# Semiformal Verification of Temporal Properties in Automotive Hardware Dependent Software

Djones Lettnin*, Pradeep K. Nalla†, Jörg Behrend,
Jürgen Ruf, Joachim Gerlach, Thomas Kropf and Wolfgang Rosenstiel
University of Tübingen
Department of Computer Engineering - Sand 13, 72076 Tübingen - Germany
E-mail: {lettnin,nalla,behrend,ruf,gerlach,kropf,rosenstiel}@informatik.uni-tuebingen.de

Volker Schönknecht and Stephan Reitemeyer
NEC Electronics (Europe) GmbH, Arcadiastrasse 10, 40472 Düsseldorf - Germany

## Abstract

*The verification of embedded software has become an important subject over the last years. This work presents a new semiformal verification approach called SofTPaDS. It combines assertion-based and symbolic simulation approaches for the verification of embedded software with hardware dependencies. SofTPaDS shows to be more efficient than the software model checkers in order to trace deep state spaces and improves the state coverage relative to a simulation-based verification tool. We have successfully applied our approach to an industrial automotive embedded software.*

## 1 Introduction

Automotive embedded software is responsible for controlling the main car functions. It plays a key role in order to overcome the time-to-market pressure and to provide new functionalities, like reduction of fuel emissions, improvement of security and comfort. 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., non-volatile memory). 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 both simulation and formal verification (FV) approaches. Testing, co-debug and/or co-simulation approaches result in a high effort to create test vectors. Furthermore, critical corner case scenarios might be left unnoticed, that is, they have traditionally the coverage problem.

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. However, it is not suitable for ESW, which has no timing reference and contains more complex data structures (e.g., integers, pointers). Therefore, we need a new mechanism to apply an assertion-based methodology to embedded software.

In order to verify temporal properties in ESW, formal verification techniques are efficient, but only up to medium sized software systems. For larger software designs, formal verification using model checking often suffers from the state space explosion problem. Therefore, abstraction techniques (e.g., predicate abstraction [1]) are applied to alleviate the burden for the back-end model checker. However, verification of temporal properties in the realm of hardware dependent software is still a concern.

In this paper we present a new hybrid verification approach to combine both assertion-based verification and formal verification, called SofTPaDS (Semiformal Verification of Temporal Properties in Hardware Dependent Software). Formal verification verifies static and control-flow operations, on the other hand, assertion-based verification verifies dynamic and data-flow operations. This hybrid approach allows to go deeper into the system (compared to software model checkers) and cover exhaustively local regions of state space (compared to simulation). This combination enables the verification of large industrial hardware dependent software. Our approach employs the SystemC Temporal Checker (SCTC) [10] and the symbolic bounded property checker (SymC) [8]. This whole approach is best suited for fast falsification and covers additionally larger state spaces.

In the remainder of this paper, section 2 presents briefly semiformal verification approaches. Section 3 covers the combination of simulation and formal verification. Section 3.1 summarizes the software modeling. Section 4 gives the experimental results. Section 5 concludes this work.

*CNPq scholarship holder, Brazil.

†This work has in part been funded by the German Research Council (DFG) within project KOMFORT.

978-3-9810801-5-5/DATE09 © 2009 EDAA

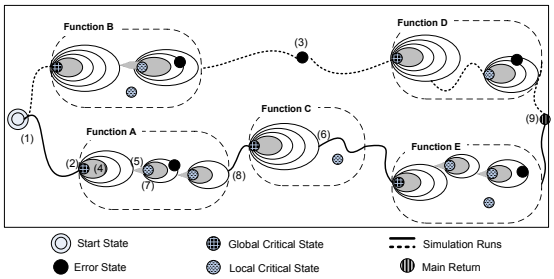


**Figure 1. SofTPaDS overview**

## 2 Related Work

In our previous work [5] only assertion-based verification (i.e., simulation) has been applied to verify temporal properties for hardware independent software, where coverage limitations could be identified. However, in hardware dependent software, simulation has to consider even more test cases for the hardware-software interface variables (e.g., the 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 verification can be used to overcome the drawbacks of both simulation (i.e., coverage) and formal verification (i.e., state space). However, to the best of our knowledge, semiformal verification approaches have been applied only to hardware verification [4, 6], but not to the area of embedded software using C language. Furthermore, the application of a current semiformal hardware model checker to verify embedded software is not viable for large industrial programs [3].

## 3 SofTPaDS Verification Approach

Fig. 1 shows our semiformal verification approach. We start a simulation run with the assertion-based (SCTC) approach (Fig. 1.(1)), which requires one simulation global model, one or many global properties to be checked and one or many global critical states (GCS). SCTC is responsible for finding GCSs (Fig. 1.(2)) and also error states (Fig. 1.(3)). A GCS indicates which function should be chosen in order to generate formal local models on demand for the symbolic simulation. The on-demand generation of the formal models aims to avoid the state space explosion due to the complexity of the global model. GCSs are basically the initial states of local functions (Fig. 1.(2)), which contain the state variables of the global properties. These critical states are automatically generated based on the state variables of the global properties. Therefore, the global properties are used as a guiding mechanism in order to determine which function should be verified at the formal verification phase.

As soon as we find the global critical states, we store these states and generate, for the first time, a local formal model on demand of a *Function X*. The transfer process from simulation to formal verification occurs only for the state variables that were updated during the simulation phase. This heuristic avoids the over-constraining of the state space in formal verification. A container stores the names of updated variables during the simulation and this container signalizes which variables should be sent to the formal verification during the transfer process. As a result, symbolic simulation has not only a unique starting state (as usual in simulation), but an initial state set (Fig. 1.(4)) which will improve the state space coverage of the semiformal verification.

During the symbolic simulation, the tool SymC verifies the system exhaustively until (1) a dynamic operation (e.g., dynamic allocation) or a data-flow arithmetic operation (e.g., multiplication and division) is found (Fig. 1.(5)) or (2) the size of the current state set reaches the threshold limit (Fig. 1.6). At this point, SymC selects one minterm and stores it onto the disk and allows the simulation phase to continue until it reaches a local critical state (Fig. 1.(7)). Local critical states are defined as a number $n$ of transition steps (e.g., one time step for the execution of a data-flow operation in the simulation phase). When a local critical state is reached, the aforementioned local model is only updated with respect to the new state. This local interaction between simulation and formal verification will continue iteratively until one of the engines finds the *return* operation of a *Function X* (Fig. 1.(8)) or all the properties have been evaluated in the local function.

When the simulation run reaches the *return* operation of the *main* function, a new simulation run is started. The global interaction between simulation and formal verification will continue until all the properties were evaluated or, a time bound or maximum number of simulation runs is reached (Fig. 1.(9)).

In the next section, the modeling of embedded software is presented.

### 3.1 Software Modeling

In order to extract simulation and formal models we use four main tools in the front-end: 1) **CIL** is used to convert the C program (compatible with MISRA) into three-address code (3-AC)(Fig. 2.b). 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; 2) The conversion of the original C program into three-address format by CIL transforms arrays and structure pointers into indirect memory access (Fig. 2.c). Our **LValue Analysis** tool replaces all indirect memory accesses with direct memory accesses; 3) The **BLAST** [1] front-end (Fig. 2.d) is used to compute the flow-insensitive pointer analysis and to generate a control flow automata (CFA) for every C function. The points-to analysis (i.e., information about what each pointer points to) contains information about single pointers, double pointers and function pointers; 4) Our **Semiformal Model**

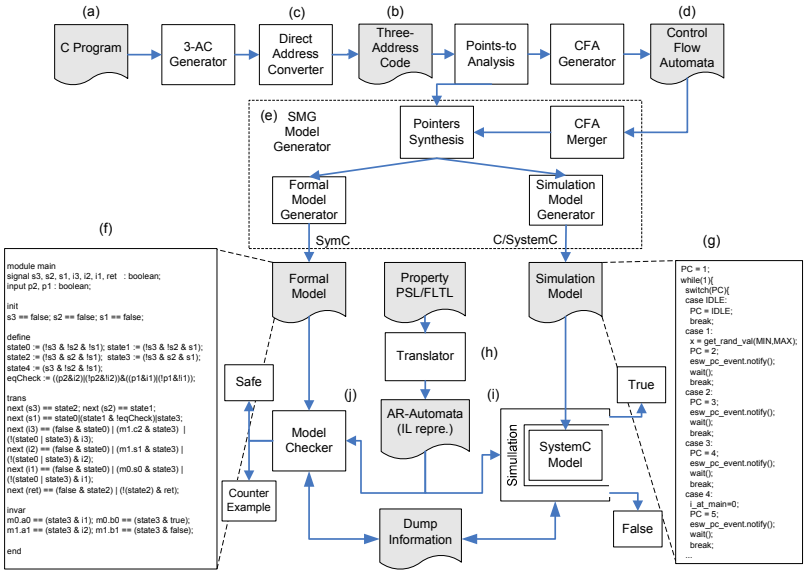


**Figure 2. Semiformal verification approach**

| | $BLAST$ | | $CBMC$ | | $SymC$ | |
|---|---|---|---|---|---|---|
| $Property$ | $V_t(s)$ | $Result$ | $V_t(s)$ | $Result$ | $V_t(s)$ | $Result$ |
| $Read$ | 98.10 | Exception | $> 10800$ | unwind | $> 10800$ | Building BDD |
| $Write$ | 97.09 | Exception | $> 10800$ | unwind | $> 10800$ | Building BDD |
| $Startup1$ | 75.12 | Exception | $> 10800$ | unwind | $> 10800$ | Building BDD |
| $Startup2$ | 43.34 | Exception | $> 10800$ | unwind | $> 10800$ | Building BDD |

**Table 1. State-of-the-art BLAST, CBMC and SymC results**

**Generator** (SMG) tool (Fig. 2.e) generates simulation and formal modeles from the previous analysis.

### 3.2 Semiformal Model Generator

We perform the SMG tool to generate both simulation and formal models. The following issues should be considered in the generation process: **Control Flow Automaton (CFA)**: For the generation of the simulation global model, SMG takes all the CFAs and merges them to a global CFA. During this merging process, SMG maps the local function names with its correspondent global state, in order to keep the locality of local functions. For the generation of the formal local model, SMG reuses the information of the global CFA to generate the local CFA. In both local and global merging processes, all functions are merged only once in order to minimize the state space. In addition, all the local variables become global variables. Finally, a new variable *pc* is added, in order to represent the symbolic transition relation. This variable is also used in the simulation phase to trigger the embedded software monitors.

**Synthesis of Pointers**: SMG performs the synthesis of C pointers based on the analysis of BLAST and on multiplexed expressions [9].

**Simulation Model Generation**: The simulation and the formal model (Fig. 2.g) are based on the same CFA. Additionally, complex structures, which are not suitable to be modeled in the formal model such as *data-flow arithmetic operation* (e.g., multiplication and division), are added to the simulation model.

**Formal Model Generation**: The formal model that we provide as an input to SymC works only with Boolean variables and consists of five basic blocks: *declaration*, *init*, *define*, *trans* and *invariant* (Fig. 2.f). The aforementioned simulative operations are considered as local critical states for the formal verification. In the formal model, these operations are modeled by simulation states via flag variables. Whenever one of these flag states is reached, the simulation engine is called.

**State Variables**: The arithmetic (e.g., addition and subtraction) and relational expressions are modeled using adders. We allocate a vector with $n$ latches, where $n$ can be 8, 16 and 32 bits for both simulation and formal models. The *pc* variable needs $log_2 m$ latches to represent the whole transition relation. The input variables are assigned to state variables at the initialization phase to keep the range of values constant during the symbolic simulation.

## 4 Automotive Case Study

Our case study is an EEPROM Emulation software from NEC Electronics [7]. It uses a layered approach splitted into two parts: the Data Flash Access layer (DFALib) and the EEPROM Emulation layer (EEELib). The EEPROM Emulation layer provides a set of higher level operations for the application layer. Some main operations are: `read`, `write`, `startup1` and `startup2`. Due to the close connection to the hardware, the DFALib is responsible for configuring the hardware with respect to the high level operation. In total, the EEPROM Emulation code comprises approximately 8,500 lines of C code and 81 functions.

| | $SofTPaDS$ | | | | | | $SCTC$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $Property$ | $SR$[1] | $S_{TV}$[2] | $SF_{Mem}(MB)$[3] | $SFx$[4] | $V_t(s)$[5] | $C_P(\%)$[6] | $SR$[1] | $TV$[2] | $V_t(s)$[5] | $Mem(MB)$[3] | $C_P(\%)$[6] |
| $Read$ | 16 | 1435 | 67.70 | 2 | 172.97 | 100 | 1000 | 87026 | 0.90 | 0.32 | 10 |
| $Write$ | 788 | 68456 | 66.89 | 2 | 109.06 | 100 | 1000 | 86740 | 0.73 | 0.32 | 10 |
| $Startup1$ | 87 | 7478 | 67.10 | 2 | 316.51 | 100 | 1000 | 86318 | 1.94 | 0.33 | 10 |
| $Startup2$ | 17 | 1438 | 59.46 | 2 | 84.42 | 100 | 1000 | 90339 | 2.22 | 0.32 | 40 |

[1] Simulation run [2] Test vectors [3] Consumed memory [4] Simulation-Formal interaction [5] Verification time [6] Property coverage

**Table 2. New SofTPaDS approach and state-of-the-art SCTC results**

### 4.1 Preliminaries

We performed two sets of experiments conducted on an Intel Pentium dual core 3Ghz, 4GB RAM with Linux OS. The first set of experiments represents the results using the state-of-the-art formal verification tools BLAST, CBMC [2] and SymC (table 1) and, state-of-the-art simulation-based verification tool SCTC (table 2). The second set of experiments show the verification results of our new semiformal verification SofTPaDS (table 2).

We extracted our property set (FLTL standard) from the NEC specification manual. Each property in this set describes the basic functionality on DFALib's operation. A sample of our FLTL properties is: $(FOp \rightarrow XG!(HW_{Reg} == Val_X))$. This safety property defines when a specific EEELib operation (e.g., startup1) is called from the application layer, the DFALib lower layer should not assign an illegal setting value to a hardware register. In total, 40 properties were evaluated, which corresponds to 10 properties for each of the four EEELib operations. In our previous work [5] only properties at EEELib layer were evaluated, which is a hardware independent software layer. Therefore, we re-evaluate the performance of the state-of-the-art verification tools.

### 4.2 Results and Discussion

As we can observe the results in table 1, BLAST faces abort exceptions for all the properties and we were not able to finish the verification process. We surmise the abort exceptions result from the theorem prover. CBMC spent for all the properties more than 3 hours in unwinding C loops. Therefore, we always faced time limit problems. In our experiments we used the limit of 20 for unwinding loops. The property checker SymC suffered also in the preprocessing phase during the building of the BDDs, and therefore, we also faced time limit problems.

In table 2 column SCTC, presents the results of the SCTC simulation-based verification. We applied the maximum of 1000 simulation runs, which results to less than 1 MB consumed memory, up to 3 seconds verification time and to more than 85,000 test vectors. However, SCTC could only achieve up to 40% of property coverage.

In table 2 column SofTPaDS, we present the results of our new semiformal SofTPaDS approach. Compared to the state-of-the-art model checkers in table 1, SofTPaDS resulted neither in abort exceptions nor in timeout problems and could verify all the properties. SofTPaDS has performed two interactions between simulation and formal verification, consumed up to 68 MB of memory and achieved the results in up to 317 seconds. Compared to the state-of-the-art simulation-based SCTC in table 2, SofTPaDS was able to achieve 100% property coverage with less simulation runs and less application of test vectors. Therefore, SofTPaDS enables us to go deeper into the system, whereas state-of-the-art model checkers suffer from exceptions or timeouts. SofTPaDS improves the property coverage up to nine times relative to the simulation-based verification tool.

## 5 Conclusion and Future Work

We presented a new methodology to combine both simulation and formal verification in order to verify temporal properties in hardware dependent software. Both techniques complement each other to allow a fast falsification of large embedded software designs. SofTPaDS achieved two important goals: (1) improvement of the state space coverage up to nine times compared to simulation-based verification and (2) verification of deeper states in hardware dependent software, which could not be achieved with the formal verification tools. As future work, we are working on analysis of semiformal counterexamples and on guiding approaches to advice the simulation engine in search of critical states.

## References


[1] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *In STTT'07*.

[2] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.

[3] S. A. Edwards, T. Ma, and R. Damiano. Using a hardware model checker to verify software. In *ASICON*, 2001.

[4] S. Gorai, S. Biswas, L. Bhatia, P. Tiwari, and R. S. Mitra. Directed-simulation assisted formal verification of serial protocol and bridge. In *DAC*, 2006.

[5] D. Lettnin, P. Nalla, J. Ruf, T. Kropf, W. Rosenstiel, T. Kirsten, V. Schönknecht, and S. Reitemeyer. Verification of temporal properties in automotive embedded software. In *DATE*, 2008.

[6] K. Nanshi and F. Somenzi. Guiding simulation with increasingly refined abstract traces. In *DAC*, 2006.

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

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

[9] L. Séméria and G. D. Micheli. SpC: synthesis of pointers in C: application of pointer analysis to the behavioral synthesis from C. In *ICCAD*, 1998.

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