Optimized Hybrid Verification of Embedded Software

Jörg Behrend*, Alexander Gruenhage*, Douglas Schroeder†, Djones Lettnin†,
Jürgen Ruf*, Thomas Kropf*, and Wolfgang Rosenstiel*
* Department of Computer Engineering
University of Tübingen
Sand 13, 72076 Tübingen, Germany
E-mail: {behrend,gruenhag,ruf,kropf,rosenstiel}@informatik.uni-tuebingen.de
† Graduate Program in Computer Science (PPGCC)
Federal University of Santa Catarina
Campus Universitário s/n - Trindade - CEP 88040-900
Florianópolis, SC - Brazil
E-mail: lettnin@eel.ufsc.br

Abstract—The verification of embedded software has become an important subject over the last years. However, neither standalone verification approaches, like simulation-based or formal verification, nor state-of-the-art hybrid/semiformal verification approaches are able to verify large and complex embedded software with hardware dependencies. This work presents an optimized scalable hybrid verification approach for the verification of embedded software with hardware dependencies using a mixed bottom-up/top-down algorithm with optimized static parameter assignment (SPA). These algorithms and methodologies like SPA and counterexample guided simulation are used to combine simulation-based and formal verification in a new way. SPA offers a way to interact between dynamic and static verification approaches based on an automated ranking heuristic of possible function parameters according to the impact on the model size. Furthermore, SPA inserts initialization code for specific function parameters into the source code under test and supports model building and optimization algorithms to reduce the state space. We have successfully applied this optimized hybrid verification methodology to an embedded software application: Motorola’s Powerstone Benchmark suite. The results show that our approach scales better than stand-alone software model checkers to reach deep state spaces.

I. 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 (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) approaches. Testing, co-debugging and/or co-simulation techniques result in a tremendous effort to create test vectors. Furthermore, critical corner case scenarios might remain unnoticed. An extension of simulation is the assertion-based verification (ABV) methodology that captures a design’s intended behavior in temporal properties and monitors the 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 take 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 scalable hybrid verification approach using a mixed bottom-up/top-down algorithm taking advantage of automated static parameter assignment (SPA) [4]. 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 other problem is the selection of the value to assign. The selection of the value 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 resulting model size. The ranking is based on estimation according to the usage of the parameters in the function body. Based on this ranking, SPA can be automatically applied to select parameters in an optimized way in order to reduce the model size in a controlled manner.

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

II. RELATED WORK

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 [5] to check the formula for satisfiability. For instance, C bounded model checker (CBMC) [6], [7], [8] has proven to be a successful approach for automatic software analysis. Codeiro et al. [9] have implemented ESBMC based on the front-end of CBMC and a new back-end based on SMT. All the aforementioned 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 [10] introduced a way to construct abstracted models and allowed to introduce automated constraints like loop invariants [11]. Based on abstraction Clarke et al. developed a refinement technique to generate even smaller models using counterexamples [12] [13]. BLAST [14] and SATABS [15] are formal verification tools for ANSI-C programs. Both programs 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 [16], [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 checking. However, the symbolic model checker (SymC) [20] was the bottleneck for the scalability of the formal verification. The model checker SymC was originally developed for the verification of hardware designs. Cordeiro et al. [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 hardware-dependent software. 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. The previous approach to select the function parameter for SPA was a manually process or using a random selection.

A. Contributions

Our main contribution in this work is a novel semiformal approach for the verification of embedded software (compatible with MISRA [22]) with temporal properties. We provide a new methodology to combine different state-of-the-art formal and simulation-based approaches to a new hybrid verification approach, that is, a mixed bottom-up/top-down algorithm. 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 (SystemC) and the testbench environment can be automatically generated including randomization polices 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 this work, we enable for the first time the automated assignment of static parameters via a new ranking heuristic.

III. VERIFICATION METHODOLOGY

In this section we will give a short overview over the used verification methodology. Further details on the used VERIFYPR platform can be found in [3], [23] and about the automated SPA heuristic in [4]. The verification approach consists of three phases: preprocessing, bottom-up and top-down. Additionally, an orchestrator coordinates the aforementioned phases and the interaction between simulation and formal verification. We start a verification run with preprocessing the C code. As we have seen in the related work, the state-of-the-art software model checkers suffer from the embedded software complexity. In order to overcome this complexity, we developed a mixed bottom-up/top-down approach. Details about the single steps are given below.

A. Software Preprocessing

We convert the C program into three-address code (3-AC) and merge the C source code into one single file using CIL [24]. 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. Then a SystemC model is derived from the embedded software. No abstraction is used and therefore, the derived model is as precise as the original C program. A testbench is automatically generated and all input variables are randomized with constrains. Afterwards a function call graph (FCG) [25] is generated. We use this FCG as input to guide our bottom-up verification. Then we include the user-defined properties into the C code. Therefore, we translate the LTL-style properties into assert/assume statements based on [26].

B. Bottom-up Phase/Formal Exploration

After preprocessing the C code we start the bottom-up verification. We verify all functions of the FCG beginning with the leaves using state-of-the-art software model checkers with build-in properties and the user-defined properties specified in LTL. We distribute the computation controlled by the orchestrator of every function to a different verification instance of the supported software model checkers. If it is not possible to verify all functions of the FCG using the SMCs
(bottom-up/exploration), we switch to the hybrid top-down phase. Therefore, a marked FCG is returned including the status of the verification of all functions.

C. Top-down Phase/Hybrid Verification

The hybrid top-down phase starts with the analysis of the marked FCG (mFCG). All functions that were not yet verified due to failed verification (e.g., time out (TO) or out of memory (MO)) are marked as point of interest (POI). We use a simulation approach based on SystemC. Therefore, we derive a SystemC model from the embedded software, which supports specification of user defined properties in LTL [27]. The derived model is automatically generated using no abstractions. An automatically generated testbench is included with all input variables constrained. For refinement of the testbench we monitor the behavior using coverage metrics (e.g., code coverage). The orchestrator observes the simulation process (i.e., properties and variables) during the hybrid phase based on SystemC Temporal Checker (SCTC) [28] in order to start a new formal verification process at every POI. We use the monitored information to initialize variables (interaction with formal) to statically assign parameters and to create a temporary version of the source code of the function under test (FUT). These functions under test are distributed and checked with the formal SMCs. Static parameter assignment (SPA) will lead to different access points for the software model checkers and it will help shrinking the state space of the function. Therefore, the formal verification benefits from the simulation.

If a counterexample is reported, this information is used to guide the simulation (learning process). For instance, the randomization of input variables in our testbench is constrained in order to generate more efficient test vectors. The checked properties are the same as in bottom-up phase.

D. 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 subsection III-D2. 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 depending variables (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.

1) Parameter value depending 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 depending variables.

In order to cover this effect the heuristic 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 a 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 heuristic. 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.

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.

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 an 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 SPA) that are returned have a 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 the value of a parameter 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.

### IV. **Technical Details**

The VERIFYR framework [3] 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 [6], ESBMC [9], simulation tools (e.g., SCTC [28]), heuristics (e.g., SPA) and a communication gateway in order to invoke verification commands and to exchange status information of the hybrid verification process. For refinement of the testbench we monitor the behavior using code coverage provided by Gcov [29].

### V. **Results and Discussion**

#### A. Testing Environment

We performed two sets of experiments based on our case study conducted on a Intel® Core™ 2 Quad CPU Q9650 @ 3.00 GHz with 8GB RAM and Linux OS. The first set represents the results of the SPA heuristic based on Motorolas Benchmark Suite [30]. The second set of experiments shows the verification results of our new scalable and extendable hybrid verification methodology (VERIFYR) using this new heuristic.

#### B. Results for SPA Heuristic on Motorola Benchmark Suite

For these results the “search dict” function taken 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 I shows the result for each statement (the one point each parameter gains on each appearance is missing). 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 “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 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 seconds and with a memory usage of up to 175 megabytes. Using SPA on the parameter “string” results in a runtime of 4 seconds and a maximum memory usage of 65 megabytes. 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 II. Using CBMC without unwinding this function needs more than 3 gigabyte 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 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 II. Unlike the “memcpy” function the scoring of two parameters are close by. This suggests similar results when using SPA on either of them. Table II 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.

#### C. 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
TABLE I  
STATEMENT SCORING

<table>
<thead>
<tr>
<th>Statement Parameter</th>
<th>Points</th>
<th>Reason</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (!string) string</td>
<td>1025</td>
<td>switch statement</td>
<td>also introduces new parameter “kid”</td>
</tr>
<tr>
<td>data + 3;</td>
<td>-20</td>
<td>return statement</td>
<td>not a loop statement as the loop defines “kid”</td>
</tr>
<tr>
<td>for (kid = dict[string].kids; string 2050</td>
<td>loop statement</td>
<td>actually it is the same switch statement</td>
<td></td>
</tr>
<tr>
<td>kid; kid 0</td>
<td>0</td>
<td>not a statement</td>
<td>still part of the “for” instruction</td>
</tr>
<tr>
<td>kid; kid 0</td>
<td>not a statement</td>
<td>still part of the “for” instruction</td>
<td></td>
</tr>
<tr>
<td>kid = dict[kid].sibling)</td>
<td>1025</td>
<td>switch statement</td>
<td>two parameter, scored twice</td>
</tr>
<tr>
<td>dict[kid].data == data) data 1025</td>
<td>switch statement</td>
<td>actually it is the same switch statement</td>
<td></td>
</tr>
<tr>
<td>return (kid); kid 0</td>
<td>one point for use</td>
<td>not a return statement because “kid” is inherited</td>
<td></td>
</tr>
</tbody>
</table>

TABLE II  
RESULTS OF CASE STUDY II

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameter</th>
<th>Score</th>
<th>CPU¹</th>
<th>Memory²</th>
<th>Virt. Memory²</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcpy</td>
<td></td>
<td>458,318</td>
<td>276,558</td>
<td>2935,808</td>
<td>out of memory</td>
<td></td>
</tr>
<tr>
<td>void *d</td>
<td></td>
<td>3599,565</td>
<td>107,089</td>
<td>35,652</td>
<td>out of time</td>
<td></td>
</tr>
<tr>
<td>void *s</td>
<td></td>
<td>120,324</td>
<td>101,972</td>
<td>2931,712</td>
<td>out of memory</td>
<td></td>
</tr>
<tr>
<td>long t</td>
<td></td>
<td>6,464</td>
<td>0,197</td>
<td>59,488</td>
<td>Verified 51 clauses</td>
<td></td>
</tr>
<tr>
<td>strcmp</td>
<td></td>
<td>212,686</td>
<td>185,147</td>
<td>2928,64</td>
<td>Out of memory</td>
<td></td>
</tr>
<tr>
<td>char *1</td>
<td></td>
<td>2052</td>
<td>201,603</td>
<td>2939,904</td>
<td>Out of memory</td>
<td></td>
</tr>
<tr>
<td>char *2</td>
<td></td>
<td>2052</td>
<td>210,341</td>
<td>2921,472</td>
<td>Out of memory</td>
<td></td>
</tr>
<tr>
<td>long n</td>
<td></td>
<td>8207</td>
<td>1,503</td>
<td>35,668</td>
<td>Verified 10704 clauses</td>
<td></td>
</tr>
</tbody>
</table>

¹ Seconds of runtime  
² Megabyte (virtual) memory used, all results are retrieved with no unwind bound. First line of each function is without using SPA and other lines with SPA and the corresponding variable.

verify the build-in properties (e.g., division by zero, array out of bounds) from CBMC and ESBMC. It was not possible to verify the whole program using one of the above mentioned SMCs with a unwinding parameter (bound) bigger than 4. For every function we used a different instance of CBMC or ESBMC in parallel. The results are shown in Table III. Based on this bottom-up 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 III shows the comparison between CBMC (SAT), ESBMC and our VERIFYR platform. The used symbols are P (passed), F (failed), MO (out of memory), TO (time out, 90 minutes) and PH (passed using hybrid methodology). PH means that it was possible to verify this function with our hybrid methodology using simulation to support formal verification with static parameter assignment. All tested properties were safe. This table shows that VERIFYR presented the same valid results as CBMC (SAT) and ESBMC, and no MO or TO has occurred. Furthermore, the table presents the verification time in seconds in order to reach P, MO or PH results. The time for PH consist of the time for the simulation runs plus formal verification using static parameter assignment. We have used 1000 simulation runs. Overall, we have simulated the whole modem encoding/decoding software using our automatically generated testbench and beyond that we are able to verify 6 out of 12 observed functions using formal verification and the 6 remaining with hybrid verification. However, VERIFYR outperforms the single state-of-the-art tools in complex cases where they are not capable to reach a final verification result.

VI. CONCLUSION AND FUTURE WORK

We have presented our scalable and extendable hybrid verification approach for embedded software. We have described our new bottom-up/top-down verification methodology and have pointed out the advantages of this approach. Furthermore we have shown our SPA heuristic, which shows promising results on the Motorola Powerstone Benchmarks Suite. Using this heuristic, 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 bottom-up/exploration phase formal verification tries to verify all possible functions under test based on a FCG until a time bound or memory limit has been reached. The FCG is marked to indicate the Points-of-Interest. Then, we start with simulation and whenever one of 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 hardware dependencies. It scales better than stand-alone software model checkers and reaches deep state spaces. Furthermore, our approach can be easily integrated in a complex software development process. Currently, we are working on quality metrics for hybrid verification.

REFERENCES


