embeddedworld2021 Exhibition&Conference ... it's a smarter world DIGITAL

Testing, Model Checking and Static Analysis

Dream Team or Rivals?

Sebastian Krings
Axivion GmbH
Nobelstr. 15
70565 Stuttgart, Germany
krings@axivion.com

***Abstract*—Ensuring reliability and quality of software has become a necessity. This is especially true for safety critical systems. To do so, different techniques have been established in industry and academia. For instance, coding standards such as the MISRA ruleset or the AutosarC++14 ruleset have been developed. Additionally, software testing, static analysis and even model checking and formal proof are used. These approaches differ with respect to the effort needed and the environments they can be used efficiently in.**

In this paper, we will present a simple case study. Following the development of a light and speed control system, we will highlight how and when verification techniques can be employed and how selected requirements can be translated into properties to be checked. In particular, we outline how to employ the mentioned techniques, how they work and which results to expect from them. Furthermore, we highlight strengths and weaknesses of the different methods and discuss possible combinations.

Keywords—component; formatting; style; styling; insert (key words)

I. Introduction

With the ongoing digitalization, the trend towards the internet of things, industry 4.0, and the ongoing integration of embedded devices into everyday objects, our dependency on software components increases by the hour. From cars to trains to medical equipment, software components are more safety critical than ever. Or, as Marc Andreessen put it: "Software is eating the world" [1]. As a consequence, ensuring both reliability and quality of software has become a necessity. This is especially true for safety critical systems, where a malfunction can directly endanger users as well as innocent bystanders. However, even for software not deemed safety-critical, malfunction can and has led to enormous damage and financial losses [2].

In consequence, several techniques for ensuring the proper functionality of software have been developed and are in active use in industry as well as being actively researched in academia. In the following, we will use a simple case study to highlight some key techniques for verification and validation and discuss their application to an embedded software product.

II. Case Study

The case study used as an example throughout this paper is based on the ABZ 2020 case study [3]. It describes two assistant systems commonly found in modern cars. The overall system consists of two loosely coupled components, namely an adaptive exterior light system (ELS) and a speed control system (SCS). The ELS controls head- and taillights, setting their brightness depending on the surroundings and user preference. At the same time, the SCS controls the vehicle's speed, again by considering the environment as well as parameters given by the driver. Obviously, both are safety critical components, rendering safety and security a development priority.

Both components are to be developed in C, following the MISRA C 2012 coding guidelines [4]. A (simplified) component diagram, taken from [5], is shown in Fig. 1. As you can see, both ELS and SCS are supposed to be realized as continuous loops of reading sensors, executing their particular functionality and storing the last state for future reference. This might be needed to compute timer offsets or react to changing user inputs.

Both systems receive inputs from the driver via the steering wheel and various other actuators. Further inputs are provided by multiple sensors, e.g., for speed or brightness. The system contains a clock and might have to fulfill various requirements regarding real-time operations. Finally, the systems provide output to the actors controlling the car's light and speed.

III. Testing, Model Checking and Static Analysis

In the following section, we will outline how common verification techniques could be used to validate and verify the correct operation of ELS and SCS.

A. Testing

Testing is perhaps the most commonly known technique for asserting the quality of a software system. Simply speaking, testing tries to ensure the correct behavior of software by executing it on a set of input data. After execution, the results are compared to a set of expected results.

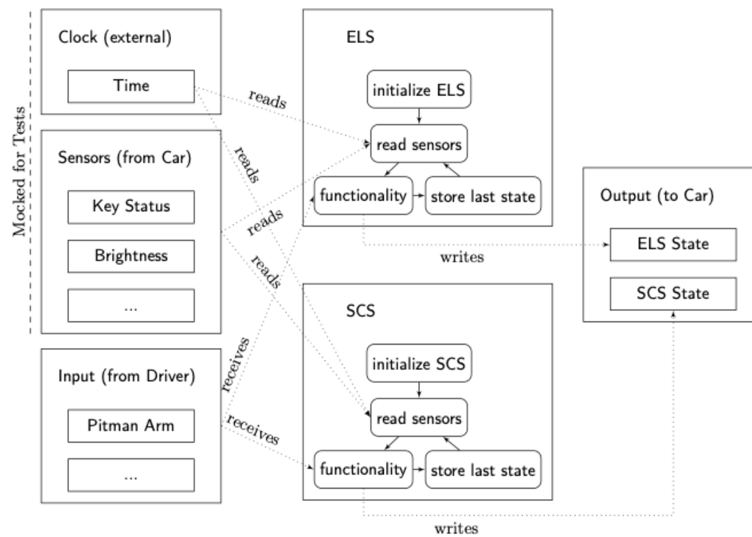

Fig. 1. System Components

For our case study, consider a requirement such as "If the ignition is 'On' and the light rotary switch is in the position 'On', then low beam headlights are activated". A test case verifying this requirement could consist of several steps:

1. Initialize the system,
2. Drive the system into an appropriate state, i.e., one where the ignition runs but the rotary switch has not yet been turned or vice-versa,
3. Virtually turn the switch to simulate user input,
4. Wait to allow the system to react,
5. Check the control output for the headlights and compare it to the expected activation.

For thorough and extensive testing to be feasible (early in the development cycle), parts of the system are mocked. That is, they are replaced by a placeholder mimicking the original behavior to some extent. At the same time, the mocked version is supposed to be easier to handle and to behave more invariant.

For our case study, one could decide to mock away the (external) clock, simplifying the test cases by abstracting away from timing issues. Furthermore, the actual sensors of the car could be replaced by some software component which just emits the desired sensor readings. The components to be mocked are marked by the dashed line in Fig. 1.

Especially in the automotive and railway domains we might also see hardware-in-the-loop tests, where the software is tested on the (embedded) device build into the actual surrounding hardware (i.e., car or train). The key idea here is to replace artificial test environments and mockups as much as feasible by real usage scenarios.

Major obstacles for thorough testing are the number of test cases needed and gaining appropriate test data. As you can see in the example above, the system has to be driven into a certain state for the test to be meaningful. This could be achieved by settings its state variables directly or by executing an appropriate set of actions that will lead the system to the desired state. Either way, the test will never consider other ways to reach the state or situations different from the one set up. As a consequence, only a (possible narrow) part of the system's behavior is indeed tested. Different coverage criteria are available to assert the degree of test coverage and to identify areas where further testing might be needed.

In addition, gaining test data for software tests is notoriously hard. Typical limitations include lack of properly formulated requirements or impractically large amount of test data needed to cover the system under test. For larger applications, difficulties stem from the amount and quality of test data available and the volume of data needed for realistic testing scenarios [6]. To date, manufacturers have to put enormous efforts into testing their systems under various environments. Artificial test data might not be diverse enough to enable desired test cases [7], whereas the use of real data might be prohibited due to security or privacy concerns or other regulations such as the GDPR or ISO 27001.

B. *Model Checking*

The key idea behind model checking is to systematically explore all possible states a software system might be in [8]. At the same time, for each state an analyzer discovers, a set of given properties is checked. As an example, consider the following source code.

```
extern int some_value();
int main {
    int c = 1;
    c += some_value();
    …
}
```

A simple model checker would try to compute the set of all reachable states, called the state space. This is done iteratively, e.g. by building up a graph structure. After the assignment of `c`

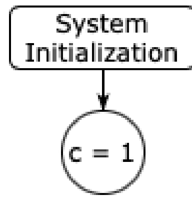

Fig. 2. Intermediate State-Space before Addition

we would end up with the intermediate graph depicted in Fig. 2. The next line, in which `c` is incremented by an (as far as we know) arbitrary value now shows where simple testing and model checking differ. Running test cases just executes the code and thus computes a single resulting value for the variable `c`. Model checking however would now follow all possible executions, each extending the state space. As a result, we end up with the graph show in Fig. 3.

Clearly, this can quickly lead to a blow-up, rendering simple approaches to model-checking infeasible for non-trivial systems. To mitigate the blow-up, different approaches and extensions have been suggested. These range from avoiding to explore symmetrical paths to symbolic computations [8], with further approaches in active research. However, the sheer size of state spaces of modern software systems often limits the applicability of model checking.

Regarding our case study, let's have a look at one of the requirements again. For the ELS, one of the given requirements might be that "Whenever the low or high beam headlights are activated, the tail lights are activated, too". As you can see, this requirement cannot be fully verified by testing alone, as we might never be sure to cover all possible situations in which the low or head beam highlights are activated.

Using a model checker, the requirement can be formulated as an invariant, i.e., a property that has to hold for all states on all execution paths. Let's assume we have a struct `state`, holding the current brightness levels. We can then formulate the requirement as a number of C-style assertions[1]:

```
assert(implies(state.lowBeamLeft > 0,
               state.tailLampLeft > 0 &&
               state.tailLampRight > 0));
```

A model checker such as CBMC [9] now searched for counter-examples, i.e., states in which the property does not hold. If such a state is found, the path to the state is reported to the user. However, this can only seldom be used to prove the absence of errors. As the illumination level of the two low beams and the two tail lamps are stored as a percentage, there are already

$$100^4 = 100.000.000$$

combinations of light levels to be checked. Each of those could be reached on different and perhaps multiple paths. Additionally, the system's state does not only consist of these four variables. For instance, even though not directly

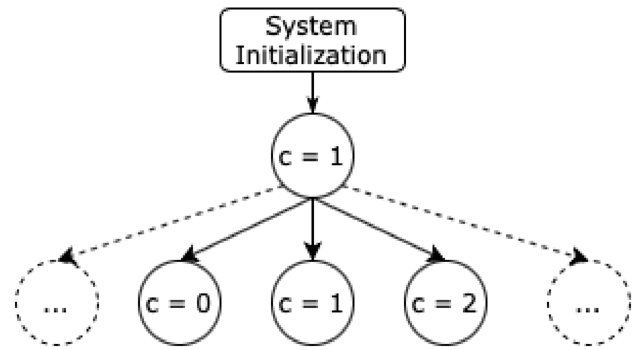

Fig. 3. Intermediate State-Space after Addition

responsible for the implementation of the desired functionality, the number of variables used for product configurations or product variants alone can be very large. By including additional variables, e.g., for direction lights, the system's state-space grows even larger, highlighting the combinatorial blow-up discussed above. In case the implementation uses a running timer or clock, the state-space might even grow infinitely.

C. Static Analysis

As outlined in the two preceding sections, both testing and the more systematic model checking have their weaknesses. Independent of the coverage metric used, tests usually cannot cover all possible scenarios. At the same time, complete exploration of the state space of any non-trivial software via model checking is often technically infeasible.

Static analysis bundles multiple approaches to software analysis. All have in common, that they avoid code execution. Instead, properties of interest are derived from the source code itself or an intermediate representation. Different static analysis techniques include

- Style checking or linting, i.e., searching the source code for discouraged constructs,
- Data flow and pointer-based analyses, where the analyzer tries to infer possible values of variables and possible targets of pointer at certain program points. This could be used to detect dereferences of invalid pointers.
- Abstract interpretation, in which the source code's behavior is overapproximated by executing an abstracted version of the instructions on an abstract representation of the data used. As an example, an abstract interpreter might abstract all integers to either be zero, positive or negative and compute the effect of an addition accordingly.
- Symbolic techniques, where the source code is represented as some kind of mathematical formula, which is then evaluated.

While these techniques are able to cope with much larger systems, the increased applicability comes at a price. In

---

[1] `Implies` is a special macro provided by the CBMC model checker. Basically, it just unrolls the implication to shorthand `||` and `&&`.

particular, with increasing distance to the actual execution, precision issues start to appear. For instance, as pointer-based static analyses can never be precise, one often has to decide between over- and underapproximations. This leads to the possibility of both false positive as well as false negative reports. Furthermore, static analysis tools often have to find a good tradeoff between precision and speed of execution.

Regarding the case study, static analysis can be used to ensure we indeed follow the MISRA coding guidelines throughout the implementation. Furthermore, a static analyzer could pinpoint many programming errors which are not explicitly given as requirements: absence of null-pointer dereferences, absence of division by zero or proper usage of locking mechanisms. In addition, the implementation could be compared to the architecture outlined in Fig. 1, verifying if we adhere to the desired calling and communications relationships.

As an example, the check of MISRA C 2012 Rule 17.7 [4] uncovered a possible problem in our initial implementation. The rule states that a value returned by a function call shall be used. As can be seen in the detail view provided by the Axivion Suite shown in Fig. 4, we did not use the return value of `pthread_create`. This is bad practice, since the return value is used to communicate whether a new thread has indeed been started (returns `0`) or whether an error occurred (returns error code).

When it comes to the functional requirements, a static analysis could be used to verify properties such as "After engine start, there is no previous desired speed. The valid values for desired speed are from 1 km/h to 200 km/h.". To verify if the

Details @ 2021-01-29 07:36:25

Violation Version	2021-01-29 07:36:25 (2021-01-29T07:36:25.17 Z)
Id	abz-2020-case-study:SV11
Owners	NOBODY
Justification	
Location	main.c:42:19
Provider	axivion
Severity	required
Error Number	MisraC2012-17.7
Message	Return value of function discarded.
Entity	pthread_create()
Additional SLocs	/Library/Developer/CommandLineTools/SDKs/MacOSX11.1.sdk/usr/include/pthread/pthread.h:329:5

Fig. 4. Axivion Report

implementation adheres to the specification, possible values of the variable holding the desired speed could be inferred using one of the techniques mentioned above. Afterwards, these values could be compared to the desired interval.

D. *Formal Proof and Deductive Verification*

While all approaches introduced so far are very helpful for verifying software safety, they all leave a certain doubt. Either they are unable to fully cover the program under test or they might suffer from false positives / negatives due to imprecision. With formal proof and deductive verification, the key idea is to proof a software's adherence to its specification with mathematical rigor.

This is done by formulating mathematical models of the software to be analyzed and the properties to be verified. Afterwards, different methods ranging from pen-and-paper proofs to (semi-) automatic proof tools can be used to verify if the desired properties indeed are guaranteed to hold.

Even though the impact of formal proof to general industrial software and system development is low so far, it has been successfully employed in different high-risk areas:

- In avionics software by Airbus [10],
- In railway engineering and signaling by RATP, operator of the fully autonomous Paris Line 14, and many other companies [11],
- For cryptographic and communication protocols,
- Microkernels,
- Medical devices, and
- (autonomous) vehicles.

While formal proof provides an extremely high level of confidence, it also abstracts away from the original implementation that might not adhere to its mathematical representation as much as we think. Or, as Donald Knuth remarked [12]: "Beware of bugs in the above code; I have only proved it correct, not tried it."

IV. Combining Techniques

As the previous section has shown strengths and weaknesses of the verification methods discussed, one might think about using all of them in conjunction. While this is a valid way to overcome the individual limitations, it drives up computation time and resource consumption. Furthermore, this approach only helps uncover the set of all errors uncovered by the individual methods.

To improve, different promising integrations of the techniques into unified verification procedures could be imagined. For instance, the results of a coarse data flow analysis could be used to identify interesting value ranges to be later examined in detail by testing or model checking techniques.

At the same time, static techniques could be used to show the absence of some kinds of errors for parts of the program. This would allow to skip costly execution and testing steps.

Furthermore, combinations of different static verification techniques have been developed and are in active use. For instance, errors reported by an imprecise data flow analysis could serve as an input for a costly but more precise symbolic analysis. In this setup, the costly analysis would only run if we have an initial suspicion. It could then be used to disprove the existence of an error, thus reducing the number of false positives.

At Axivion, we recently started integrating our static analysis tools with the unit and integration testing solution VectorCAST by Vector Informatik. Currently, the integration is at the level of reciprocal information exchange between the tools for simplifying the access to analysis reports as we only mutually report findings for presentation in the tools' dashboards.

However, deeper integration steps that could boost the effectiveness and the performance of both the static analysis as well as the testing tools are planned for future releases.

V. CONCLUSION

Software checking and verification techniques have come a long way from academia to industry and are now in widespread use. However, new techniques are still explored and combinations of different algorithms are still an active research topic.

In particular, the ever-increasing complexity of software systems poses a challenge to analysis tools. Simultaneously, more businesses rely on software, pushing the boundary of systems that neeed to be considered mission critical. At the same time, the increasing demand for software security and its analysis results in new applications areas for verification tools.

As we have shown, there is no one-fits-all solution for software verification. Rather, these gathering demands will lead to further refined and tighter integrated analysis techniques.

Even though this paper has focused on safety aspects, security is a growing concern for critical systems as well. Frameworks such as the CERT coding guidelines [13] or CWE's list of most common security issues [14] raised awareness for security concerns. Especially CERT is used to provide coding rules for critical system development throughout different industries. However, tools support for automatic verification of security issues is not as mature as it is for safety issues.

This might be due to security concerns being both very broad and diverse while often relying on subtle implementation details as well as on external influences such as (tainted) user input. Again, we suspect that refined and integrated approaches to software verification will play an important role in checking software for potential security issues.

ACKNOWLEDGMENT

Sebastian Krings thanks Philipp Körner, Jannik Dunkelau and Chris Rutenkolk for tackling the original ABZ case study with him.

REFERENCES

[1] M. Andreessen, "Why Software Is Eating The World," *The Wall Street Journal,* 2011.

[2] NIST, "The Economic Impacts of Inadequate Infrastructure for Software Testing," National Institute of Standards and Technology, Gaithersburg, 2002.

[3] A. Raschke and F. Houdek, "Adaptive Exterior Light and Speed Control System," in *Proceedings ABZ 2021*, unpublished.

[4] Misra, "Misra C:2012 - Guidelines for the use of the C Language in Critical Systems," 2013.

[5] S. Krings, P. Körner, J. Dunkelau and C. Rutenkolk, "A Verified Low-Level Implementation of the Adaptive Exterior Light and Speed Control System," in *Proceedings ABZ 2021*, unpublished.

[6] N. El Gamal, A. El Bastawissy and G. Galal-Edeen, "Data Warehouse Testing," in *Proceedings EDBT / ICDT*, ACM, 2013, pp. 1-8.

[7] F. Haftmann, D. Kossmann and E. Lo, "A Framework for Efficient Regression Tests on Database Applications," in *The VLDB Journal*, 2007, pp. 145-164.

[8] C. Baier and J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.

[9] E. Clarke, D. Kroening and F. Lerda, "A Tool for Checking ANSI-C Progams," in *Proceedings TACAS, LNCS 2988*, Springer, 2004, pp. 168-176.

[10] J. Souyris, V. Wiels, D. Delmas and H. Delseny, "Formal Verification of Avionics Software Products," in *Proceedings FM 2009, LNCS 5850*, Springer, 2009, pp. 532-546.

[11] M. Butler, P. Körner, S. Krings, T. Lecomte, M. Leuschel, L.-F. Mejia and L. Voisin, "The first twenty five," in *Proceedings FMICS, LNCS 12327*, Springer, 2020, pp. 189-209.

[12] D. Knuth, *Notes on the van Emde Boas construction of priority deques: An instructive use of recursion,* 1977.

[13] R. Seacord, The CERT ® C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems, Addison-Wesley Professional, 2014.

[14] Mitre, "Common Weakness Enumeration," [Online]. Available: https://cwe.mitre.org.