Test-Case Generation for Embedded Binary Code Using Abstract Interpretation

Thomas Reinbacher[1], Jörg Brauer[2], Martin Horauer[3], Andreas Steininger[1], and Stefan Kowalewski[2]

1 **Embedded Computing Systems Group, Institute of Computer Engineering, Vienna University of Technology, Treitlstrasse 3, A-1040 Vienna, Austria**
{treinbacher, steininger}@ecs.tuwien.ac.at

2 **Embedded Software Laboratory, RWTH Aachen University, Ahornstraße 55, D-52074 Aachen, Germany**
{lastname}@embedded.rwth-aachen.de

3 **Department of Embedded Systems, University of Applied Sciences Technikum Wien, Höchstädtplatz 5, A-1200 Vienna, Austria**
horauer@technikum-wien.at

Abstract

This paper describes a framework for test-case generation for microcontroller binary programs using abstract interpretation techniques. The key idea of our approach is to derive program invariants a priori, and then use backward analysis to obtain test vectors that are executed on the target microcontroller. Due to the structure of binary code, the abstract interpretation framework is based on propositional encodings of the program semantics and SAT solving.

1998 ACM Subject Classification C.3, D.2.4, D.2.5

Keywords and phrases Test-Case Generation, Embedded Binary Code, Abstract Interpretation

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.101

1 Introduction

Traditionally, formal verification and structural testing are considered as orthogonal concepts for increasing the quality of software. Whereas formal verification techniques such as model checking or abstract interpretation establish a full proof of correctness, testing increases confidence in the correctness of a system by meeting certain coverage criteria, where none of the examined paths violates the specification. However, the underlying coverage criteria, which are often dictated by industrial standards [20], are typically insufficient for finding property violations as argued by Heimdahl et al. [13].

In the embedded systems domain, verification and validation techniques should ideally be applied to the executable binary code of a program, since the exact semantics of the program is not unambiguously specified in high-level representations such as C code [1]. Further, it is not unknown for compilation itself to introduce errors [10]. However, embedded systems code often strongly relies on the behavior and state of the hardware and on interaction with the environment. The need to model these two properties properly, among others, aggravates the state explosion in model checking and limits its applicability. On the other hand, abstract interpretation provides a scalable approach to verification that often suffers from imprecision, and subsequently, a high number of spurious warnings. This is even more so on the binary-code level, where interleavings of arithmetic and logical operations as well as the finite precision of registers pose additional challenges.

Sixth Doctoral Workshop on Math. and Eng. Methods in Computer Science (MEMICS'10)—Selected Papers.
Editors: L. Matyska, M. Kozubek, T. Vojnar, P. Zemčík, D. Antoš; pp. 101–108

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In case of a violated property, abstract interpretation typically does not provide a counterexample, which is extremely helpful for fixing the defect [7]. By way of contrast, this property is fulfilled by both model checking and testing.

1.1 Approach

The ultimate goal of our work is to derive real counterexample traces for binary programs. To do so, our approach uses abstract interpretation to detect potential violations, and then derive paths through the program that could have led to that violation using backward analysis. These paths define test vectors, which are examined on the real hardware to filter spurious traces that have been introduced through over-approximation.

1.2 Contributions

Spurious warnings are a major issue when applying abstract interpretation in industrial practice. Typically, investigating spurious warnings relies on manual inspection of program invariants. The complex structure of embedded code makes manual inspection difficult and time-intensive. To leverage these issues in embedded-software verification, we contribute a framework that: (i) applies abstract interpretation to generate assertion-directed test cases; (ii) provides a link to the actual target hardware; (iii) automatically identifies spurious test traces.

2 Test-Case Generation Using Abstract Interpretation

Our framework (cf. Fig. 1) takes an executable binary file and a specification (cf. Sect. 2.1) as inputs. The binary file is ready to be run on the target hardware. After parsing, we build an initial control flow graph (CFG) of the binary and apply abstract interpretation (cf. Sect. 2.2) to derive program invariants. These invariants are used by the test-case generator to identify possible specification violations. Then, a backward analysis derives actual program inputs (cf. Sect. 2.3), that drive execution towards the specification violation. The test traces are then transferred to and executed on real hardware (cf. Sect. 2.4), i.e., an IP-core instance of the target microcontroller running within an FPGA embedded in its operating environment. A test-case monitor is attached to the IP core that tracks specification items during execution and provides runtime feedback.

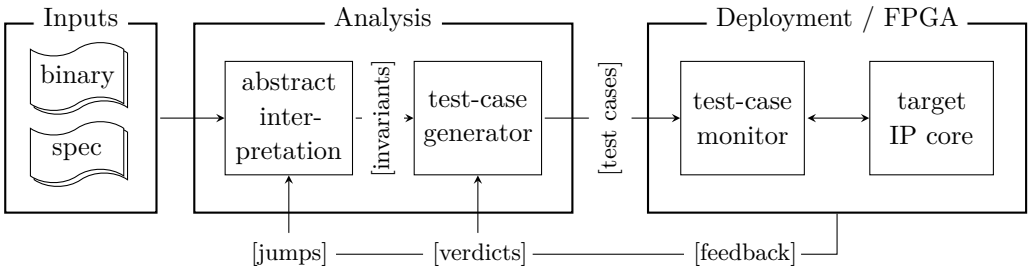

Figure 1 Framework overview

2.1 Specification Language

In the past, we have carried out a case study [18] in cooperation with an industry partner using [MC]SQUARE [21], which is a binary code verification tool. When confronting our partner with the full expressive power of temporal logics (CTL in this case), it turned out that it is particularly difficult for test engineers to translate their well-understood textual specifications into temporal logic formulas. Moreover, most specification items of the case study were local assertions (properties that hold at a specific program location) or global invariants (properties that hold at any program location), an observation also emphasized by Hoare [14, p. 10]. Consequently, to express program properties of interest, we propose a simple specification language, which is defined through the following grammar:

$$\begin{array}{lcl} \Psi & ::= & \mathcal{A}(pc, \varphi) \mid \mathcal{I}(\varphi) \\ \varphi & ::= & \mathsf{true} \mid \mathsf{false} \mid \neg\varphi \mid \varphi_1 \ \wedge \ \varphi_2 \mid \varphi_1 \ \vee \ \varphi_2 \mid \mathsf{AP} \end{array}$$

To express the semantics of this specification language, let a state of a program be a tuple $\langle pc, m\rangle \in \mathsf{Locs} \times \mathsf{Mem}$, where Locs is a finite set of program locations, and Mem represents the set of all possible memory configurations of the microcontroller. Then, the state space of the program is a subset of $\mathsf{Locs} \times \mathsf{Mem}$. The property φ is a predicate over memory locations $m \in \mathsf{Mem}$. Additionally, AP denotes the set of atomic propositions about memory cells in Mem. The satisfaction relation associated with φ is intuitively clear, following the standard inductive definition. If $m \in \mathsf{Mem}$ satisfies φ, we write $m \models \varphi$.

Properties, in turn, can be of local or global nature. A *local* assertion is a property $\mathcal{A}(pc, \varphi)$ attached to a certain program location $pc \in \mathsf{Locs}$. Given a set of states $S \subseteq \mathsf{Locs} \times \mathsf{Mem}$, then $\mathcal{A}(pc, \varphi)$ holds w.r.t. S iff $m \models \varphi$ for all $\langle pc', m\rangle \in S$ with $pc = pc'$. Similarly, a *global* invariant $\mathcal{I}(\varphi)$ holds iff $m \models \varphi$ regardless of pc'.

Our framework either reads a user-defined specification or uses existing assertions from the high-level representation of the program by parsing compiler-generated debug information.

2.2 Abstract Interpretation

The key idea in abstract interpretation is to simulate the execution of each concrete operation $g : C \to C$ in a program using an abstract analogue $f : D \to D$, where C and D denote the domains of concrete values and descriptions. Each abstract operation f is designed to model its concrete counterpart g in the following sense: If $d \in D$ describes a concrete value $c \in C$, then the result of applying g to c is described by applying f to d. Typically, the abstract operations are designed manually. However, handcrafting transformers for the complete instruction set of a microcontroller, which consists of more than 100 instructions, is time-consuming and error-prone. Consequently, we synthesize optimal transfer functions [19] from propositional encodings of the instructions' semantics using SAT solving [4]. The process of translating instructions into propositional Boolean formulas is often colloquially referred to as *bit-blasting*.

To derive a set of test cases, our abstract interpretation framework first computes invariants using intervals and synthesized transformers. If the invariants exhibit a potential property violation, we use backward analysis to derive a path (the test case) from the property violation to the start of the program. It is important to observe that sound abstract interpretation itself requires a CFG of the program to be available. However, recovering indirect control from binaries is a notoriously difficult problem [16]. Consequently, the CFG used in the abstract interpretation framework is incrementally extended using information gained through the test-case execution. Since the aim of our work is to detect test traces that

exhibit faulty behavior instead of proving correctness of an implementation, this approach is convenient. The remainder of this section discusses two approaches used to derive program invariants.

2.2.1 Affine transfer functions of basic blocks

The semantics of a microcontroller instruction can be encoded in propositional logic, which has become a standard technique in software verification, owing much to the advances in bounded model checking [3]. To illustrate this, consider the instruction `INC A` on an 8 bit architecture, which increments register `A` by one. The input and output values of `A` are represented by bit-vectors of length 8, denoted $\boldsymbol{a}$ and $\boldsymbol{a}'$, respectively. Then, the effects of applying `INC A` can be expressed propositionally, where $\boldsymbol{a}[i]$ denotes the i-th bit of $\boldsymbol{a}$ and $\oplus$ denotes the exclusive-or:

$$\texttt{INC A} := \bigwedge_{i=0}^{7} \left(\boldsymbol{a}'[i] \leftrightarrow \boldsymbol{a}[i] \oplus \bigwedge_{j=0}^{i-1} \boldsymbol{a}[j] \right)$$

Similar encodings can be derived for the entire instruction set [5]. The value of these encodings is that optimal transfer functions for either single instructions or whole sequences of instructions can be derived using successive calls to a decision procedure, in this case a SAT solver, prior to executing the actual analysis. Affine equalities [15] are systems of the form $\bigwedge_{i=0}^{m-1}(\sum_{j=0}^{n_i-1} \lambda_{i,j} \cdot v_j = d_i)$, where v_j are program variables and $\lambda_{i,j}, d_i \in \mathbb{Z}$, which can be used to describe relations between variables. Our approach derives optimal affine transformers for basic blocks from the Boolean encodings, using the algorithm developed by Brauer and King [4, Sect. 3.2]. As an example, consider the above instruction, and for brevity, let $\langle\!\langle \boldsymbol{a} \rangle\!\rangle = \sum_{i=0}^{7} 2^i \boldsymbol{a}[i]$. Then, we obtain the following affine system:

$$(\langle\!\langle \boldsymbol{a} \rangle\!\rangle \leq 254) \Rightarrow (\langle\!\langle \boldsymbol{a}' \rangle\!\rangle = \langle\!\langle \boldsymbol{a} \rangle\!\rangle + 1) \qquad (\langle\!\langle \boldsymbol{a} \rangle\!\rangle = 255) \Rightarrow (\langle\!\langle \boldsymbol{a}' \rangle\!\rangle = 0)$$

Using this representation, linear constraints — most notably octagons [17] — that distinguish inputs that lead to overflows are derived from the Boolean formulas. Otherwise, no affine relation between $\boldsymbol{a}$ and $\boldsymbol{a}'$ could be determined since, e.g., $254 + 1 = 255$ and $255 + 1 = 0$ in unsigned machine-arithmetic.

2.2.2 Local invariants through interval analysis

Interval analysis determines invariants using the computationally attractive interval abstract domain [8]. Let $\mathbb{N}^\star = \{0, \ldots, 255\}$ denote the set of numbers representable with a single 8-bit word. Then, a word-level interval is composed of $[a, b]$ with $a, b \in \mathbb{N}^\star$ and $a \leq b$. With $\top = [0, 255]$, $\bot = \emptyset$, and a *join* defined as $[a_1, b_1] \sqcup [a_2, b_2] = [min(a_1, a_2), max(b_1, b_2)]$, the domain forms a complete lattice.

To illustrate interval arithmetic, consider an `ADD A,B` instruction, summing the operands `A` with `B` and storing the result back to `A`. Suppose, we enter the instruction with the intervals $\texttt{A} = [12, 74]$ and $\texttt{B} = [10, 14]$, then we can derive that the resulting value in `A` will be within the interval $[12 + 10, 74 + 14] = [22, 88]$. These invariants are derived for each program counter location using fixed-point iteration and a combination with affine relations, following the reduction algorithm described in [5, Sect. 6]. More details are given in [6].

As a result, the analysis yields a list of word-level intervals over memory locations attached to every *pc* location, i.e., $\langle pc, (\texttt{A}[a_0, b_0], \texttt{B}[a_1, b_1], \ldots) \rangle$. These invariants are used to detect potential violations of the specification. For example, if the global invariant $\mathcal{I}(\texttt{A} < 25)$ should hold, then we identify all locations as potential violations that have intervals for `A` including valuations ≥ 25. The test-trace generation algorithm starts from these program locations.

2.3 Test-trace generation

Our algorithm starts from a program location where the specification may be violated, and systematically searches for traces that lead to this violation. Given an assertion Ψ and an invariant θ, we convert $\neg\Psi$ into a disjunctive normal form and treat $\neg\Psi \wedge \theta$ as the desired postcondition. Next, we apply the affine transfer function in reverse using *integer linear programming*, which gives us a precondition, and then, this step is iteratively applied for all possible predecessors, until the entry of the program is reached. The preconditions are computed in breadth-first order, which guarantees that shortest paths to the entry are found. For reasons of continuity, we defer the presentation of an example to Sect. 3.

2.4 Test-trace deployment and execution

A single test trace t is a path of program counter locations $\pi := \langle pc_0, \ldots, pc_n \rangle$ with $pc_i \in \mathsf{Locs}$ and a set of external inputs $In := \langle pc, i \rangle$ attached to certain program locations. For example, $In := \langle 0xC1C1, \texttt{p1} \leftarrow 0xB2 \rangle$ represents that $0xB2$ will be provided on I/O port `p1` at program counter location $0xC1C1$.

In our approach, we do not explicitly alter the code itself, nor do we insert additional event-triggers into the source code, which is a common practice in runtime verification [12]. Our monitoring is done by a hardware monitor unit, attached to an industrial IP core of the target microcontroller. The whole execution takes place on an FPGA, connected to the actual environment of the application. The monitor unit allows us to non-intrusively and on-the-fly monitor and track memory accesses of the microcontroller core. Besides, the monitor compares the current program counter with the expected one given in π. Whenever this comparison fails, we halt the microcontroller, mark t as *infeasible,* and load the next test trace, thus, subsequently ruling out spurious test traces. However, if the unexpected branch was caused by an indirect jump, we add the newly detected jump target to the CFG. In case the actual execution follows the predicted path π, the monitor will verify whether the specification items hold along the path (for global invariants) or on certain program locations (for local assertions).

3 Worked Example

Fig. 2 shows an embedded C code snippet and its CFG. The labels of the CFG nodes relate to the program counter locations on the left. The code reads a sensor value from an 8-bit input port and converts the value to its absolute value, storing the result in *val*. Next, a while loop is entered sending *val* times PWM pulses to the output and decrementing *val* each iteration. Whenever the predicate $val > 0$ is violated the assertion is reached and the loop starts again.

Based on a first intuition, the assertion will hold, regardless of the sensor values. The presumably positive variable *val* is decremented towards 0. Interestingly, the assertion does not hold under all inputs. Consider the binary sensor input $b1000000$, which corresponds to -128 in two's complement. The `ABS` macro will not alter the value since $-(-128) = -128$ due to the limited bit-width. It is obvious that the predicate $(-128 > 0)$ at the beginning of the while loop is false and the assertion does not hold.

Our algorithm starts by negating the predicate in the assertion, which gives $(\texttt{val} < 0) \vee (\texttt{val} > 0)$ in program location 6. The assertion has a single predecessor, i.e., node 3, for

```
#define ABS(a) (((a)<0)?-(a):(a))
char getSensor(void){return P1;}
void sendPWM(void){...;}

    int main(void){
      char val;
(1)   while(1){
(2)     val = ABS(getSensor());
(3)     while(val > 0){
(4)       sendPWM();
(5)       val--;
        }
(6)     ASSERT(val == 0);
      }
    }
```

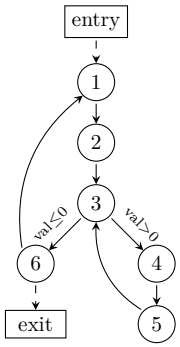

Figure 2 Example code (left) and CFG (right)

which we have derived the following transfer function:

$$\begin{aligned}
&(\texttt{getSensor()} \geq 0 \wedge \texttt{getSensor()} \leq 127) && \Rightarrow (\texttt{val}' = \texttt{getSensor()})\\
&(\texttt{getSensor()} \geq -127 \wedge \texttt{getSensor()} \leq -1) && \Rightarrow (\texttt{val}' = -\texttt{getSensor()})\\
&(\texttt{getSensor()} \geq -128 \wedge \texttt{getSensor()} \leq -128) && \Rightarrow (\texttt{val}' = -128)
\end{aligned}$$

The third one is examined, which gives us a test trace with inputs that lead to a violation of the assertion, namely $\pi = \langle 1, 2, 3, 6\rangle$; $In = \langle 2, \texttt{getSensor()} \leftarrow -128\rangle$ where the input in line 2 is -128. This test trace is executed on the IP core and the runtime monitor confirms that π is indeed a real counterexample trace.

4 Related Work

Test-case generation using formal methods, is an active area of research. Cousot and Cousot introduce abstract interpretation based program testing as *abstract testing* in [9], an approach closely related to our work. However, we apply abstract interpretation to machine code and offer a way to automatically rule out spurious counterexamples. Another popular approach is to use model checkers to derive test suites that comply with industrial coverage criteria [11]. With increasing complexity, these approaches suffer from similar problems as traditional model checking.

Wenzel et al. [22] describe cross-platform verification of embedded C code. Platform-specific C code is translated into semantically equivalent C code used by CBMC to generate counterexamples, which are executed on the host and on the target platform. Thus, their approach can find errors introduced by the compiler. Our approach is independent of the high-level implementation and does not require to instrument the code, which is vital for verifying timing properties. Deriving test data for machine code with a structural coverage goal is described in [2]. Their tool Osmose translates executable code to a generic assembly language and uses concolic execution for path exploration.

5 Discussion & Future Work

5.1 Summary

In this paper, we have addressed the question of deriving test cases from microcontroller binary code. Unlike other techniques, our approach uses abstract interpretation using a combination of different abstract domains to derive test cases directly from the executable

program code. The purpose of our work is not necessarily to derive test cases that satisfy certain coverage criteria, but rather to systematically infer paths that exhibit faulty behavior.

5.2 Future Work

In addition to the global and local assertions (cf. Sect. 2.1), we want to include time-bounded properties of the form $\Theta(\varphi_1, \varphi_2, \delta)$. Such properties state that if the predicate φ_1 holds then φ_2 must hold within $\delta \in \mathbb{N}$ clock cycles. Clearly, future efforts also include a case study showing the feasibility of our approach when applied to industrial embedded code.

Acknowledgements The work of Thomas Reinbacher and Andreas Steininger has been supported within the FIT-IT project CEVTES managed by the Austrian Research Agency FFG under grant 825891. The work of Martin Horauer has been supported within the FHplus project DECS managed by the Austrian Research Agency FFG under grant 811414. The work of Jörg Brauer and Stefan Kowalewski has been, in part, supported by the UMIC Research Centre of Excellence at the RWTH Aachen University.

References

1 G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What you see is not what you execute. In *VSTTE*, Toronto, Canada, 2005.

2 S. Bardin and S. Herrmann. OSMOSE: Automatic structural testing of executables. *Softw. Test., Verif. & Reliab.*, 2009.

3 A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

4 J. Brauer and A. King. Automatic abstraction for intervals using boolean formulae. In *SAS*, volume 6337 of *LNCS*, pages 167–183. Springer, 2010.

5 J. Brauer, A. King, and S. Kowalewski. Range analysis of microcontroller code using bit-level congruences. In *FMICS*, volume 6371 of *LNCS*, pages 82–98. Springer, 2010.

6 J. Brauer, T. Noll, and B. Schlich. Interval analysis of microcontroller code using abstract interpretation of hardware and software. In *SCOPES*. ACM, 2010.

7 E. M. Clarke and H Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 41–43. Springer, 2004.

8 P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming*, pages 106–130, April 1976.

9 P. Cousot and R. Cousot. Abstract interpretation based program testing. In *SSGRR*. Scuola Superiore G. Reiss Romoli, 2000. Invited paper.

10 E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *EMSOFT*, pages 255–264. ACM, 2008.

11 G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Softw. Test., Verif. & Reliab.*, 19(3):215–261, 2009.

12 K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Form. Methods Syst. Des.*, 24(2):189–215, 2004.

13 M. P. E. Heimdahl, D. George, and R. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *HASE*, pages 178–186. IEEE, 2004.

14 C.A.R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25:14–25, 2003.

15 M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

16 J. Kinder, H. Veith, and F. Zuleger. An abstract interpretation-based framework for control flow reconstruction from binaries. In *VMCAI*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

17 A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

18 T. Reinbacher, M. Horauer, B. Schlich, J. Brauer, and F. Scheuer. Model checking assembly code of an industrial knitting machine. In *EM-Com*, pages 97–104. IEEE, 2009.

19 T.W. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 252–266. Springer, 2004.

20 RTCA/DO-178B. Software considerations in airborne systems and equipment certification, 1992. Washington DC, USA.

21 B. Schlich. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Aachen, Germany, June 2008.

22 I. Wenzel, R. Kirner, B. Rieder, and P. Puschner. Cross-platform verification framework for embedded systems. In *SEUS*, pages 137–148. Springer, 2007.