take the load of the back-end model checker.

Semiformal or hybrid approaches have been proposed many times before with only limited success. Therefore, this paper presents a new scalable hybrid verification approach using for the first time a new mixed bottom-up/top-down algorithm, which combines assertion-based verification and formal verification based on state-of-the-art software model checkers (SMCs). It provides new algorithms and methodologies for hybrid verification, that is static parameter assignment and counterexample guided simulation. This hybrid approach allows to reach deep state spaces (compared to software model checkers) and improves the state space coverage relative to a simulation-based verification approach. Our approach employs the SystemC Temporal Checker (SCTC) [5] and different software model checkers (e.g., CBMC [6] and ESBMC [7]). During the bottom-up/formal exploration phase software model checkers try to verify individual modules or functions until a time bound or memory limit has been reached. SCTC is used to verify temporal properties in ESW during the top-down/hybrid phase and it can also be used to initialize/pre-define variables and other structures of the function under test in order to reduce the state space (formal phase). Informations from counterexamples are used to guide the simulation (learning process). For instance, the randomization of input variables in our testbench is constrained to generate more efficient test vectors. Therefore, both techniques complement each other in order to enable the verification of large industrial hardware-dependent software. The whole approach is intended for early bug detection using fast debugging iterations. Therefore, it is best suited for fast falsification.

The paper is organized as follows. Section II summarizes the related work. Section III describes the verification methodology. Section IV presents the technical details. Section V summarizes our case studies and presents the results. Section VI concludes this paper and describes the future work.

II. Related Work

BLAST [8] and SATABS [9] are formal verification tools for ANSI-C programs. Both programs use predicate abstraction mechanisms to enhance the verification process and support satisfiability modulo theory solver (SMT) [10] as verification back-end. Due to the scalability of recent SMT-solvers, more and more software model checkers are using them as back-ends. C bounded model checking (CBMC) [6] has proven to be a successful approach for automatic software analysis. CBMC is a bounded model checker for ANSI-C developed by Kroening *et al.* The key idea is to build a propositional formula whose models correspond to program traces (with bounded length) that violate some given property and then use state-of-the-art SAT solvers to check the formula for satisfiability. Armando *et al.* [11] implemented SMT support into CBMC. Kroening *et al.* [6] have also enhanced CBMC to support different SMT solvers. Codeiro *et al.* [7] have implemented ESBMC based on the front-end of CBMC and a new back-end based on SMT. Lettnin *et al.* [12] have presented an assertion-based verification approach to verify temporal properties for hardware independent software. However, in embedded software with hardware dependencies, simulation has to consider more test cases for the hardware-software interface variables requiring higher efforts in the generation of test cases (e.g., pointers that enable direct accesses to the hardware, which are commonly used to set the hardware registers). This results in even more coverage limitations. Semiformal/hybrid verification approaches have been applied successfully to hardware verification [13], [14]. However, the application of a current semiformal hardware model checker to verify embedded software is not viable for large industrial programs [15]. In the area of embedded software using C language, Lettnin *et al.* [16] proposed a semiformal verification approach based on simulation and symbolic model checking. However, the symbolic model checker (SymC) [17] was the bottleneck for the scalability of the formal verification. The model checker SymC was originally developed for the verification of hardware designs. Cordeiro *et al.* [18] have published a semiformal approach to verify medical software. But they have scalability problems caused by the used model checker. The aforementioned related work have their pros and cons. However, they still have scalability limitations in the verification of complex hardware-dependent software.

A. Contributions

As shown in related work it is hard or sometimes even impossible to select the suitable approach for the verification of ESW (compatible with MISRA [19]). To overcome these cons we take advantage of the pros of the related work and combine different state-of-the-art formal-based and simulation-based approaches to a new hybrid verification approach. This paper presents for the first time a new mixed bottom-up/top-down algorithm. The main contributions are:

- Hybridization:
  - The simulation supports the formal verification e.g., static parameter assignment to shrink the state space.
  - Formal SMCs support the simulation process e.g., counterexample guided simulation.
- Automated Testbench:
  - Use of a simulation approach with SystemC model derivation from ESW. No abstraction is used. Therefore, the derived model is as precise as the original C program.
  - An automatically generated testbench with randomized input variables is included.
- Scalability:
  - Splitting of the verification task into independent subtasks.
  - Distribution of the verification process among different computational nodes (e.g., single multicore node, cluster).
- Expandability:
  - Adaptable to different software model checkers.

III. Verification Methodology

The verification approach consists of three phases: preprocessing, bottom-up and top-down. Additionally, an orchestrator coordinates the aforementioned phases and the interaction between simulation and formal verification. We start a verification run with preprocessing the C code. As we have seen in the related work, the state-of-the-art software model checkers suffer from the embedded software complexity. In order to overcome this complexity, we developed a mixed bottom-up/top-down approach. Fig. 1 shows the application flow as a whole. Details about the single steps are given below.

A. Software Preprocessing

The C code is preprocessed (Fig. 1, lines 3-12). This process consists of following parts:

1) Conversion of the C program into three-address code (3-AC) and merge the C source code into one single file using CIL [20]. 3-AC is normally used by compilers in order to support code transformations and it is easier to handle compared to the degrees of freedom of a user implementation (Fig. 1, lines 5-6).
2) A SystemC model is derived from the embedded software. No abstraction is used and therefore, the derived model is as precise as the original C program. A testbench is automatically generated and all input variables are randomized with constrains (Fig. 1, line 7-8).
3) A function call graph (FCG) [21] is generated (see Fig. 2 and Fig. 1, lines 9-10). We use this FCG as input to guide our bottom-up verification.
4) We include the user-defined properties into the C code. Therefore, we translate the LTL-style properties into assert/assume statements based on [22] (Fig. 1, lines 11-12).

B. Bottom-up Phase/Formal Exploration

After preprocessing the C code we start the bottom-up verification (Fig. 1, lines 14-20). We verify all functions of the FCG beginning with the leaves using state-of-the-art software model checkers with build-in properties and the user-defined properties specified in LTL. We distribute the computation controlled by the orchestrator of every function to a different verification instance of the supported software model checkers (SMC) (Fig. 3). The default distribution heuristic is a “try-all” approach, which means that all functions are checked with all supported SMCs. Furthermore, the user can orchestrate the distribution of the functions manually and choose between the different SMCs. If it is not possible to verify all functions of the FCG using the SMCs (bottom-up/exploration), we switch to the hybrid top-down phase. Therefore, a marked FCG is returned including the status of the verification of all functions.

C. Top-down Phase/Hybrid Verification

The hybrid top-down phase (Fig. 1, lines 26-41) starts with the analysis of the marked FCG (mFCG). All functions, that were not yet verified due to failed verification (e.g., time out (TO) or out of memory (MO)) are marked as point of interest (POI) (Fig. 1, lines 21-25).

We use a simulation approach based on SystemC. Therefore, we derive a SystemC model from the embedded software and apply SCTC, which supports specification of user defined properties in LTL [23]. The derived model is automatically generated using no abstractions. An automatically generated testbench is included with all input variables constrained. For refinement of the testbench we monitor the behavior using coverage metrics. The derived model consists of one SystemC class (`ESW_SC`) mapped to a corresponding C program. The `main` function in C is converted into a SystemC process (`SC_THREAD`). Since software itself does not have any clock information, we propose a new timing reference using a program counter event (`esw_pc_event`) [12]. The automatically generated testbench includes all input variables an it is possible to choose between different randomization strategies like constrained randomization and different random distributions, supported by the SystemC Verification Library (SCV) [24].

The orchestrator monitors the simulation process-properties and variables-during the hybrid phase based on SystemC Temporal Checker (SCTC) [5] in order to start a new formal verification process at every POI (Fig. 1, line 32). We use the monitored information to initialize variables (interaction with formal) to statically assign parameters (Fig. 1, line 34) and to

```
Verifyr($Cprog$,$prop$)                                      1
{                                                            2
  preProcess($Cprog$)                                        3
  {                                                          4
    cil ($Cprog$)                                            5
      return $preC$                                          6
    c2sctc ($Cprog$)                                         7
      return $simC$                                          8
    genFCG($preC$)                                           9
      return $FCG$                                           10
    ltl2Assert($preC$, $prop$) //manual task                 11
      return($propC$)                                        12
  }                                                          13
  startBOTTOM−UP($propC$, $FCG$)                             14
  {                                                          15
    while functionsNotChecked                                16
      distribute()                                           17
        checkFunctions()                                     18
      return $VerifiedFunctions$                             19
  }                                                          20
  markFCG($VerifiedFunctions$)                               21
  {                                                          22
    visualize()                                              23
    return $mFCG$;                                           24
  }                                                          25
  startTOP−DOWN($preC$, $simC$, $prop$, $mFCG$)              26
  {                                                          27
    while $time < timeBound$                                 28
      simulate($simC$, $prop$, $mFCG$)                       29
        extractPOI($mFCG$)                                   30
        return $POI$                                         31
        if $actualState == POI$ then                         32
          saveState()                                        33
          staticParameterAssignment()                        34
          distribute()                                       35
          checkFunctions()                                   36
            if $counterexample == TRUE$ then                 37
            guideSimulation()                                38
      update($mFCG$)                                         39
    visualize()                                              40
    }                                                        41
}                                                            42
```

Fig. 1. Algorithm

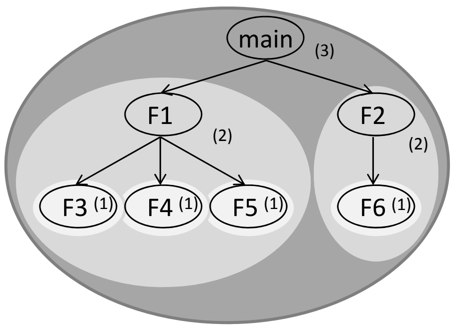

Fig. 2. Hybrid verification approach

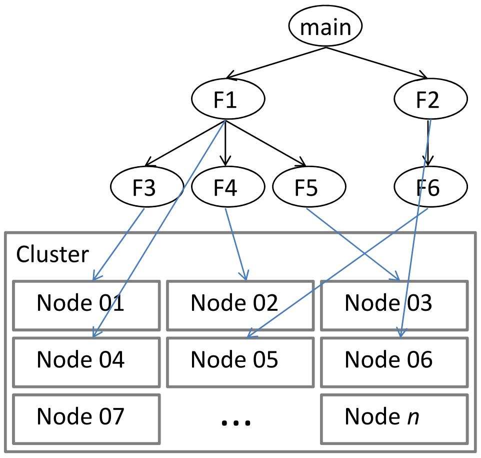

Fig. 3. Distribution of computation

create a temporary version of the source code of the function under test (FUT). These functions under test are distributed and checked with the formal SMCs (Fig. 1, lines 35-36). Static parameter assignment will lead to different access points for the software model checkers and it will help shrinking the state space of the function. Therefore, the formal verification benefits from the simulation.

If a counterexample is reported, this information is used to guide the simulation (learning process). For instance, the randomization of input variables in our testbench is constrained in order to generate more efficient test vectors (Fig. 1, lines 37-38). To present the counterexample to the user we save the global variable assignment of the used simulation run ("seed") to trace back from the counterexample given by the SMC to the entry point of the simulation run. Then we translate the CIL generated information back to the original C code. The checked properties are the same as in bottom-up phase.

IV. Technical details

The main task of this new approach is to provide a scalable and extendable hybrid verification service. We have implemented our new approach as a verification platform called Verifyr which can verify embedded software in a distributed and hybrid way. To make use of the advantage of several CPU cores on more than one computing node, we have to split the whole verification process into multiple verification jobs. Furthermore, Verifyr is platform independent and extendable by using a standard communication protocol to exchange information. The Verifyr framework provides a service to verify a given source code written in C language. It consists of a collection of formal verification tools such as CBMC and ESBMC and simulation tools (e.g., SCTC) and a communication gateway in order to invoke verification commands and to exchange status information of the hybrid verification process. These commands are passed to the orchestrator using the simple object access protocol (SOAP) [25] over HTTP respectively HTTPS. The whole set of the SOAP calls are stored in the web service description language (WSDL) file for the verification service. The client application passes the SOAP document including the name of the command and its parameters such as function name, verification information and authorization credentials. For refinement of the testbench we monitor the behavior using code coverage provided by Gcov [26].

V. Results and Discussion

A. *Testing environment*

We performed two sets of experiments based on two different case studies (cf., Sections V-B and V-C) conducted on a cluster with one Intel® Core™ 2 Quad CPU Q9650 @ 3.00 GHz and two Intel® Core™ 2 Duo CPU E8400 @ 3.00 GHz all with 8GB RAM and Linux OS. The first set of experiments represents the results using the state-of-the-art formal verification tools CBMC [6] and ESBMC [7]. The second set of experiments shows the verification results of our new scalable and extendable hybrid verification methodology (Verifyr).

B. *Motorola Powerstone Benchmark Suite*

For our first case study we used Motorola's Powerstone Benchmark Suite [27] and tried to verify the build-in properties (e.g., division-by-zero) from CBMC and ESBMC. We evaluated CBMC [6] with SAT and SMT back-ends versus ESBMC [7]. Overall ESBMC shows the best results except for *jpeg.c*, where CBMC (SAT) outperforms ESBMC. Therefore, we decided to support more than one SMC. As a next step we focused our interests on Modem Encoding/Decoding (*v42.c*). In total, the whole code comprises approximately 2,700 lines of C code and 12 functions. Again, we tried to verify the build-in properties (e.g., division by zero, array out of bounds) from CBMC and ESBMC. It was not possible to verify the whole program using one of the above mentioned SMCs with a *unwinding* parameter (bound) bigger than 4. Therefore, we decided to use this example as a case study for our new approach. The function call graph (FCG) (see Fig. 4) is used as start point for our bottom-up approach/exploration. For every function we used a different instance of CBMC or ESBMC in parallel. It was possible to verify the marked nodes (see Fig. 4) using the SMCs. For all other nodes we faced problems, such as memory limit reached or time out exceeded. Based on this bottom-up analysis, we switched to our top-

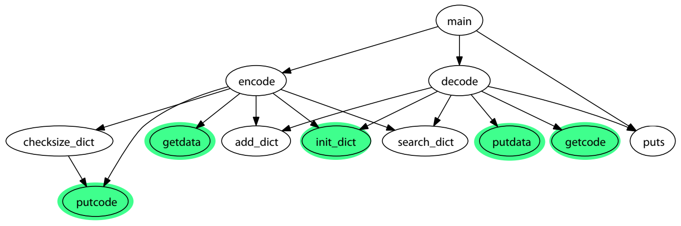

Fig. 4. Function call graph after bottom-up approach for Modem Encoding/Decoding (v42.c). (Marked functions mean successfully verified)

down verification phase triggered by the simulation tool. At every entry point (POI), SCTC exchanges the actual variable assignment with the orchestrator, which uses this information to create temporary versions of the source code of the function under test with static assigned variables. Table I shows the comparison between CBMC (SAT), ESBMC and our Verifyr platform. The used symbols are P (passed), F (failed), MO (out of memory), TO (time out, 90 minutes) and PH (passed using hybrid methodology). PH means that it was possible to verify this function with our hybrid methodology using simulation to support formal verification with static parameter assignment. All tested properties were safe. This table shows that Verifyr presented the same valid results as CBMC (SAT) and ESBMC, and no MO or TO has occurred. Furthermore, the table presents the verification time in seconds in order to reach P, MO or PH results. The time for PH consist of the time for the simulation runs plus formal verification using static parameter assignment. We have used 1000 simulation runs. Overall, we have simulated the whole modem encoding/decoding software using our automatically generated testbench and beyond that we are able to verify 6 out of 12 observed functions using formal verification and the 6 remaining with hybrid verification. However, Verifyr outperforms the single state-of-the-art tools in complex cases where they are not capable to reach a final verification result.

C. EEPROM emulation software from NEC Electronics

Our second case study is an automotive EEPROM Emulation software from NEC Electronics [28], which emulates the read and write requests to a non-volatile memory. This embedded software contains both hardware-independent and hardware-dependent layers. Therefore, this system is a suitable automotive industrial application to evaluate the developed methodologies with respect to both abstraction layers. The EEPROM emulation software uses a layered approach divided into two parts: the Data Flash Access layer (DFALib) and the EEPROM Emulation layer (EEELib). The Data Flash Access layer is a hardware-dependent software layer that provides an easy-to-use interface for the FLASH hardware. The EEPROM Emulation layer is a hardware independent software layer and provides a set of higher level operations for the application level. These operations include: *Format*, *Prepare*, *Read*, *Write*, *Refresh*, *Startup1* and *Startup2*. In total, the whole EEPROM emulation code comprises approximately 8,500 lines of C code and 81 functions. We extracted from the NEC specification manual two property sets (LTL standard). One for EEELib and one for DFALib. Each property in the EEELib set describes the basic functionality on each EEELibs operation (i.e., read, write, etc.). A sample of our LTL properties is as follows:

$$\mathsf{F}\ (\text{Read}\ \rightarrow \text{X F (EEE\_OK} \mid\mid \ldots))\quad \text{(A)}$$

The property represents the calling operations in the EEELib library (e.g., Read) and several return values (e.g., EEE_OK) that may be received. For CBMC we translated the LTL properties to assert/assume style properties based on [22]. We have selected for both EEELib and DFALib (hardware-dependent) two leaf functions and two corresponding parent functions in relation to the corresponding FCG. We have

Function	**CBMC (SAT)**		**ESBMC**		**Verifyr**	
	result	time	result	time	result	time
Leaves						
putcode	P	2s	P	2s	P	2s
getdata	P	2s	P	2s	P	2s
add_dict	MO	135s	MO	155s	PH	535s
init_dict	MO	152s	P	40s	P	40s
search_dict	MO	161s	MO	234s	PH	535s
putdata	P	1s	P	1s	P	1s
getcode	P	1s	P	1s	P	1s
puts	MO	163s	MO	134s	PH	535s
Parents Level 1						
checksize_dict	TO		TO		PH	535s
encode	MO	354s	MO	289s	PH	2s
decode	P	1s	P	1s	P	1s
ALL						
main	MO	351s	MO	274s	PH	535s

TABLE I
VERIFICATION BOTTOM-UP MODEM ENCODING/DECODING (V42.C) POWERSTONE

Function	**CBMC (SAT)**		**ESBMC**		**Verifyr**	
	result	time	result	time	result	time
EEELib						
Eee_Leaf01	P	1s	P	1s	P	1s
Eee_Leaf02	P	1s	P	1s	P	1s
Eee_Parent01	MO	231s	MO	174s	PH	1840s
Eee_Parent02	MO	110s	MO	119s	PH	1840s
DFALib						
DFA_Leaf01	P	1s	P	1s	P	1s
DFA_Leaf02	MO	109s	MO	90s	PH	1840s
DFA_Parent01	MO	112s	MO	92s	PH	1840s
DFA_Parent02	MO	125s	MO	100s	PH	1840s

TABLE II
VERIFICATION RESULTS NEC.

renamed the selected functions for convenience. Table I shows that Verifyr presented the same valid results as CBMC (SAT) and ESBMC, and no MO or TO has occurred. All tested properties were safe. Overall, when we look at the results, we have simulated the whole NEC software using our generated testbench and beyond that we were able to verify 3 out of 8 observed functions using formal verification and the remaining using hybrid verification. Verifyr outperforms the state-of-the-art tools in this complex application where they are not able to reach a final verification result for all functions.

VI. Conclusion and Future Work

We have presented our scalable and extendable hybrid verification approach for embedded software. We have described our new bottom-up/top-down verification methodology and have pointed out the advantages of this approach. It is possible to use different strategies for the whole or parts of the verification process. We start with the formal phase and end up with hybrid verification based on simulation and formal verification. During the bottom-up/exploration phase formal verification tries to verify all possible functions under test based on a FCG until a time bound or memory limit has been reached. The FCG is marked to indicate the Points-of-Interest. Then, we start with simulation and whenever one of these POI is reached, the orchestrator generates a temporary version of the function under test with initialized/pre-defined variables in order to shrink the state space of the formal verification. Our results show that the whole approach is best suited for complex embedded C software with hardware dependencies. It scales better than stand-alone software model checkers and reaches deep state spaces. Furthermore, our approach can be easily integrated in a complex software development process. Currently, we are working on a new hybrid coverage metric to estimate the quality of the hybrid verification and to show, that it boosts the state space coverage.

Acknowledgment

This work has been partially funded by German Research Council (DFG) within project RO1030/16-1 and by the BMBF project SANITAS under grant 01M3088A. We would like to thank NEC Electronic (Europe) GmbH for providing an example component of ESW. The authors also would like to thank Alexander Grünhage, Edgar Auerswald and Patrick Köcher for supporting the development of the Verifyr platform.

References

[1] A. A. Jerraya, S. Yoo, D. Verkest, and N. Wehn, *Embedded Software for SoC*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[2] W. Ecker, W. Mueller, and R. Doemer, *Hardware-dependent Software: Principles and Practice*. Springer Publishing Company, Incorporated, 2009.

[3] T. Kropf, "Software Bugs Seen from an Industrial Perspective or Can Formal Methods Help on Automotive Software Development?" in *CAV'07: Proceedings of the 19th international conference on Computer aided verification*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 3–3.

[4] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast: Applications to software engineering," *INT. J. SOFTW. TOOLS TECHNOL. TRANSFER*, 2007.

[5] R. J. Weiss, J. Ruf, T. Kropf, and W. Rosenstiel, "Efficient and customizable integration of temporal properties into SystemC," in *FDL*, 2005.

[6] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *In Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 168–176.

[7] L. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ansi-c software," in *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 137–148.

[8] T. A. Henzinger, R. Jhala, and R. Majumdar, "The BLAST software verification system," *Model Checking Software*, vol. 3639, pp. 25–26, 2005.

[9] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *TACAS*, vol. 3440. Springer Verlag, 2005, pp. 570–574.

[10] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, *Satisfiability Modulo Theories*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185, ch. 26, pp. 825–885.

[11] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 1, pp. 69–83, 2009.

[12] D. Lettnin, P. K. Nalla, J. Ruf, T. Kropf, W. Rosenstiel, T. Kirsten, V. Schönknecht, and S. Reitemeyer, "Verification of temporal properties in automotive embedded software," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA: ACM, 2008, pp. 164–169.

[13] S. Gorai, S. Biswas, L. Bhatia, P. Tiwari, and R. S. Mitra, "Directed-simulation assisted formal verification of serial protocol and bridge," in *DAC '06: Proceedings of the 43rd annual Design Automation Conference*. New York, NY, USA: ACM, 2006, pp. 731–736.

[14] K. Nanshi and F. Somenzi, "Guiding simulation with increasingly refined abstract traces," in *DAC '06: Proceedings of the 43rd annual Design Automation Conference*. New York, NY, USA: ACM, 2006, pp. 737–742.

[15] S. A. Edwards, T. Ma, and R. Damiano, "Using a hardware model checker to verify software," in *In Proc. of the 4th International Conference on ASIC (ASICON*, 2001.

[16] D. Lettnin, P. K. Nalla, J. Behrend, J. Ruf, J. Gerlach, T. Kropf, W. Rosenstiel, V. Schönknecht, and S. Reitemeyer, "Semiformal verification of temporal properties in automotive hardware dependent software," in *DATE*, 2009, pp. 1214–1217.

[17] J. Ruf, P. M. Peranandam, T. Kropf, and W. Rosenstiel, "Bounded property checking with symbolic simulation," in *FDL*, 2003.

[18] L. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva, "Semiformal verification of embedded software in medical devices considering stringent hardware constraints," *Embedded Software and Systems, Second International Conference on*, vol. 0, pp. 396–403, 2009.

[19] MISRA, "MISRA - The Motor Industry Software Reliability Association," 2000. [Online]. Available: http://www.misra.org.uk/

[20] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Computational Complexity*, 2002, pp. 213–228.

[21] R. Shea, *Call Graph Visualization for C and TinyOS programs*, Dept. of Computer Science School of Engineering UCLA, 2009. [Online]. Available: http://www.ambleramble.org/callgraph/index.html

[22] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of C and verilog programs using bounded model checking," in *DAC '03: Proceedings of the 40th annual Design Automation Conference*. New York, NY, USA: ACM, 2003, pp. 368–371.

[23] E. Clarke, O. Grumberg, and K. Hamaguchi, "Another look at LTL model checking," in *Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, D. L. Dill, Ed., vol. 818. Stanford, California, USA: Springer-Verlag, June 1994, pp. 415–427.

[24] Open SystemC Initiative, "SystemC Verification Standard Library 1.0p Users Manual," 2003.

[25] D. e. a. Box, "Simple Object Access Protocol (SOAP) 1.1," World Wide Web Consortium (W3C), Tech. Rep., 2000. [Online]. Available: http://www.w3.org/TR/soap/

[26] GNU, "Gcov coverage," 2010. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[27] "The M'CORE(TM) M340 Unified Cache Architecture," in *ICCD '00: Proceedings of the 2000 IEEE International Conference on Computer Design*. Washington, DC, USA: IEEE Computer Society, 2000, p. 577.

[28] NEC, "NEC Electronics (Europe) GmbH," http://www.eu.necel.com/.