Formal Verification of Machine-Code Systems by Translation of Simulable Descriptions

Jan Onderka
Czech Technical University in Prague
Faculty of Information Technology
onderjan@fit.cvut.cz

***Abstract*—Correctness of safety- and security-critical systems can be ensured by formal verification. While formal verification of source-code and hardware systems is widely used, existing approaches to formal verification of machine code are severely limited in verification strength and processor choice. In this paper, a formal verification tool written in the Rust programming language is introduced, with emphasis on machine-code verification. The problems of previous tools are circumvented by abstraction refinement, an advanced formal verification technique. System behaviour is defined using simulation descriptions written in a limited subset of Rust, allowing straightforward extension of the tool to new architectures and processor model using architecture and processor manuals. A novel technique of translating the descriptions to their verification analogues at compile-time enables fast verification using abstraction refinement without any need for formal verification knowledge by the description writer.**

***Index Terms*—machine code, embedded systems, formal verification, model checking, Rust**

I. Introduction

A large amount of embedded devices in our lives is safety- and security-critical. Many modern microcontrollers are designed around safety (such as the ARM Cortex-R series) or security (such as ARM Cortex-M33). Considerable efforts are being spent to verify that both the hardware and the source code are correct. Verification is either informal, possibly finding a bug but giving no guarantees, or formal, guaranteeing that the system upholds properties given by a specification.

Even though formal verification is crucial for creating safe and secure systems, to the author's knowledge, there have been only abortive attempts at formal verification of machine-code systems, i.e. systems comprised of a processor (in embedded devices, typically a microcontroller) and the executed program in the machine language of the processor. There is much difficulty in tackling the problem: the machine-code program is typically highly complicated and lacks constructs beyond the instruction level, and its behaviour is dependent on the processor's behaviour, complicated by itself.

Unfortunately, some properties are impossible or infeasible to verify by other means than by formal machine-code verification (which is visualised in Figure 1), mainly relating to peripheral and memory manipulation dependent on the device and its configuration. Even if the problem is introduced in source code, formal verification of machine code can reveal the possibility of a bug due to the possibility of unexpected memory manipulation, potentially preventing a critical, hard-to-fix vulnerability. While some vulnerabilities can be fixed by Over-the-Air updates, other vulnerabilities may be unfixable even if discovered. In an extreme case, a security vulnerability in manufacturer-provided read-only machine code affecting a whole line of secure-by-design microcontrollers was unpatchable in the already produced microcontrollers [1].

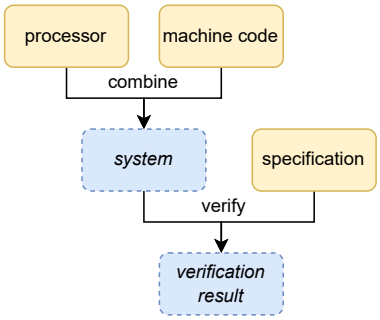

Fig. 1. A high-level overview of formal verification of machine-code systems. The solid yellow cells represent inputs, while the dashed blue cells represent automated results. The processor and machine code are combined to form the system under verification. It is then determined if the specification holds.

The author has developed **machine-check**, a formal verification tool written in the Rust programming language. The tool can verify Computation Tree Logic properties [2] of arbitrary finite-state machines but is tailored especially for machine-code verification in microcontroller-based systems. Unlike previous model-checking tools, **machine-check** is free, open-source, and publicly available[1]. To avoid problems encountered by the previous tools, the microcontroller behaviour is specified by finite-state machine *simulable descriptions* written in the Rust language, which are then automatically translated to their verification analogues. This means the microcontroller description writer does not need any formal verification knowledge, but the verification still benefits from advanced formal verification techniques implemented in **machine-check**. In this paper, the specifics of simulation descriptions in **machine-check** will be given using a simple RISC microcontroller example, showing the benefits of the novel translation to verification analogues.

[1]The latest version is available from https://crates.io/crates/machine-check. The version discussed in this paper is available from https://crates.io/crates/machine-check/0.2.0.

II. Previous Work

The most important previous research into machine-code verification for embedded systems was the [mc]square project (later renamed to Arcade.µC) [3]–[6]. The verifier was initially tailored to a specific microcontroller, leading to further research into support for other microcontrollers by creating state space generators from system descriptions [7]. However, that approach was limited by the necessity of using a specific newly designed language and a requirement for the description writer to tailor the description to use formal verification techniques. While the Arcade.µC project is now defunct, the publications inspired further research [8]–[10]. There were also other approaches to machine-code formal verification unrelated to Arcade.µC [11], [12]. However, no previously developed tools seem to be publicly available.

As for formal verification techniques used for machine-code systems, the predominant approach is *model checking*, where a graph of successive states (the *state space*) corresponding to the system under verification is constructed before verifying the state space against the specification [13]. However, as even simple microcontrollers contain kilobytes of uninitialized memory, it is infeasible to represent the whole state space explicitly. Therefore, advanced machine-code verification tools use *abstraction*, representing multiple concrete states by one abstract state. For example, in three-valued abstraction [4], each abstract state bit can be zero, one, or *unknown* (X), which means the abstract state represents both the concrete states where the bit is zero and ones where the bit is one. Such abstraction allows uninitialized memory and inputs to have little effect on state space size, but many properties of the system become unverifiable due to loss of information (the verification becomes *incomplete*). Incompleteness can be counteracted by *abstraction refinement*, which deductively reduces the amount of abstraction, but brings yet another layer of difficulty to tool development. As such, previous machine-code verification tools did not use abstraction refinement or used only simple versions of it, allowing incompleteness and reducing tool usefulness.

III. Verifying Systems with machine-check

The high-level overview of machine-code verification via **machine-check** is visualised in Figure 2. The simulable processor description, written in Rust code, is translated to verification analogues (allowing usage of abstraction with refinement during the formal verification), which are compiled together with algorithms that control the verification process. The machine code and specification are provided as arguments to the resultant executable. As such, the verification is faster and uses less memory than if the system was interpreted, yet allows for flexible, iterative development of the machine code and specification. The verifier executable can also be used on a dedicated server without installing the Rust language ecosystem.

Simulation description and machine code examples will be given later in Section IV. As for the specifications, only a cursory description will be given here due to the scope of the paper.

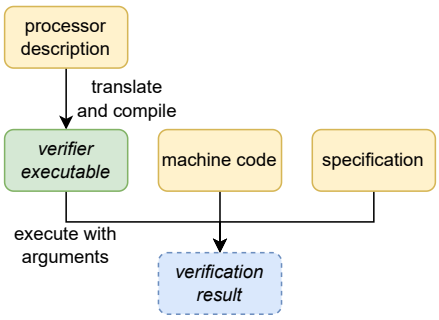

Fig. 2. A high-level overview of **machine-check** machine-code system verification process. The processor description is translated to verification analogues, then compiled together with verification control algorithms to form a verifier executable for the given processor, visualised in a solid green cell. The verifier is executed with the machine code and specification given as arguments, performing formal machine-code verification as in Figure 1. The compilation step ensures speed gain over interpretation.

Currently, Computation Tree Logic (CTL) [2] specifications are supported. Essentially, in addition to expressions on the states of systems, CTL can also be used to specify a variety of causal relationships between states of the system. A few examples of what can be expressed using CTL follow:

- **Safety.** A reserved register is never written to.
- **Stabilization.** Once a peripheral is enabled, it remains so.
- **Action-reaction.** Once a button is pressed (input pin becomes low), an LED toggles (output pin becomes negated).
- **Recovery.** It is always possible to return to the start of the main program loop with some sequence of inputs, i.e. the system never becomes irrecoverably stuck.

The verification result is a yes-no answer of whether the specification holds for the system. By design, **machine-check** is complete, always producing the yes-no answer in finite time (although the needed computation time and memory may be impractical for some combinations of system and specification). If necessary, the user can inspect the abstract state space to determine the cause of a bug.

IV. Processor Descriptions

The simulable descriptions in **machine-check** are designed to make describing microcontroller-based systems simple. Even so, real architectures are still time-consuming to implement due to the size of the instruction set. For example, the AVR ATmega328P microcontroller was described in ∼3000 lines of code, with simple peripheral support only. Fortunately, once coded, the vast majority of the description can be reused for other similar microcontrollers with the same architecture.

A simulable description of a very simplified RISC microcontroller[2] is shown in Figure 3. The description is written in a subset of valid Rust code, using specially provided **machine-check** types for simple transcription of behaviour from datasheets. The machine-code system can be immediately simulated in Rust

[2]The whole description is available at https://docs.rs/crate/machine-check/0.2.0/source/examples/simple_risc.rs.

by constructing the `System` structure, with the machine code under simulation contained in field `progmem`, and using the `init` and `next` functions to generate successive states using a given sequence of inputs.

While simulation is performed with a single input sequence, all input sequences must be considered for formal verification. Since each successive state only depends on the previous state and input, it is possible to generate a graph of successive states (i.e. the state space) that completely captures the system behaviour. However, as discussed in Section II, this is infeasible in practice. As such, the `machine_description` macro in Figure 3 automatically generates verification analogues of the machine, allowing the use of advanced abstraction-refinement techniques. In case the description code does not conform to the subset of Rust processable by **machine-check** translation, a compilation error is issued so the problem can be fixed.

Each system has specific construction parameters. For example, classic finite-state machines are constructed without any parameters, while machine-code systems must be provided with the machine code, with varying specifics such as instruction length and number of instructions. As such, in **machine-check**, constructing the system is the responsibility of the description writer. For machine-code systems, the intended approach is to read the machine code from a file given as an argument to the verifier. However, for conciseness, in Figure 4, the example system from Figure 3 is constructed with a hard-coded toy machine-code program. The constructed system is handed off to the main routine of **machine-check** afterwards, which verifies a specification obtained from arguments to the executable. As such, properties of the system obtained by compiling the code from Figures 3 and 4 can be formally verified. For example:

- Register 1 is set to 1 before the main loop is reached.
- It is always possible to reach program location 9.
- Program locations above 9 are never reached.

The properties are verified nearly instantaneously (< 1 s) and with insignificant memory usage. In comparison, formal verification by state space construction would require constructing more than $2^{2^8} = 2^{256}$ states, which is completely infeasible.

V. Future Work

Currently, **machine-check** is still in an experimental stage. While the underlying approach is general and flexible enough to build on, there is still work to be done before it is truly ready for industrial and academic use.

Full stabilization of machine descriptions and specifications is paramount. The specification format in particular is currently very rudimentary and requires a reasonable redesign so that it is easy to write both simple and complex specifications. To comfortably support compiled programs, it is also necessary to add the ability to refer to their debug symbols within specifications. The author plans to release an initial stable version of **machine-check** once these problems are resolved.

A more interactive and helpful verification process is also planned, so that the reason why a specification does not hold can be discovered quickly and easily.

```
1  #[machine_check::machine_description]
2  mod machine_module {
3    pub struct Input {
4      gpio_read: BitvectorArray<4, 8>,
5      uninit_reg: BitvectorArray<2, 8>,
6      uninit_data: BitvectorArray<8, 8>,
7    }
8    impl ::machine_check::Input for Input {}
9    pub struct State {
10     pc: Bitvector<7>,
11     reg: BitvectorArray<2, 8>,
12     data: BitvectorArray<8, 8>,
13   }
14   impl ::machine_check::State for State {}
15   pub struct System {
16     pub progmem: BitvectorArray<7, 12>,
17   }
18   impl ::machine_check::Machine for System {
19     type Input = Input;
20     type State = State;
21     fn init(&self, input: &Input) -> State {
22       State {
23         pc: Bitvector::<7>::new(0),
24         reg: Clone::clone(&input.uninit_reg),
25         data: Clone::clone(&input.uninit_data),
26       }
27     }
28     fn next(&self, state: &State, input: &Input)
29       -> State {
30       let instruction = self.progmem[state.pc];
31       let mut pc = state.pc + Bitvector::<7>::new(1);
32       let mut reg = Clone::clone(&state.reg);
33       let mut data = Clone::clone(&state.data);
34       ::machine_check::bitmask_switch!(instruction {
35         "00dd_00--_aabb" => { // add
36           reg[d] = reg[a] + reg[b];
37         }
38         "00dd_01--_gggg" => { // read input
39           reg[d] = input.gpio_read[g];
40         }
41         "00rr_1kkk_kkkk" => { // jump if bit 0 is set
42           if reg[r] & Bitvector::<8>::new(1)
43             == Bitvector::<8>::new(1) {
44             pc = k;
45           };
46         }
47         "01dd_kkkk_kkkk" => { // load immediate
48           reg[d] = k;
49         }
50         "10dd_nnnn_nnnn" => { // load direct
51           reg[d] = data[n];
52         }
53         "11ss_nnnn_nnnn" => { // store direct
54           data[n] = reg[s];
55         }
56       });
57       State { pc, reg, data }
58     }
59   }
60 }
```

Fig. 3. Example description of a simplified Harvard-architecture RISC microcontroller as a finite-state machine. Less important code details are omitted for clarity.
Input, state, and system structures are defined on lines 3–17. Power-of-two array sizes and bit-vector lengths are determined by generic constants, so e.g. the register array `reg` contains $2^2 = 4$ registers, each containing 8 bits.
On lines 18-59, finite-state-machine behaviour is described by `init` and `next` functions. In Rust, if the last statement in a function is not terminated by a semicolon, it is the return value. Both functions return newly constructed states. The `init` function returns a state with the program counter set to zero and other fields uninitialized (having arbitrary values). The function `next` reads current instruction from read-only program memory, increments the program counter, and decides on the action to perform depending on the instruction value. The `bitmask_switch` macro is designed to have the same format as conventional instruction set descriptions, filtering on zeros and ones and creating new variables for letters.

```
fn main() {
  let toy_program = [
    // (0) set r0 to zero
    Bitvector::new(0b0100_0000_0000),
    // (1) set r1 to one
    Bitvector::new(0b0101_0000_0001),
    // (2) set r2 to zero
    Bitvector::new(0b0110_0000_0000),
    // --- main loop ---
    // (3) store r1 content to data location 0
    Bitvector::new(0b1100_0000_0000),
    // (4) store r2 content to data location 1
    Bitvector::new(0b1100_0000_0001),
    // (5) read input location 0 to r3
    Bitvector::new(0b0011_0100_0000),
    // (6) jump to (3) if r3 bit 0 is set
    Bitvector::new(0b0011_1000_0011),
    // (7) increment r2
    Bitvector::new(0b0010_0000_1001),
    // (8) store r2 content to data location 1
    Bitvector::new(0b1110_0000_0001),
    // (9) jump to (3)
    Bitvector::new(0b0001_1000_0011),
  ];
  let mut progmem = BitvectorArray::new_filled(
    Bitvector::new(0));
  for (index, instruction) in toy_program
    .into_iter().enumerate() {
    progmem[Bitvector::new(index as u64)] = instruction;
  }
  let system = machine_module::System { progmem };
  machine_check::run(system);
}
```

Fig. 4. Example of construction of a machine-code system based on the simplified RISC processor from Figure 3. Here, the program memory values are hardcoded, but arbitrary Rust code can be used to construct the system. The constructed system is handed off to **machine-check**, which verifies properties determined by command-line arguments against the system.
Considering data locations 0 and 1 to be memory-mapped peripherals (e.g. general-purpose outputs), the output behaviour of the program is that the locations are set to zero on initialization, after which the data location 1 is incremented each time bit 0 of input location 0 is read as set.

Finally, more advanced types of reasoning could be added to **machine-check**, so that even more complex systems and specifications can be verified in reasonable time. Fortunately, due to the strict separation between the descriptions and specifications on one side and the verification internals on the other, such internal improvements can be made without breaking the description and specification code.

VI. Conclusion

In the paper, the concept of formal verification was introduced, with an observation that there are no tools publicly available for formal verification of machine-code systems despite the need for verification of properties that are hard or impossible to verify by source-code or hardware verification. Previous work on machine-code formal verification was summarized and the problems of previous approaches were discussed. A free, publicly available tool for verification of machine-code systems **machine-check** was introduced. The shortcomings of previous approaches are resolved by a novel combination of abstraction refinement and compile-time translation of simulation descriptions. The descriptions were studied in detail using an example of a simplified RISC processor. Finally, future work required to make the tool usable for common use was discussed.

Acknowledgment

This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS23/208/OHK3/3T/18.

References

[1] "LPC55Sxx SB2 loader vulnerability," 2022, CVE-2022-22819. [Online]. Available: https://community.nxp.com/t5/LPC-Microcontrollers-Knowledge/LPC55Sxx-SB2-loader-vulnerability/ta-p/1433661

[2] E. A. Emerson and E. M. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons," *Sci. Comput. Program.*, vol. 2, no. 3, pp. 241–266, 1982.

[3] B. Schlich and S. Kowalewski, "[mc]square: A model checker for microcontroller code," in *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, 2006, pp. 466–473. [Online]. Available: https://doi.org/10.1109/ISoLA.2006.62

[4] T. Reinbacher, M. Horauer, and B. Schlich, "Using 3-valued memory representation for state space reduction in embedded assembly code model checking," in *2009 12th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2009, pp. 114–119. [Online]. Available: https://doi.org/10.1109/DDECS.2009.5012109

[5] B. Schlich, "Model checking of software for microcontrollers," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 4, Apr. 2010. [Online]. Available: https://doi.org/10.1145/1721695.1721702

[6] T. Noll and B. Schlich, "Delayed nondeterminism in model checking embedded systems assembly code," in *Hardware and Software: Verification and Testing*, K. Yorav, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 185–201. [Online]. Available: https://doi.org/10.1007/978-3-540-77966-7_16

[7] D. Gückel, "Synthesis of state space generators for model checking microcontroller code," Dissertation, Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen, November 2014. [Online]. Available: http://aib.informatik.rwth-aachen.de/2014/2014-15.pdf

[8] Y. Wu and S. Yamane, "Model checking of real-time properties for embedded assembly program using real-time temporal logic RTCTL and its application to real microcontroller software," *IEICE Transactions on Information and Systems*, vol. E103.D, no. 4, pp. 800–812, 2020. [Online]. Available: https://doi.org/10.1587/transinf.2019EDP7172

[9] Y. Wu, H. Kamide, and S. Yamane, "Software model checking for real-time properties of embedded assembly programs based on lazy abstraction and refinement," in *9th IEEE Global Conference on Consumer Electronics, GCCE 2020, Kobe, Japan, October 13-16, 2020*. IEEE, 2020, pp. 62–65. [Online]. Available: https://doi.org/10.1109/GCCE50665.2020.9291966

[10] T. Kiriyama, Y. Wu, and S. Yamane, "Reduction of timer interrupts for embedded assembly programs based on reduction of interrupt handler executions," in *10th IEEE Global Conference on Consumer Electronics, GCCE 2021, Kyoto, Japan, October 12-15, 2021*. IEEE, 2021, pp. 464–466. [Online]. Available: https://doi.org/mqzw

[11] E. G. Mercer and M. Jones, "Model checking machine code with the GNU debugger," in *12th International SPIN Workshop*, ser. Lecture Notes in Computer Science, vol. 3639. San Francisco, USA: Springer, August 2005, pp. 251–265. [Online]. Available: https://doi.org/10.1007/11537328_20

[12] T. Mehler, "Challenges and applications of assembly-level software model checking," Ph.D. dissertation, University of Dortmund, 2006. [Online]. Available: http://doi.org/10.17877/DE290R-8397

[13] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*, 1st ed. Springer Publishing Company, Incorporated, 2018.