Verification of Temporal Properties in Automotive Embedded Software

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

Tobias Kirsten, Volker Schönknecht and Stephan Reitemeyer
NEC Electronics (Europe) GmbH
Arcadiastrasse 10, 40472 Düsseldorf - Germany
E-mail: {Tobias.Kirsten,Volker.Schoenknecht,Stephan.Reitemeyer}@eu.necel.com

Abstract

The amount of software in embedded systems has increased significantly over the last years and, therefore, the verification of embedded software is of fundamental importance. One of the main problems in embedded software is to verify variables and functions based on temporal properties. Formal property verification using model checker often suffers from the state space explosion problem when a large software design is considered. In this paper, we propose two new approaches to integrate assertions in the verification of embedded software using simulation-based verification. Firstly, we extended a SystemC hardware temporal checker with interfaces in order to monitor the embedded software variables and functions that are stored in a microprocessor memory model. Secondly, we derived a SystemC model from the original C program in order to integrate directly with the SystemC temporal checker. We performed a case study on an embedded software from automotive industry which is responsible for controlling read and write requests to a non-volatile memory.

1 Introduction

Almost all the classical car functions are being controlled by microprocessor elements. Embedded software (ESW) plays a key role in order to overcome the time-to-market pressure and to provide new functionalities, like reduction of gas emissions and improvement of security and comfort. Finite-state machine errors, timing errors, stack/memory overflow errors and non-volatile memory errors are some examples of severe coding errors and, therefore, the verification of embedded software is of fundamental importance.

Assertion-based verification methodology captures a design's intended behavior in temporal properties and monitors the properties during system simulation [9]. However, this methodology has been successfully used at lower levels of hardware designs, especially at register transfer level (RTL), which requires a clock mechanism as timing reference and signals at the Boolean level. Thus, it is not suitable to apply this hardware verification technique directly on embedded software, which has no timing reference and contains more complex structures (e.g. integers, pointers, etc.). Therefore, we need a new mechanism in order to apply assertion-based methodology on embedded software.

The most commonly used approaches to verify embedded software are based on both simulation and formal approaches. Directed test approaches possibly taking advantage of co-debug and/or co-simulation solutions. This results in a high effort to create test vectors and critical corner case scenarios might go unnoticed. Furthermore, one of the main problems in embedded software verification is monitoring variables and functions based on temporal properties.

In order to verify temporal properties in embedded software, formal method techniques are efficient, but only for medium sized software systems, where they have less state space to explore. For larger software designs, formal verification using model checker often suffers from the state space explosion problem. Therefore, we need to use abstraction techniques in order to alleviate the burden for the back-end model checker. For example BLAST [6] checks the software based on predicate abstraction. It verifies temporal safety properties of C programs via a specification language (SpC) [3]. However, for complex properties it is as laborious task as programming to describe the properties

*CNPq scholarship holder, Brazil.
†This work has been funded by the BMBF and edacentrum within project FEST (01M3072).

978-3-9810801-3-1/DATE08 © 2008 EDAA

using SpC. In addition, we need to introduce new global variables that debilitate the strength of the model checker. Therefore, verification of a temporal properties in the realm of embedded software is still a concern.

In this paper, we used two new simulation-based approaches to tackle these problems. At first we integrated temporal properties into a microprocessor via SystemC Temporal Checker (SCTC) to perform verification under real-time conditions. Secondly, we derive the SystemC model from the original C program and later this model is verified using SCTC. SystemC [5] is used to support modeling of both hardware and software components.

The remainder of this paper is organized as follows. Section 2 presents briefly the state-of-the-art in the embedded software verification field. Section 3 covers the combining of temporal properties and embedded software. Section 4 gives the experimental results and discussion. Section 5 concludes and describes briefly the future work.

2 Related Work

There are several works in the simulation and debugging area, which are specific to a given platform or prototyping environment [12] or specific for a processor family [2]. Some works were published using emulators to speed up the simulation of cycle accurate microprocessor models [10]. Hardware coverage driven verification has been extended to perform hardware/software co-verification [17]. The SystemC Verification Library [7] provides no means of embedded software verification and does not contain a mechanism for specifying and checking temporal properties.

However, these verification approaches do not contain mechanisms to enable the verification of temporal properties of both functions and variables in the embedded software side.

2.1 Formal Software Verification

Often used software (i.e. C program) model checking approaches are: a) Bounded Model Checking (BMC) [4]; b) Model checking with predicate abstraction using a theorem prover or a SAT-solver [6, 16, 14]; c) Convert the C program to a model and feed into a model checker [8].

The work in [8] focuses on model checking C programs. They model the semantics of C programs as finite state systems by using suitable abstractions. Later these abstract models are verified using both BDD-based and SAT-based model checkers. CBMC [4] performs the formal verification of full ANSI-C programs using BMC. However, the tool has restrictions with upper time bound. Due to the boundedness CBMC can be used for finding errors and not for proving correctness. BLAST [6] checks software based on an *abstract-check-refine* paradigm. It constructs an abstract model based on predicates, then checks the safety property, and if the check fails, refines the model and iterates the whole process. Therefore, each model checker has its own strengths and weaknesses. A detailed survey on software model checking approaches is made in [15].

2.2 Contributions

Our main contributions in this paper are two new approaches to integrate temporal assertions in the verification of embedded software. Firstly, we have extended a SystemC temporal checker with new interfaces in order to monitor the embedded software variables and functions that are stored in a SystemC microprocessor memory model. Secondly, we derive a SystemC model from the original C program to integrate directly with the SystemC temporal checker. To the best of our knowledge there is no previous work related to the temporal property verification of both embedded software variables and functions in the realm of simulation-based verification.

3 Temporal Checker Framework

The C language does not support any means to check temporal properties in software modules. Therefore, we use the existing SCTC, which is a hardware oriented temporal checker. SCTC supports specification of properties either in PSL (Property Specification Language) [1] or FLTL (Finite Linear time Temporal Logic) [13], an extension to LTL with time bounds on the temporal operators.

Temporal logics are used to describe sequences of states in reactive systems. A formula is satisfied if a path in the system corresponds to the sequence of states the formula represents.

SCTC has a synthesis engine which converts the plain text property specification into a format that can be executed during system monitoring. We translate the property to Accept-Reject automata (AR-automata) [13] in the form of Intermediate Language (IL) and later to a monitor in SystemC. The AR-automata can detect validation (i.e., `True`) or violation (i.e., `False`) of properties on finite system traces, or they stay in pending state if no decision can be made yet.

SCTC can also check properties which include complex structures using a base class `Proposition`. This class allows wrapping arbitrary source code entities as named objects. Fig. 1 lists the interface of class `Proposition`. The virtual member function `is_true` (line 4) has to

```
class Proposition {                                          1
public:                                                      2
  // A proposition must evaluate to either true or false.    3
  virtual bool is_true() = 0;                                4
  bool is_false() { return !is_true(); }                     5
  // Create clone of the current proposition.                6
  virtual Proposition* clone() = 0;                          7
  // Ensure proper destruction with virtual destructor.      8
  virtual ~Proposition() { }                                 9
};                                                           10
```

Figure 1. Proposition class interface

be provided by any subclass of `Proposition`. The checker evaluates these functions in order to get the current system states. The return value of this function is connected with the Boolean layer of the temporal property. These atomic entities constitute the predicates in the temporal logic formulas. Typically, propositions are stateless. However, for more advanced predicates, they can carry state.

The existing SCTC does not support a mechanism to monitor the variables and functions of embedded software. Therefore we need to extend the SCTC.

3.1 1st Approach: Verification using Microprocessor Model

In embedded software, SCTC needs to communicate with the software running on the processor. The use of a microprocessor model enables us to verify real operating conditions of the embedded software. The architecture of this extension can be seen in Fig. 2. SCTC needs a SystemC microprocessor model and an interface to the main memory (e.g. *sc_uint <32> sctc_sc_read_uint (sc_uint <32> addr)*). With this interface we can provide the ESW variable address and read its content from memory. The verification of the temporal properties in ESW should follow some steps: a) Determine the program variables that should be verified. b) Determine the addresses of the variables, which are located in the embedded memory. c) For all functions, add the assignment `fName=FUNCTION_NAME`. This allows us to monitor function sequels through a variable. Thus, the function names can be also used in the property specification. d) Create an `ESW_monitor` module in order to wrap the SCTC in the SystemC microprocessor design. This module will handle the handshake protocol between the ESW and the SCTC. e) Create the ESW `Propositions` that should be used for temporal properties. f) Instantiate temporal properties in the `ESW_monitor` module.

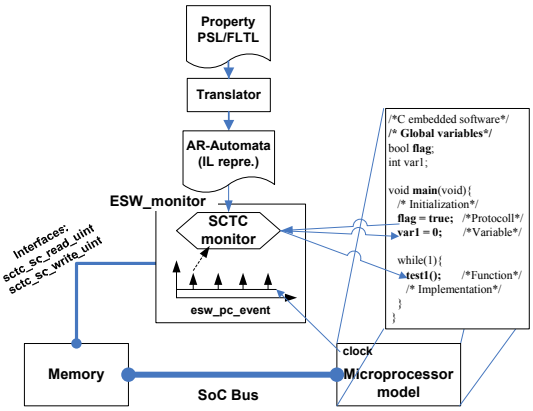

Figure 2. Verification of temporal properties in C program

```
void ESW_monitor :: esw_monitor(){                           1
  define_clock_asTrigger();                                  2
  while !initialized                                         3
    initialized =                                            4
      readfromMemory(flag_address);                          5
    register_ThePropositions();                              6
    instantiate_TheTemporalProperties();                     7
  forever                                                    8
    monitor_TheTemporalProperties();                         9
}                                                            10
```

Figure 3. Protocoll between SCTC and embedded sofware.

We also need a timing reference in order to trigger the temporal properties during the simulation. Using the microprocessor clock as our timing reference enables us to verify the temporal properties in real-time conditions (see line 2, Fig. 3). When the SCTC needs to make a call to a function in the embedded software, it first needs to check that the software is active and has been initialized. This can be done by reading the status of the *flag* variable in the software (lines 3-5). When the ESW is initialized, we register the propositions and instantiate the temporal property monitors (lines 6-7). This process occurs only in the initial phase of `ESW_monitor` module. After this initialization phase, temporal properties (i.e. assertions) will be monitored during the simulation.

3.2 2nd Approach: SystemC Model Derivation from Embedded Software

The microprocessor model in the first approach allows real-time temporal properties to be verified. Albeit we perform verification under real-time conditions, more simulation time is consumed due to microprocessor model. In order to speed up the verification process, we propose a second approach where we derive a SystemC model from the

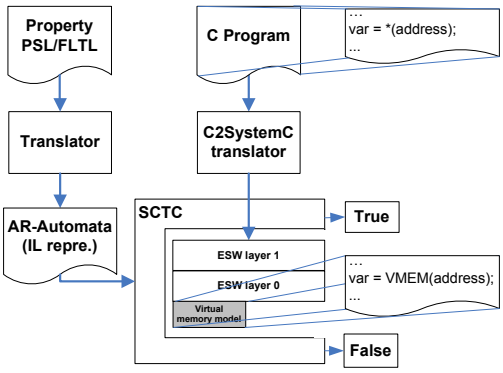

Figure 4. Verification without using microprocessor model

```
1  void C2SystemC_Translator() {
2    create_ESW_SC_class();
3    define_esw_pc_event_asTrigger();
4    create_VirtualMemModel();
5    for all directMemAccessVars
6      convert_DirectMemAccessToVM();
7    for all Cvars
8      define_CvarsToSCmembers();
9    for all Cfunctions
10     define_CfuncsToSCmemberFuncs();
11   for all FunctionBody
12     add fName=FUNCTION_NAME;
13     after every statement
14       add esw_pc_event.notify();
15       add wait();
16 }
```

Figure 5. Derivation of a SystemC model from C program.

embedded software and thereafter apply SCTC. Fig. 4 shows the verification approach without using a microprocessor model.

The algorithm responsible for deriving the SystemC model is presented in Fig. 5. The derived model is as precise as original C program. It consists of one SystemC class (`ESW_SC`) mapped to a corresponding C program. The `main` function in C will be 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 (`esw_pc_event`) event (lines 3 and 13-15). Additionally the `wait();` statement is necessary to suspend the SystemC process. The program counter event will be notified after every statement and will be responsible to trigger the SCTC. It is important to point out that the timing reference is not the same as the absolute time from the microprocessor model (see Section 3.1). This makes a huge difference in length of the AR-automaton if we specify the properties involving fixed time length. Since the prior approach works with absolute time, it needs larger time bounds to be specified in the property in order to execute each statement in the C program. This second approach uses the program counter (`esw_pc_event`) as clock reference, i.e. each statement execution is one time step. Therefore, it needs relatively lower time bounds in AR-automaton if we check the same functionality using both approaches.

The embedded software works close to the hardware, for instance automotive software where we need to access memory frequently. We consider that the verification is performed without having hardware (original microprocessor memory) and in such cases the access should be mapped to a virtual memory model. Thus, all direct memory access (e.g. `*(address)`) should be converted into virtual memory requests. Lines 4-6 in the above algorithm implements these functionalities. Fig. 4 shows the use of the virtual memory for the lower ESW model layer.

Lines 7-10 in the algorithm convert the global variables and functions in our C program into SystemC conventions (i.e. class members and member functions, respectively). As aforementioned in Section 3.1, we need to create a new variable named $fName$ that helps to inspect function sequel properties. The variable will be updated in each function context with the assignment `fName=FUNCTION_NAME`.

In contrast to the first approach, we do not need to implement any protocol in order to initialize the verification. The derived SystemC model and the SCTC are integrated in `sc_main` environment.

4 Automotive Case Study

Our case study is an EEPROM Emulation software from NEC Electronics company [11]. It uses a layered approach towards the EEPROM Emulation. The software is therefore split into two parts: the Data Flash Access layer (DFALib) and the EEPROM Emulation layer (EEELib), as shown in Fig. 6. The Data Flash Access Layer provides an easy to use interface for the flash hardware. The EEPROM Emulation layer provides a set of higher level operations for the application layer. These operations include: `format`, `prepare`, `read`, `write`, `refresh`, `startup1` and `startup2`. The EEELib is a highly state driven program. Each of these operations are defined by a series of machine states that the emulation flow must follow in order to complete the process. In most cases, the states are unique to a procedure. However the `ready`, `abort`, `error` and `finish` states are shared states. In total the whole EEPROM Emulation code comprises approximately 8.500 lines of

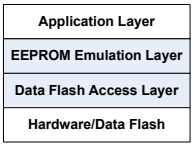

Figure 6. NEC software

C code and 81 functions. The verification goal is to check the correctness of the software with respect to all the operations.

4.1 Results and Discussions

We performed 3 sets of experiments. The first set of experiments presents the results using a state-of-the-art software verification tools. These experiments were conducted on an Intel Pentium machine 2Ghz, 2GB RAM with Linux OS. The second and third sets of experiments show the verification results with and without the microprocessor model. These experiments were conducted on an Intel Pentium machine 3.2Ghz, 2GB RAM with Linux OS.

We extracted our property set (FLTL standard) from the NEC specification manual. Each property in this set describes the basic functionality on each EEELib's operation, (i.e. read, write, etc). A sample of our FLTL properties is as follows:

$$\mathsf{F}\,(\text{Read} \rightarrow \text{X F[b] (EEE\_OK} \,||\, \ldots)) \quad \text{(A)}$$

where $b > 0$ is an explicit time bound. The property represents the calling the operations in the EEELib library (e.g. Read) and the several return values (e.g. EEE_OK) may be received. All the tested properties were safe.

4.1.1 Experience with BLAST and CBMC Model Checkers

First, we checked our software with two state-of-the-art formal software verification tools BLAST and CBMC. To specify complex temporal properties, BLAST uses a specification language (SpC). BLAST faces an integer overflow problem, i.e. when the value of the variable exceeds 1073741823 ($2^{30} - 1$) then the tool could result in either a false positive or false negative. For all the properties we were not able to finish the verification process due to abort exceptions (as shown in Fig. 7), which we surmise resulted from theorem prover. CBMC does not support any mechanism to specify temporal properties. Therefore, we required the use of the Spec tool [3] in order to describe the properties and then a newly generated C file (consisting of the property described in it) is fed into CBMC. For all the properties, CBMC spent more than 5 hours in unwinding C loops. Therefore, we always faced time limit problems. In our experiments we used the limit of 20 for unwinding loops. In addition, all the input variables have to be constrained in order to avoid false reasoning.

	BLAST		CBMC	
Property	V.T.(s)	Result	V.T.(s)	Result
Read	2001	Exception	> 18000	unwind
Write	1115	Exception	> 18000	unwind
Startup1	1358	Exception	> 18000	unwind
Startup2	1428	Exception	> 18000	unwind
Prepare	674	Exception	> 18000	unwind
Refresh	489	Exception	> 18000	unwind
Format	355	Exception	> 18000	unwind

Figure 7. BLAST and CBMC results

4.1.2 Verification with Microprocessor Model

We needed to generate stimuli (constrained randomization) for all the external input variables and hardware (i.e. Data Flash) elements. We used the maximum of 10000 test cases in our experiments. The properties we verified are of the type (A) in the above equation and the results are shown in the first column of Fig. 8. The Verification Time in seconds is shown in the subcolumn `V.T.(s)`. The Test Cases (`T.C.`) subcolumn corresponds to the number of test cases applied during the verification. The Coverage (`C.(%)`) subcolumn describes the percentage of the return values that we received. 100% indicates that we received all the return values. To trigger on each statement in C program requires a large number of system clock cycles and therefore, we did not use any time bound in our properties.

4.1.3 Verification with SystemC ESW Model

We checked our properties with time bound 1000, time bound 10000 and no time bound as shown in the second column of Fig. 8. Properties with no time bound (TB) are pure LTL properties. We used the maximum of 100000 test cases in this set of experiments. For the properties `Read`, `Format` and `Prepare` the increase in time bound resulted in better coverage of the returned values. In some of our experiments, the properties without time bound outperforms the ones with time bound due to the higher number of test cases. The verification time in all our approaches comprises both AR-automaton generation and simulation times. The subcolumn V.T. in column TB-10000 includes large AR-automaton generation time. For all our experiments the second approach took less verification time compared to the first approach. We achieved a speedup of up to 900. This is mainly due to the dropped real-time conditions of the microprocessor model. All our results show that we can verify the properties without having any false positives or false negatives.

	With microprocessor model			SystemC ESW model								
	No-TB			TB-1000			TB-10000			No-TB		
Property	V.T.(s)	T.C.	C.(%)	V.T.(s)	T.C.	C.(%)	V.T.(s)	T.C.	C.(%)	V.T.(s)	T.C.	C.(%)
Read	505	721	100	1.4	47	40	166	467	60	0.57	721	100
Write	3886	10000	33	1.4	45	33	165	437	33	81	100000	33
Startup1	18	10	100	1.4	10	100	165	10	100	0.02	10	100
Startup2	12987	10000	66	1.4	51	66	179	484	66	60	100000	66
Format	3581	10000	66	1.4	43	33	165	417	66	65	100000	66
Prepare	9807	10000	50	1.4	44	25	166	403	50	76	100000	75
Refresh	3604	10000	40	1.4	48	40	166	449	40	74	100000	40

Figure 8. 1st and 2nd approaches results

5 Conclusion and Future Work

Automotive embedded software are often very large and complex. Formal verification often suffers from the state space explosion problem when we intend to verify large embedded software. In this paper, we detailed two new approaches to integrate temporal assertions in the verification of embedded software using simulation based verification: Firstly, we integrated the temporal properties into a SystemC microprocessor model. Secondly, we derived a SystemC model (without performing any abstraction) from the original C programs. The first approach demonstrates the advantage of verifying real-time temporal properties in C program using the microprocessor clock as a timing reference. However, we had an overhead of simulating the microprocessor model. The second approach uses only a SystemC ESW model. Therefore, we achieved shorter verification times. However, we had an overhead of generating a SystemC ESW model. Both approaches are suitable for verifying complex temporal properties and are easy to apply in the industrial design flow. As we can notice in Section 4, neither of new approaches faced exception or false reasonings compared to the state-of-the-art tools. In future, we would like to combine the simulation-based verification and formal verification approach in order to improve the coverage.

References

[1] Accellera. *Property Specification Language (PSL), Version 1.1*, June 2004.

[2] J. Andrews. *Co-Verification of Hardware and Software for Arm Soc Design*. Newnes, 2005.

[3] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *Proceedings of the 11th international Static Analysis Symposium*, pages 26–28. LNCS 3148, Pages 2-18, 2004, 2004.

[4] CMU. CBMC: Bounded Model Checking for ANSI-C. http://www.cs.cmu.edu/~modelcheck/cbmc/.

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

[6] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *10th International Workshop on Model Checking of Software (SPIN), volume 2648 of Lecture Notes in Computer Science*, pages 235–239, 2003.

[7] C. N. Ip and S. Swan. A Tutorial Introduction on the New SystemC Verification Standard. In *White Paper*, 2003.

[8] F. Ivanicic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checking C programs using F-SOFT. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 297–308, Washington, DC, USA, 2005. IEEE Computer Society.

[9] C. N. C. Jr. and H. D. Foster. Assertion-based verification. In R. Drechsler, editor, *Advanced Formal Verification*, pages 167–204. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[10] Y. Nakamura, K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura. A Fast Hardware/Software Co-Verification Method for System-On-a-Chip by Using a C/C++ Simulator and FPGA Emulator with Shared Register Communication. In *Design Automation Conference (DAC)*, pages 299–304, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

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

[12] G. Post, P. Venkataraghavan, T. Ray, and D.R.Seetharaman. A SystemC-based verification methodology for complex wireless software IP. In *DATE'04:Design, Automation and Test in Europe Conference and Exhibition, 2004*, volume 1, pages 544–550. IEEE Computer Society, 2004.

[13] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on a multi-valued AR-automata. In W. Nebel and A. Jerraya, editors, *Design, Automation and Test in Europe 2001*, pages 742–748. IEEE Press, 2001.

[14] SatAbs. SATABS - predicate abstraction with SAT for ANSI-C. http://www.verify.ethz.ch/satabs/.

[15] B. Schlich and S. Kowalewski. Model checking C source code for embedded systems. In *IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation (IEEE/NASA ISoLA 2005)*, pages 65–77. NASA, 2005.

[16] SLAM. Software model checking with SLAM. http://research.microsoft.com/SLAM/.

[17] M. Winterholer. Transaction-based Hardware Software Co-verification. In *FDL'06: Proceedings of the conference on Forum on Specification & Design Languages*, 2006.