Chapter 8
Scalable and Optimized Hybrid Verification of Embedded Software

Jörg Behrend, Djones Lettnin, Alexander Grünhage, Jürgen Ruf, Thomas Kropf and Wolfgang Rosenstiel

8.1 Introduction

Embedded software (ESW) is omnipresent in our daily life. It plays a key role in overcoming the time-to-market pressure and providing new functionalities. Therefore, a high number of users are dependent on its functionality [1]. ESW is often used in safety critical applications (e.g., automotive, medical, avionic), where correctness is of fundamental importance. Thus, verification and validation approaches are an important part of the development process.

The most commonly used approaches to verify embedded software are based on simulation or formal verification (FV). 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. An extension of simulation is the assertion-based verification (ABV) methodology that captures a design's intended

J. Behrend (✉) · A. Grünhage · J. Ruf · T. Kropf · W. Rosenstiel
Department of Computer Engineering, University of Tübingen,
Sand 13, 72076 Tübingen, Germany
e-mail: behrend@informatik.uni-tuebingen.de

A. Grünhage
e-mail: gruenhag@informatik.uni-tuebingen.de

J. Ruf
e-mail: ruf@informatik.uni-tuebingen.de

T. Kropf
e-mail: kropf@informatik.uni-tuebingen.de

W. Rosenstiel
e-mail: rosenstiel@informatik.uni-tuebingen.de

D. Lettnin
Department of Electrical and Electronic Engineering, Federal University of Santa Catarina,
Campus Universitário s/n, Trindade, Florianópolis, SC CEP 88040-900, Brazil
e-mail: djones.lettnin@ufsc.br

D. Lettnin and M. Winterholer (eds.), *Embedded Software Verification and Debugging*, Embedded Systems, DOI 10.1007/978-1-4614-2266-2_8

behavior in temporal properties. 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 designs, formal verification using model checking often suffers from the state space explosion problem. Therefore, abstraction techniques (e.g., predicate abstraction [2]) are applied to reduce the load of the back-end model checker.

Semiformal or hybrid approaches have been proposed many times before with only limited success. In this paper we present VERIFYR [3], an optimized and scalable hybrid verification approach using a semiformal algorithm and taking advantage of automated static parameter assignment (SPA). This technique reduces the model size by assigning a static value to at least one function parameter. Information gained during simulation (dynamic verification) is used to assign values to the parameters in order to reduce the formal model (static verification). One issue is the selection of the best function parameter. This is important due to the different impact of parameters on the resulting state space. Until now, it was a manual or randomized task to assign the parameter values. The selection of the parameter values may influence the program flow and therefore, the resulting model size. This work describes a new approach in order to rank function parameters depending on their impact on the model size. The ranking is based on estimation according to the usage of the parameters in the function body. Finally, SPA can be automatically applied to select parameters in an optimized way in order to reduce the model complexity in a controlled manner.

The paper is organized as follows. Section 8.2 describes the related work. Section 8.3 details the verification methodology and the technical details. Section 8.4 summarizes our case studies and presents the results. Section 8.5 concludes this paper and describes the future work.

8.2 Related Work

Bounded model checking (BMC) is an approach to reduce the model size using bounded execution paths. The key idea is to build a propositional formula, whose models correspond to program traces (with bounded length) that might violate some given property using state-of-the-art SAT and SMT solvers [4]. For instance, C bounded model checker (CBMC) [5–7] has proven to be a successful approach for automatic software analysis. Codeiro et al. [8] have implemented ESBMC based on the front-end of CBMC and a new back-end based on SMT. All the above-mentioned work fail when the bound is not automatically determinable.

The optimization of formal models has been the reason to use abstraction methods. The automatic predicate abstraction [9] introduced a way to construct abstracted models and allowed to introduce automated constraints like loop invariants [10]. Based on abstraction, Clarke et al. developed a refinement technique to generate even smaller models using counterexamples [11, 12]. BLAST [13] and SATABS [14] are formal verification tools for ANSI-C programs. Both tools use predicate abstraction mechanisms to enhance the verification process and to reduce the model size successfully. Semiformal/hybrid verification approaches have been applied successfully to hardware verification [15–17]. However, the application of a current semiformal hardware model checker to verify embedded software is not viable for large industrial programs [18]. In the area of embedded software using C language, Lettnin et al. [19] proposed a semiformal verification approach based on simulation and symbolic model checker (SymC) [20]. However, SymC was the bottleneck for the scalability of the formal verification, since it was originally developed for the verification of hardware designs. Cordeiro et al. [21] have published a semiformal approach to verify medical software, but they have scalability problems caused by the used model checker. The aforementioned related works have their pros and cons. However, they still have scalability limitations in the verification of complex embedded software with or without hardware dependencies.

Concolic testing was first introduced by Godefroid et al. [22] and Cadar et al. [23] independently. Koushik et al. [24] extended this methodology to a hybrid software verification technique mixing symbolic and concrete execution. They treat program variables as symbolic variables along a concrete execution path. Symbolic execution is used to generate new test cases to maximize the code coverage. The main focus is finding bugs, rather than proving program correctness. The resulting tools DART, EXE, and CUTE apply concolic testing to C programs. But concolic testing has problems when very large symbolic representations have to be generated, often resulting in unsolvable problems. Other problems like imprecise symbolic representations or incomplete theorem proving often result in a poor coverage. ULISSE [25] is a tool to support system-level specification testing based on extended finite state machines (EFSM). The KLEE [26] framework compiles the source code to LLVM [27] byte code. The code under test has to be compatible with LLVM and user interaction (which is essential for our verification approach) is not supported. PEX [28] was developed at Microsoft Research to automate structural testing of .NET code but not C code. Frama-C [29–31] is an integrated development environment for C code with focus on static verification only and Frama-C needs special code annotations for the used “design by contract” approach.

Behrend et al. used SPA [3] to reduce the model size during semiformal verification. By assigning a static value to a function parameter the automatic predicate abstraction algorithm can generate a different abstraction that may lead to a smaller model. If a parameter is assigned, the parameter is no longer handled as full range variable, but as statically assigned variable. However, the previous approach the function parameter for SPA was selected manually.

8.2.1 Contributions

Our main contribution in this current work is a novel semiformal approach for the verification of embedded software with temporal properties based on VERIFYR. We provide a new methodology to extract both dynamic and static verifiable models from C programs to perform both assertion-based and formal verification. On the formal side, we are able to extend the formal engine with different state-of-the-art software model checkers (SMC). On the simulation side, simulation models (C or SystemC) and the testbench environment can be automatically generated including randomization policies for input variables. Concerning our hybrid approach, on one hand, the formal verification is able to guide the simulation process based on the counterexamples. On the other hand, the simulation engine supports the formal verification, for instance, with the assignment of automated static parameters in order to shrink the state space. In previous work [3], the SPA was determined by hand or using a random selection, that is, a try-and-error method. In this work, we enable for the first time the automated assignment of static parameters via a new ranking algorithm with the following specific contributions:

- Automated SPA: An automatic usage of SPA is possible using this heuristic.
- Testbench: Automatic generation of testbenchs and simulation/formal models.
- Code quality and safety: The ranking can be used to detect dead parameters as well as high complex functions based on high rated parameters.
- Minimize time/costs: The effort of brute-forcing all parameters using SPA can be reduced by testing specific and promising parameters.
- Maximized coverage: Using this heuristic the smallest restriction and therefore the widest coverage can be determined.

8.3 VERIFYR Verification Methodology

Figure 8.1 and Algorithm 4 delineates the semiformal verification algorithm. Our approach is based on the analysis of the embedded software structure via a function call graph (FCG), as shown in Fig. 8.1a. The FCG represents the calling relationships between functions of embedded software. The verification strategy is divided in three phases: preprocessing (Algorithm 4, lines 2–8), formal exploration phase (a.k.a. bottom-up) (Algorithm 4, lines 10–17), and semiformal verification phase (a.k.a. top-down) (Algorithm 4, lines 18–29). In summary, the Formal Exploration (bottom-up) phase identifies which functions are too complex to be verified by standalone software model checkers. After identifying these functions we start the Semiformal (top-down) phase combining simulation, SPA and formal verification in order to overcome the software complexity.

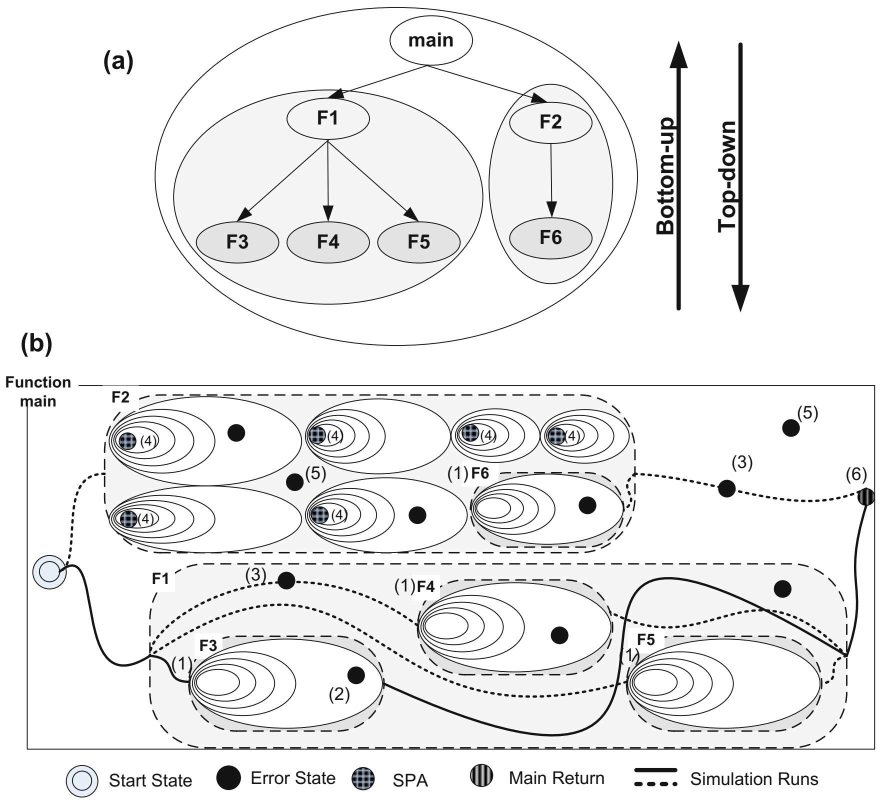

Fig. 8.1 VERIFYR verification approach

The VERIFYR verification methodology starts with the Formal Exploration phase. It uses state-of-the-art software model checkers (SMC) with built-in and user-defined properties specified in LTL to verify all functions of the FCG. It begins with the leaves (e.g., functions F3, F4, F5, F6 in Fig. 8.1) and continues in the next upper levels until the verification process reaches the function `main` (Algorithm 4, lines 11–12). If it is not possible to verify a function with the SMCs, it is marked in the FCG (Algorithm 4, line 13) (e.g., function `main`, F1, and F2 in Fig. 8.1). This means that these functions are too complex to be verified by a standalone model checker due to time out (TO) or out of memory (MO) constraints and that it is required to perform the semiformal/hybrid phase. Finally, a marked FCG (mFCG) is returned including all functions that failed during the Formal Exploration phase, however, if mFCG is empty then all function were formally verified and the verification process is completed (Algorithm 4, lines 15–16).

The Semiformal phase starts the simulation run with the assertion-based approach (ABV) (Algorithm 4, line 19), which requires one simulation model (or original C program wrapped in SystemC model) and testbench, one or many properties in LTL to be checked. We use a simulation approach based on SystemC. Thus, we derive a SystemC model from the embedded software and to be applied to SystemC Temporal Checker (SCTC) [32], which supports specification of user-defined properties in LTL [33].

During the Semiformal phase the marked FCG (mFCG) is analyzed. All functions that were not yet verified due to failed verification (in the Formal Exploration phase) are marked as point of interest (POI). POIs are basically the initial states of the local functions (F2 in Fig. 8.1b4). Therefore, the mFCG is used as a guiding mechanism in order to determine which function should be verified at the formal verification phase.

The Simulation engine monitors the simulation process (i.e., properties and variables) to start a new formal verification process at every POI (Algorithm 4, line 21). ABV is responsible for finding the POIs as well as the error states (F2 in Fig. 8.1b3). We use the monitored information to initialize variables (interaction with formal) to statically assign parameters (Algorithm 4, lines 23–24). It will lead to different access points for the software model checkers and it will help shrinking the state space of the function (Fig. 8.1b). This heuristic avoids an over-constraining of the state space in formal verification. As a result, SMC has not only a unique starting state (as usual by simulation), but an initial state set, which will improve the state space coverage of the semiformal verification. Therefore, the formal verification benefits from the simulation, as show in Fig. 8.1(F2).

Finally, a temporary version of the source code of the function under test (FUT) is created and is checked with the formal SMCs (Algorithm 4, lines 25). 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. Additionally, if desired, the user can set randomization constraints manually. Currently, when a counterexample should be reported 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.

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 or no more marked functions are available (Algorithm 4, line 20).

In the next sections, the SPA heuristic as well as the modeling details of embedded software will be presented.

Algorithm 4 VERIFYR algorithm

```
  VERIFYR(Cprog,PropSet)
     doPreProcessing()
          C3AC = 3ACGen(Cprog)
          CTestbench = testbenchGen(Cprog)
          C3AC = propToAssertionSMC(C3AC, PropSet)
          CTestbench = propToAssertionSim(CTestbench, PropSet)
          FCG = FCGgen(C3AC)
     end doPreProcessing
     startOrchestrator()
          doFormalExploration() //BOTTOM-UP
                for each CFunction in C3AC
                     VStatus = startSMC(CFunction)
                     if VStatus == FAIL then
                          mFCG += markFunction(CFunction)
                if mFCG == NULL then
                     return VCOMPLETE
           end doFormalExploration
           doSemiformal() //TOP-DOWN
                startSimulation(CTestbench, Cprog)
              while NoTimeBound or NoMaxSimRuns or !Empty(mFCG)
                     POIFunction = watchSimFunctions(mFCG)
                     if POIFunction in mFCG then
             assessParameterScore(POIFunction)
             CFunction = doSPA(POIFunction)
       VStatus = startSMC(CFunction)
       if VStatus == COUNTEREXE  then
                    guideSimulation()
             unmarkFunction(mFCG)
     end doHybridFormalSimulation
     doComputeCoverage()
     doShowCounterexample()
     end startOrchestrator
end VERIFYR
```

8.3.1 SPA Heuristic

The SPA heuristic assumes that there is a function list containing all functions with all their parameters in a structured way. The algorithm iterates through the statements of each function body in the function list, inspecting each statement. If a statement contains one of the function parameters, this statement is inspected in more details. The analysis covers 11 aspects of the statement called properties. More details on these properties are in Sect. 8.3.1.2. Based on these properties the statement is assessed and a score is computed. The scores are summed up and stored for each parameter. The statement is examined for introducing a parameter value-dependent variable

(PVDV). This is done after the property check since the first statement of this PVDV is not rated. They are queued in the parameter list marked with their depending parameter. The achieved score of a PVDV is added to the score of the parameter the PVDV depends on.

8.3.1.1 Parameter Value-Dependent Variables

Variables that are initialized using a parameter are directly affected by SPA. This observation led to the concept of PVDV. Applying SPA on a function parameter reduces the model size because the model is not required to cover the full range of possible values. This effect is passed down to PVDV as they are directly depending on the parameter value. They passed the effect on to their value-dependent variables.

In order to cover this effect this technique monitors the value dependencies by analyzing assignments. If a PVDV is found, the variable is queued in the list of parameters. Any impact on the model size (such as PVDV) is added to the impact of the function parameter on which value the PVDV was initialized. Keeping track of these PVDVs is an essential part of this technique. This is because it is a common practice to make copies of parameters if those are used at multiple locations. And the parameters that are used in multiple locations have a huge impact.

8.3.1.2 Context Properties

Every statement that contains a parameter is evaluated against a set of internal properties. These properties describe the context in which the parameter is used within the statement. Therefore, the properties cover all context aspects of a statement that are used to state an assessment. All properties are determined by inspecting the code and are the base for the later assessment. Following eleven properties reflect the aspects of a statement regarding the usage as a variable:

- **Reading**: True if the parameter is on the right hand side of an assignment.
- **Writing**: True if the parameter is on the left hand side of an assignment.
- **Compare**: True if there is a comparison.
- **Loop**: True if statement contains the keyword “for” or “while.”
- **Function**: True if the parameter is in brackets, as it would be when used as function parameter.
- **Conditional**: True if the statement contains the keyword “if,” “switch,” or “case.”
- **Return**: True if the statement starts with the keyword “return.”
- **Command**: True if the statement ends with a semicolon.
- **Multiple use**: True if the statement contains one parameter multiple times.
- **Indirect use**: True if the monitored parameter is not a direct parameter but a PVDV.
- **First use**: True only at the first appearance of a parameter.

8.3.1.3 Assessing Function

The assess function estimates the model size based on the usage of a parameter in a statement. The estimation is based on the properties and is implemented as a Boolean clause. The number of cases with significant impact on the model size is limited. In this heuristic the four following special cases are used:

- **Dead parameter**: If a parameter is written on the first appearance the parameter is considered dead. SPA may already be applied.
- **Return**: Actual function parameters (not PVDV) that are returned have lesser impact.
- **Switch statement**: Conditional parameters control the program flow and therefore, impact heavily on the model size.
- **Loop boundary**: Loops are commonly unwound within the formal model, so the loop boundary has a major impact on the model size.

Each case is rewarded with a score. The number of points per case is reflecting the impact on the model size. As every statement has a basic impact every statement receives one point. If variations of a parameter value do not impact the model size, the parameter is called a dead parameter. These dead parameters are mainly parameters that already have SPA applied. This case is rewarded with a negative score to lower the ranking to a minimum. In addition dead parameters are not assessed anymore. Return parameters are mainly data containers that have lesser impact on the model size than not returned parameters. This is an observation made while testing the heuristic on the available benchmark. An implementation should use a low negative score to cover this effect. Parameters that are used in conditional statements have great impact on the model size as they control the program flow. The score should be set to a high value to ensure that parameters that are not used in conditional or loop statements cannot reach a higher rank. Loop boundary parameters are parameters that control the boundary of a loop. As loops are commonly unwound in the formal model the impact of those parameters on the model size are huge. Experiments using the available benchmark showed that the impact on the model size of three conditional statements can surpass the impact of one loop boundary parameter. In order to cover this fact the score should be set to twice the score of a conditional statement.

8.3.2 Preprocessing Phase

Basically the preprocessing phase considers the generation of testbench and simulation models, preprocessing the C program to the software model checker, defining the temporal properties to both formal and simulation models, and finally the generation of the control flow graph, as shown in (Algorithm 4, lines 2–8).

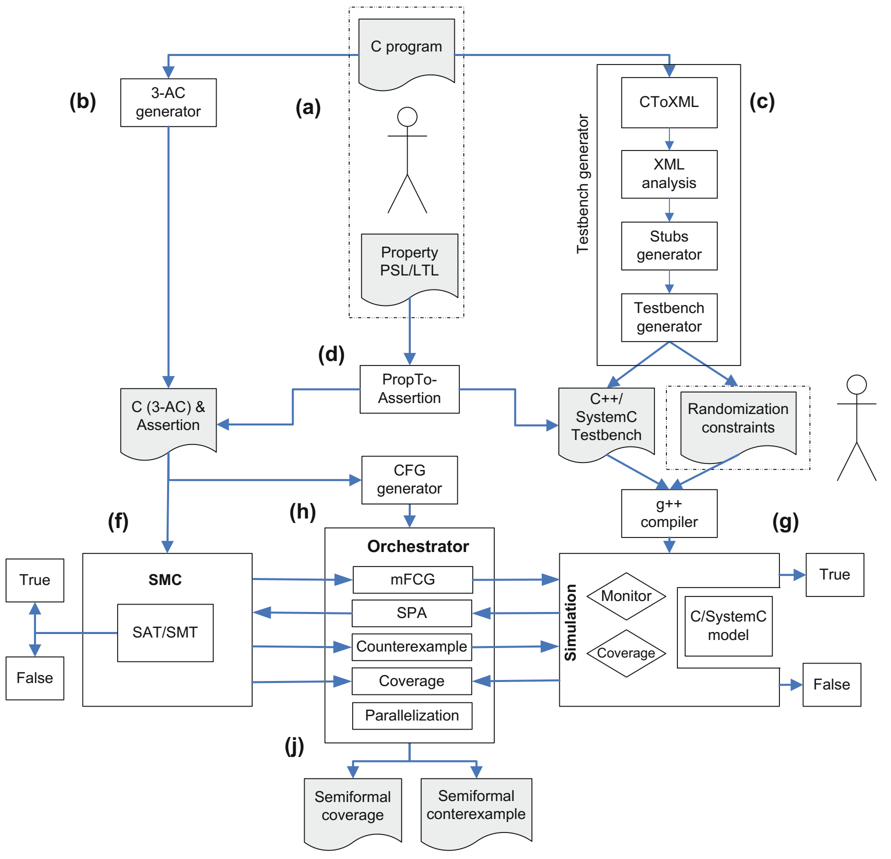

Fig. 8.2 VERIFYR overview

8.3.2.1 3-AC and FCG Generation

In order to extract the formal model we use the CIL [34] tool in the front-end to convert the C program (compatible with MISRA [35]) into three-address code (3-AC) (Fig. 8.2b). 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.

A function call graph (FCG) is generated based on [36]. We use this FCG as input to guide our Formal Exploration phase (bottom-up) verification.

8.3.2.2 Testbench Generation

For automatic testbench generation (Fig. 8.2c), we use our own XML-based approach. The objective of our testbench generator is to extract all input variables out of any

C source code and provide them in a portable overview easy to modify by hand or by using supported automatic manipulation methods. In order to reach these goals we generate a XML representation of the C code. Afterwards we analyze the corresponding XML, such as, macros, function monitoring, identification of local and input variables, value ranches of variables and loop analysis. After this step we generate the testbench using either C++ executable or SystemC model.

During the simulation we measure the coverage of the loop behavior and value ranges of all variables. The results of static XML analysis and the dynamic testbench execution are sent to the VERIFYR to enhance the automatic SPA with value ranges for variables and bounds for loop unrolling. All gathered information is presented to the user. The user can access the testbench XML description to update or manipulate the behavior.

8.3.2.3 SystemC Model

The derived simulation model is automatically generated using no abstractions. 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`) [37]. 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.

The automatically generated testbench includes all input variables and it is possible to choose between different randomization strategies like constrained randomization and different random distributions, supported by the SystemC Verification Library (SCV) [38].

8.3.2.4 Temporal Properties Definition

The C language does not support any means to check temporal properties in software modules during the simulation. Therefore, we use the existing SCTC, which is a hardware oriented temporal checker based on SystemC. SCTC supports specification of properties either in PSL (Property Specification Language), LTL or FLTL (Finite Linear time Temporal Logic) [32], an extension to LTL with time bounds on temporal operators. 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) (Fig. 8.2d) in the form of an Intermediate Language (IL) and later to a monitor in SystemC. The AR can detect validation (i.e., `True`) or violation (i.e., `False`) of properties (Fig. 8.2g) on finite system traces, or they stay in a pending state if no decision can be made yet.

For the software model checkers, we include the user-defined properties into the C code translating the LTL-style properties into assert/assume statements based on [39] (Fig. 8.2d).

8.3.3 Orchestrator

The orchestrator has as main function the coordination of the interaction between the assertion-based (i.e., simulation) and formal verification engines (Fig. 8.2h). Concerning that each SMC has their pros and cons, the formal verification is performed by the available state-of-the-art SAT/SMT based model checkers (e.g., CBMC, ESBMC). The simulation is performed by the SystemC kernel.

Additionally, the orchestrator collects the verification status of the software model checkers in order to "mark" the FCG. The mFCG is passed to the simulation engine to start the simulation process to determine values to the function parameters. Also, SPA is performed in order to identify the most important function parameter. The marked C function is updated with the static parameters and the SMC is executed in order to verify the properties. If a counterexample occurs, it will be used to guide the test vector randomization. Additionally, the orchestrator is responsible to collect the coverage data in order to determine the verification quality.

Finally, the orchestrator can distribute the computation of every function to a different verification instance of the supported SMCs (Fig. 8.3). 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 (e.g., in a cluster) of the functions manually and choose between the different SMCs by using a graphical user interface (GUI).

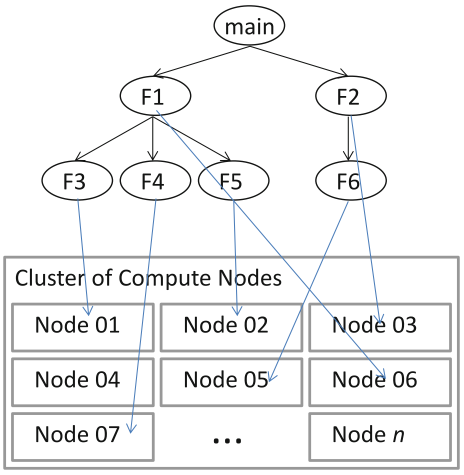

Fig. 8.3 Verification process distribution

8.3.4 Coverage

Our hybrid verification approach combines simulation-based and formal verification approaches. However, techniques to measure the achieved verification improvement have been proposed either to simulation-based or to formal verification approaches. Coverage determination for semiformal (hybrid) verification approaches is still in its infancy. For this work (Fig. 8.2j) we used a specification-based coverage metric to quantify the achieved verification improvement of hybrid software verification. Our semiformal coverage metric is based on "property coverage," which determines the total number of properties from a set of properties that were evaluated by both simulation-based or formal verification engines. Additionally, the simulation part is monitored using Gcov [40] in order to measure further implementation-based coverage inputs (e.g., line coverage, branch coverage). It is also important to point out, that due to the use of the simulation in our hybrid verification approach we still might not cover 100% of the state space, as in formal verification, as shown in Fig. 8.1b5).

8.3.5 Technical Details

The main objective 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 compute nodes 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), 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) over HTTP respectively HTTPS as shown in Fig. 8.4. 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. As shown in Fig. 8.5 the verification clients have to send their verification requests to a super node (orchestrator). The super node distributes the requests to different verification servers. At the moment VERIFYR supports multicore compute nodes and clusters. It is possible to setup any number of verification nodes to reach the desired scalability.

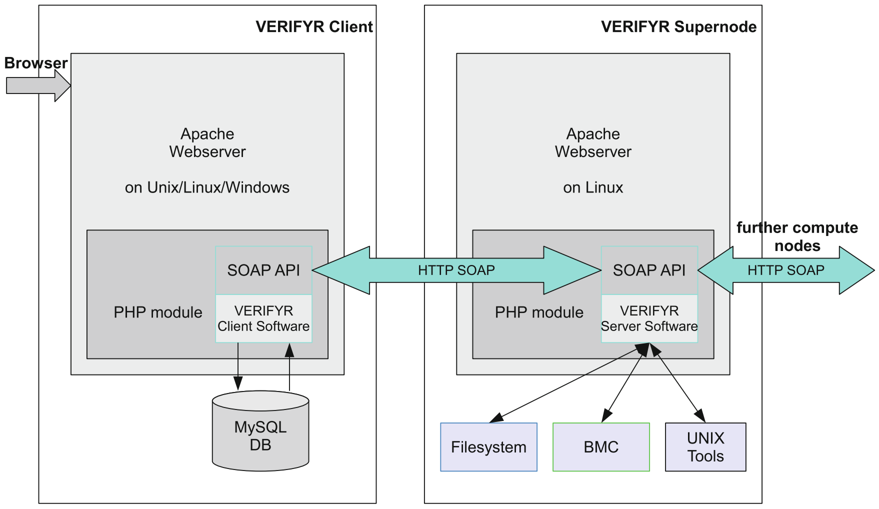

Fig. 8.4 VERIFYR client and server overview

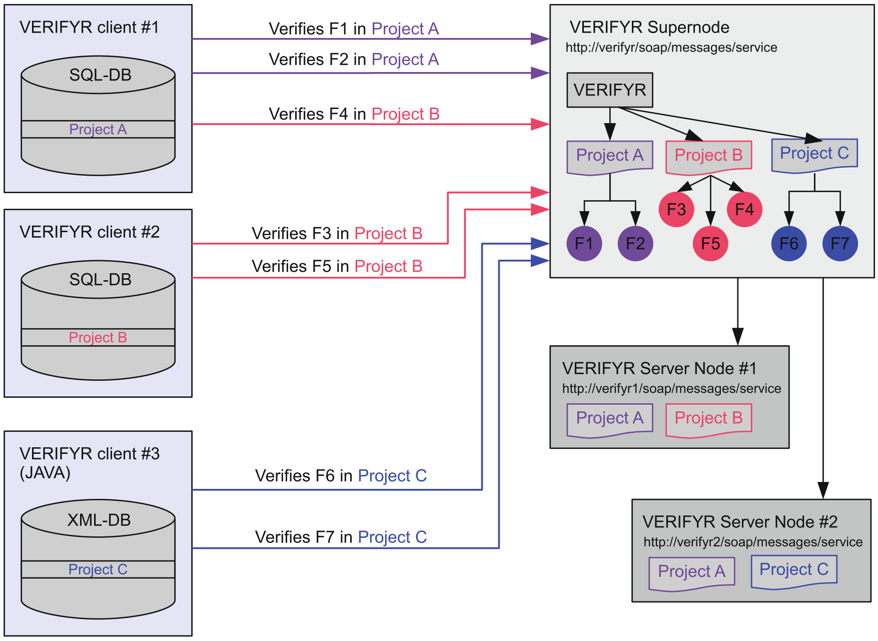

Fig. 8.5 Verification process of different clients on different servers

8.4 Results and Discussion

8.4.1 Testing Environment

We performed two sets of experiments based on two different case studies (cf., Sects. 8.4.2 and 8.4.4) 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 8 GB RAM and Linux OS. The first set of experiments represents the results of the SPA heuristic based on Motorolas Benchmark Suite [41] and the verification results of our new hybrid verification methodology (VERIFYR) using this new heuristic. The second set of experiments represents the results of the SPA heuristic based on EEPROM emulation software from NEC Electronics and the verification results of the hybrid verification methodology (VERIFYR) using this new heuristic.

The scores were set according to the rules given in Sect. 8.3.1.3. The empirically gained scores are −200 points for dead parameter, −20 points for return parameters, 1025 points for conditional parameters and 2050 points for loop parameters. Then two adjustments had to be made. The first adjustment was the score of conditional parameters. Those can easily be set too low. Setting the score too low leads to a wrong ranking compared to parameters that are often used, but not as conditional or loop parameters. The actual number 1025 is an empirical choice based on the case studies. The second adjustment is the score for loop parameters. This score is reflecting used the coding style. Using many and long conditional code blocks the score for a loop decreases, while using wide loops or conditional constructions with else case the score increases. For the Motorola Powerstone Benchmark Suite the score twice the score of conditional parameters showed to be fitting. Adjusting the scores slightly will have only a small effect, but it may swap parameters that are close.

8.4.2 Motorola Powerstone Benchmark Suite

For our first case study we used Motorola's Powerstone Benchmark Suite [41] and tried to verify the built-in properties (e.g., division-by-zero) from CBMC and ESBMC. To exemplify these results the `search_dict` function from the Motorola Powerstone Benchmark module V.42 was used. The function has two parameters `string` and `data`. With these two parameters in the parameter list the algorithm proceeds through the function body. Table 8.1 shows the result for each statement. That would be 2 points for `data` and `string` and 7 points for `kid`. The score of each parameter is summed up including the appearance points. The resulting ranking is 1007 points for `data`, 3077 points for `string` and 2057 points for `kid`. However, as `kid` is inherited by `string` the score is combined to a final result of 1007 points for `data` and 5134 points for `string`.

As the score represents the impact of each parameter, it can be expected that the `string` parameter has a much bigger impact than the `data` parameter. To test the

Table 8.1 Statement scoring

Line: statement	Parameter	Points	Reason
l2: if (!string)	String	1025	Switch statement
l3: return (data + 3);	Data	−20	Return statement
l4: for (kid = dict[string].kids; …	String	2050	Loop statement, also introduces new parameter "kid"
l4: kid; …	Kid	0	Not a statement, not a loop statement as the loop defines "kid"
l4: kid = dict[kid].sibling)	Kid	0	Not a statement still part of the "for" instruction
l5: if (kid != last && …	Kid	1025	Switch statement
l5: dict[kid].data == data)	Kid	1025	Switch statement actually it is the same switch statement
l5: dict[kid].data == data)	Data	1025	Switch statement two parameter, scored twice
l6: return (kid);	Kid	0	One point for use not a return statement because "kid" is inherited

impact of the ranking, the function has been verified using CBMC with an unwinding option of 20 and again for each parameter. Using SPA on the parameter `data` does not change that result. The verification run resulted in about 5 s and with a memory usage of up to 175 MB. Using SPA on the parameter `string` results in a runtime of 4 s and a maximum memory usage of 65 MB. So the memory usage has been more than halved and the runtime reduced when SPA is used on the parameter the heuristic suggests. This experiment shows that SPA on the parameter improves the memory usage and the runtime. In order to show the power of SPA in more detail the function `memcpy` of the V.42 module is a good example. This function has three parameters with a scoring printed in Table 8.2. Using CBMC without unwinding this function needs more than 3 GB of memory, which leads to an out-of-memory exception in the used test environment.

Using the ranking provided by this heuristic the first parameter is a dead parameter, so applying SPA on it should lead to no further information. Applying SPA to the first parameter leads indeed to an out-of-time exception after one hour of runtime. The second parameter has a low impact on the model size. Applying SPA on the second parameter leads to another out-of-memory exception. The final parameter with the highest score has the highest impact on the model size. After applying SPA

Table 8.2 SPA results for V.42

Function	Parameter	Score	CPU[a]	Mem[b]	Vmem[b]	Comment
memcpy			458,318	276,558	2935,808	MO[c]
	*void *d*	−218	3599,565	107,089	35,652	MO[c]
	*void *s*	2	120,324	101,972	2931,712	MO[c]
	long t	1029	6,464	0,197	59,488	51[d]
strncmp			212,686	185,147	2928,64	MO[c]
	*char *s1*	2052	233,442	201,603	2939,904	MO[c]
	*char *s2*	2052	244,351	210,341	2921,472	MO[c]
	long n	8207	1,503	0,013	35,668	10704[d]

[a]seconds of runtime
[b]megabyte (virtual) memory used
[c]memory out
[d]number of clauses, all results are retrieved using CBMC with no unwind bound

on that parameter CBMC, returns "verification failed." The second experiment is the `strncmp` function of the V.42 module. This function has three parameters with the scoring shown in Table 8.2. Unlike the `memcpy` function the scoring of two parameters are close by. This suggests similar results when using SPA on either of them. Table 8.2 shows that this assumption is correct in this case. The third parameter with the highest score indeed has the highest impact on the model size and leads to a final result.

8.4.3 Verification Results Using VERIFYR

We combined the new SPA heuristic with the VERIFYR platform. 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. We tried to verify the built-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. For every function we used a different instance of CBMC or ESBMC in parallel. The results are shown in Table 8.3. Based on this Formal Exploration analysis, we switched to our top-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 8.3 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 min), 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. This table shows that VERIFYR presented the same valid results as CBMC (SAT)

Table 8.3 Verification results v42.c

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

P (passed), F (failed), MO (out of memory),
TO (time out, 90 min) and PH (passed using hybrid methodology)

and ESBMC, and no MO or TO has occurred. Furthermore, the Table 8.3 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. In total, 20 properties were evaluated by both simulation and formal verification. All tested properties were safe, that is, a property coverage of 100%.

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.

8.4.4 EEPROM Emulation Software from NEC Electronics

Our second case study is an automotive EEPROM Emulation software from NEC Electronics [42], which emulates the read and write requests to a nonvolatile 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 code used is property of NEC Electronics (Europe) GmbH, embedded and marked confidential. Therefore, the details of the implementation are not discussed. 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). 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:

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

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 [39]. For the SPA heuristic the same scoring as in the Motorola Powerstone Benchmark Suite was used. The verification was done on the same computer as the previous testing and the verification runs were unbounded. We present three functions to provide evidence that the concept of this heuristic is valid and the scoring is balanced. In Table 8.4 the measured results are shown.

The function `DFA_Wr` is successfully verified using SPA on the `length` parameter. This result is suggested by the heuristic. In the function `DFA_WrSec` the parameter `val` has the highest score. And the function also finishes using SPA on that parameter. Unlike in the two other functions the function `DFALib_SetWr` is valid from the beginning. CBMC verifies the function in half a second using 1341 clauses. Still using SPA shows that if the score of the parameters increase then the number of clauses generated and proven by CBMC decreases. This shows that the score is representing the complexity of parameters concerning the resulting state space. Unbounded model checking can be restricted in order to gain a partial result. The case studies above show that the increased complexity of software can be handled using SPA.

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 renamed the selected functions for convenience. Table 8.5 shows that VERIFYR presented the same valid results as CBMC (SAT) and ESBMC, and no MO or TO has occurred. In total, 40 properties were evaluated by both simulation and formal verification, which corresponds five properties for each of the eight functions. All tested properties were safe, that is, a property coverage of 100%.

Table 8.4 SPA results for NEC

Function	Parameter	Score	CPU[a]	Mem.[b]	Vmem.[b]	Comment
DFA_Wr			127,576	107,435	2917,376	MO[c]
	*void *addSrc*	15	127,964	106,242	2929,664	MO[c]
	*void *addDest*	15	138,541	116,533	2934,784	MO[c]
	u32 length	3093	0,534	0	0	VS[d]
DFA_WrSec			129,523	109,166	2939,904	MO
	*u08 volatile *sec*	−199	129,826	106,599	2934,784	MO[c]
	*u08 volatile *dest*	829	133,273	111,806	2936,832	MO[c]
	u08 mask	1852	123,984	105,334	2922,496	MO[c]
	u08 val	4115	0,521	0,002	21,141	51[e]
DFA_SetWr			0,552	0,003	21,148	1341[e]
	*u32 *pWrite-data*	19	0,541	0	0	1031[e]
	u32 cnt	1853	0,5	0	0	29[e]

[a]seconds of runtime
[b]megabyte (virtual) memory used
[c]memory out
[d]verification successful
[e]number of clauses, all results are retrieved using CBMC with no unwind bound

Table 8.5 Verification Results NEC

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

P (passed), F (failed), MO (out of memory),
TO (time out, 90 min) and PH (passed using hybrid methodology)

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.

8.5 Conclusion and Future Work

We have presented our scalable and extendable hybrid verification approach for embedded software. We have described our new semiformal verification methodology and have pointed out the advantages. Furthermore we have shown our new SPA heuristic, which shows promising results on the Motorola Powerstone Benchmarks Suite and on the EEPROM emulation software from NEC Electronics. SPA is an automated process that optimizes the interaction between bounded model checking and simulation for semiformal verification approaches. 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 formal exploration phase the SMC 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 the POIs 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 and without hardware dependencies. It scales better than standalone 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 assessing the scores automatically and on quality metrics for hybrid verification.

Acknowledgements The authors would like to thank Edgar Auerswald, Patrick Koecher and Sebastian Welsch for supporting the development of the VERIFYR platform.

References

1. Jerraya AA, Yoo S, Verkest D, Wehn N (2003) Embedded software for SoC. Kluwer Academic Publishers, Norwell, MA, USA
2. Beyer D, Henzinger TA, Jhala R, Majumdar R (2007) The software model checker BLAST: applications to software engineering. Int J Softw Tools Technol Trans
3. Behrend J, Lettnin D, Heckler P, Ruf J, Kropf T, Rosenstiel W (2011) Scalable hybrid verification for embedded software. In: DATE '11: proceedings of the conference on design, automation and test in Europe, pp 1–6
4. Barrett C, Sebastiani R, Seshia SA, Tinelli C (2009) Satisfiability modulo theories. Frontiers in artificial intelligence and applications, Chap 26, vol 185. IOS Press, pp 825–885

5. Clarke E, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: Tools and algorithms for the construction and analysis of systems. Springer, pp 168–176
6. Kroening D (2009) Bounded model checking for ANSI-C. http://www.cprover.org/cbmc/
7. Biere A, Cimatti A, Clarke EM, Strichman O, Zhu Y (2003) Bounded model checking. In: Zelkowitz M (ed) Highly dependable software. Advances in computers, vol 58. Academic Press
8. Cordeiro L, Fischer B, Marques-Silva J (2009) 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. IEEE Computer Society, Washington, DC, pp 137–148
9. Ball T, Majumdar R, Millstein T, Rajamani SK (2001) Automatic predicate abstraction of C programs. SIGPLAN Not 36:203–213
10. Flanagan C, Qadeer S (2002) Predicate abstraction for software verification. SIGPLAN Not 37:191–202
11. Clarke E, Grumberg O, Jha S, Lu Y, Veith H (2003) Counterexample-guided abstraction refinement for symbolic model checking. J ACM 50:752–794
12. Clarke E, Grumberg O, Long D (1994) Model checking and abstraction. ACM Trans Prog Lang syst 16(5):1512–1542
13. Henzinger TA, Jhala R, Majumdar R (2005) The BLAST software verification system. Model Checking Softw 3639:25–26
14. Clarke E, Kroening D, Sharygina N, Yorav K (2005) SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS, vol 3440. Springer, pp 570–574
15. Gorai S, Biswas S, Bhatia L, Tiwari P, Mitra RS (2006) Directed-simulation assisted formal verification of serial protocol and bridge. In: DAC '06: proceedings of the 43rd annual design automation conference. ACM, New York, pp 731–736
16. Nanshi K, Somenzi F (2006) Guiding simulation with increasingly refined abstract traces. In: DAC '06: proceedings of the 43rd annual design automation conference. ACM, New York, pp 737–742
17. Di Guglielmo G, Fummi F, Pravadelli G, Soffia S, Roveri M (2010) Semi-formal functional verification by EFSM traversing via NuSMV. In: 2010 IEEE international High level design validation and test workshop (HLDVT), pp 58–65
18. Edwards SA, Ma T, Damiano R (2001) Using a hardware model checker to verify software. In: proceedings of the 4th international conference on ASIC (ASICON)
19. Lettnin D, Nalla PK, Behrend J, Ruf J, Gerlach J, Kropf T, Rosenstiel W, Schönknecht V, Reitemeyer S (2009) Semiformal verification of temporal properties in automotive hardware dependent software. In: DATE'09: proceedings of the conference on design, automation and test in Europe, pp 1214–1217
20. Ruf J, Peranandam PM, Kropf T, Rosenstiel W (2003) Bounded property checking with symbolic simulation. In: FDL
21. Cordeiro L, Fischer B, Chen H, Marques-Silva J (2009) Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In: Second international conference on embedded software and systems, pp 396–403
22. Godefroid P, Klarlund N, Sen K (2005) Dart: directed automated random testing. SIGPLAN Not 40(6):213–223. http://doi.acm.org/10.1145/1064978.1065036
23. Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR (2006) Exe: automatically generating inputs of death. In: Proceedings of the 13th ACM conference on computer and communications security. CCS'06. ACM, New York, pp 322–335. http://doi.acm.org/10.1145/1180405.1180445
24. Sen K, Marinov D, Agha G (2005) CUTE: a concolic unit testing engine for C. SIGSOFT Softw Eng Notes 30(5):263–272. http://doi.acm.org/10.1145/1095430.1081750
25. Di Guglielmo G, Fujita M, Fummi F, Pravadelli G, Soffia S (2011) EFSM-based model-driven approach to concolic testing of system-level design. In: 2011 9th IEEE/ACM international conference on formal methods and models for codesign (MEMOCODE), pp 201–209

26. Cadar C, Dunbar D, Engler D (2008) KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX conference on operating systems design and implementation. OSDI'08. USENIX Association, Berkeley, pp 209–224
27. Lattner C, Adve V (2005) The llvm compiler framework and infrastructure tutorial. In: Eigenmann R, Li Z, Midkiff S (eds) Languages and compilers for high performance computing. Lecture notes in computer science, vol 3602. Springer, Berlin, pp 15–16
28. Tillmann N, De Halleux J (2008) Pex: white box test generation for .net. In: Proceedings of the 2nd international conference on tests and proofs. TAP'08. Springer, Heidelberg, pp 134–153. http://dl.acm.org/citation.cfm?id=1792786.1792798
29. Cuoq P, Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B (2012) Frama-C: a software analysis perspective. In: Proceedings of the 10th international conference on software engineering and formal methods. SEFM'12. Springer, Heidelberg, pp 233–247
30. Correnson L, Signoles J (2012) Combining analyses for C program verification. In: Stoelinga M, Pinger R (eds) Formal methods for industrial critical systems. Lecture notes in computer science, vol 7437. Springer, Berlin, pp 108–130
31. Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B (2015) Frama-C: a software analysis perspective. Formal Aspects Comput:1–37
32. Weiss RJ, Ruf J, Kropf T, Rosenstiel W (2005) Efficient and customizable integration of temporal properties into SystemC. In: Forum on specification & design languages (FDL), pp 271–282
33. Clarke E, Grumberg O, Hamaguchi K (1994) Another look at LTL model checking. In: Dill DL (ed) Conference on computer aided verification (CAV). Lecture notes in computer science, vol 818. Springer, Stanford, pp 415–427
34. Necula GC, McPeak S, Rahul SP, Weimer W (2002) CIL: intermediate language and tools for analysis and transformation of C programs. In: Computational complexity, pp 213–228
35. MISRA (2000) MISRA—the motor industry software reliability association. http://www.misra.org.uk/
36. Shea R (2009) Call graph visualization for C and TinyOS programs. In: Department of computer science school of engineering UCLA. http://www.ambleramble.org/callgraph/index.html
37. Lettnin D, Nalla PK, Ruf J, Kropf T, Rosenstiel W, Kirsten T, Schönknecht V, Reitemeyer S (2008) Verification of temporal properties in automotive embedded software. In: DATE'08: proceedings of the conference on design, automation and test in Europe. ACM, New York, pp 164–169
38. Open SystemC Initiative (2003) SystemC verification standard library 1.0p users manual
39. Clarke E, Kroening D, Yorav K (2003) Behavioral consistency of C and verilog programs using bounded model checking. In: DAC'03: proceedings of the 40th annual design automation conference. ACM, New York, pp 368–371
40. GNU (2010) Gcov coverage. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
41. Malik A, Moyer B, Cermak D (2000) The M'CORE (TM) M340 unified cache architecture. In: Proceedings of the 2000 international conference on computer design, pp 577–580
42. NEC NEC Electronics (Europe) GmbH. http://www.eu.necel.com/