embeddedworld 2015
Exhibition&Conference
... it´s a smarter world

Formal Verification for safety critical requirements

From Unit-Test to HIL

Markus Gros
Director Product Sales Europe & North America
BTC Embedded Systems AG
Berlin, Germany
markus.gros@btc-es.de

Hans Jürgen Holberg
Senior Vice President Marketing & Sales
BTC Embedded Systems AG
Oldenburg, Germany
hans.j.holberg@btc-es.de

***Abstract*—The first part of this talk will present a method to perform an intuitive and constructive formal requirement specification. Starting point of the formalization process is the informal textual requirement, which is structured step-by-step by the user, in order to get a clear and unambiguous representation. A new specification method called Universal Patterns allows filling the gap between natural language and machine readable description, without being an expert in formal methods. This makes the formal world usable for engineers in the area of embedded software development and testing. While being very intuitive, Universal Patterns give full flexibility regarding expressiveness and parameterization. Additionally, a maximum of readability due to an on-the-fly graphical visualization is guaranteed by this method in order to give final clearness about the user specification. A**

Since the formalized requirements are machine-readable, they can afterwards dramatically improve the quality and efficiency within the test and verification process. The machine-readable requirement observers are automatically generated form the universal pattern specification. They are used to monitor the system under test automatically regarding passed/failed status and requirement coverage rates. The second part of this talk will show how formal requirements can be smoothly integrated into different development and test environments.

Finally in the conclusion of this talk, current experiences in the automotive industry in the area of functional safety (ISO 26262, ASIL C+D) are presented and future improvements are outlined.

Keywords—formal specification, formal verification, model checking, requirements-based test case generation

I. INTRODUCTION

This paper consists of two main sections. The first part (Section II) will focus on the methodology of formal specification for safety requirements. The proposed method will allow to guide the user intuitively through the different steps of the process, starting with an informal requirement in textual form. While this pattern based approach is already proven in use in many production projects, this paper will also introduce two new aspects in this context. The first aspect proposes a different and more constructive approach to formal specification using so called “Universal Patterns”. The second aspect deals with the notion of requirements coverage and introduces a reasonable definition which can serve for coverage measurement purposes as well as for generating appropriate test cases for a requirement. Section II will conclude with the synthesis of requirement observers, which can be generated from the formal specification in order observe the behavior of a specific system-under test.

Since a formal specification represents an unambiguous and machine readable representation of a requirement, it can serve as the basis for several highly automated tasks within the test and verification process. The second part of this paper (Section III) will focus on three use cases, from which the first will talk about automatic requirements-based test case generation, where functional test cases can be generated automatically in order to fulfill, and fully cover, a requirement. The second example is the so-called simulation-based formal verification, where any kind of test data can be checked against a requirement. The section will conclude with the topic of formal verification using model checking, where a mathematical proof can show that a system-under test can never violate specific requirement.

II. Formal Specification of Safety Requirements

A. *The EmbeddedSpecifier Method*

The typical workflow of the presented requirements specification method is shown in Fig. 1. It describes a step-by-step formalization process, while with each step the user specification becomes more and more structured and more formal. It starts with a “Textual Requirement” specification given typically in natural language, which is present as an “Informal Specification”. In the next design step (1), the user identifies relevant objects in the given text, which will be identified as a "Macro Definition". These macros are used to identify any kind of events, conditions and timing information of the requirement under specification. In order to continue to structure (2) the given “Textual Requirement”, the user can select a proper method for the "Structure Definition". The adequate method directly depends on the nature of the given requirement. Practice has shown that not only one specification method can be selected to cover the specification needs of different application classes. However, for a huge set of automotive applications, like body electronics, power train, motor control, transmission and chassis, simple trigger action schemes (let us call it “Patterns”), can be used to cover most of the mission and safety critical requirements. In the presented approach, so called “Universal Pattern Specifications” are used. This approach provides the ability to construct any kind of trigger/action relation for specifying functional and safety critical requirements. Patterns can be instantiated simply by filling the pattern parameters with Boolean expressions ranging over architecture (e.g. model or code) elements/variables, which can be imported (3) as an “Interface Description” from any source. Along the typical workflow, instead of directly addressing architecture / interface objects, the user can use the former defined Macros in order to fill the pattern parameters with a “Semi-formal specification” as an intermediate specification step. Later on in the specification process (4), the user can bind existing objects of the given “Interface Description” like model elements or code variables to the macros to finalize the “Formal Specification”. During this specification step, the complete traceability between the original text and the interface element is guaranteed via the macro specification. The pattern specification method guarantees an easy entry in the formal world, without having a deep mathematical and theoretical background. This schematic pattern approach allows full certainty about what has been formally specified, without any final doubt. In the near future, the set of Structured Specifications will be extended in order to cover an extended list of application classes. Especially event driven requirements, which come in long sequences, need another specification approach. Here Sequence Diagrams (SDs) or Life Sequence Charts (LSCs) have been used successfully in several research projects in the past. In a complete automated phase (5), a “Machine-readable specification” aka “Requirement Observer” (e.g. C-Code or Python etc.) can be generated for further formal verification activities.

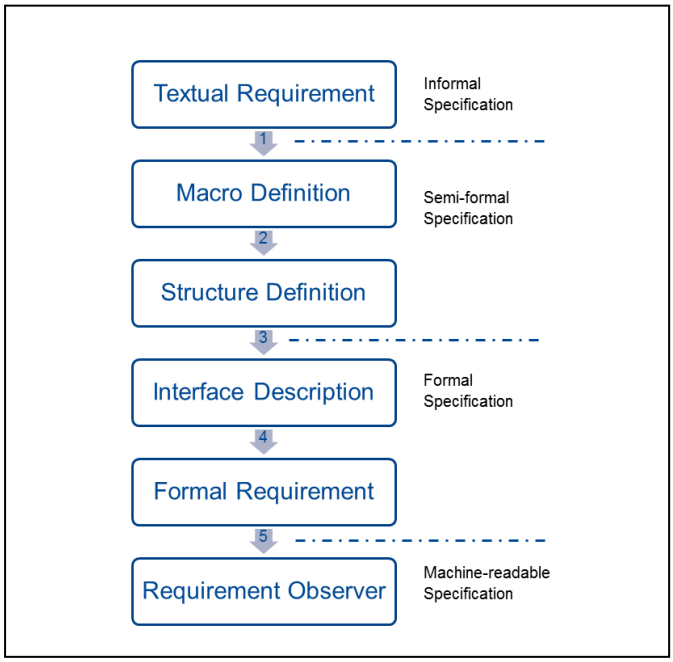


Fig. 1: Formalization process

B. *Universal Pattern*

The “Universal Pattern” structuring method allows the user to define simple and complex trigger action relations in a constructive and intuitive way. Fig. 2 shows a simple example where a single “Trigger” and a single “Action” relation have been defined. The relationship between the 2 events is defined by the user, based on different universal pattern “Interpretations”.

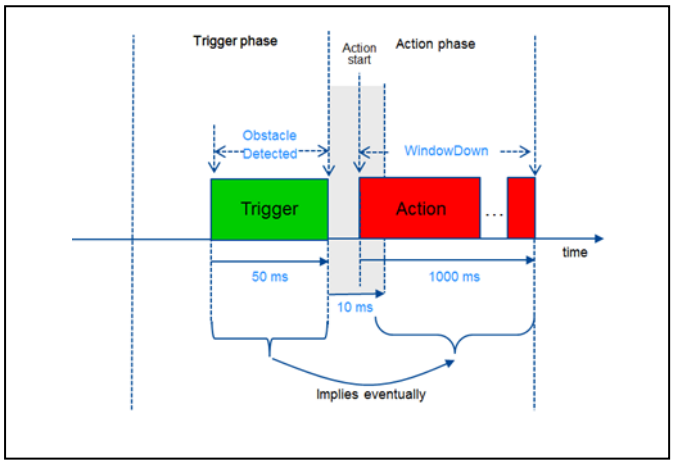


Fig. 2: Example of a Universal Pattern definition

Three different “Interpretation” can be selected:

- Progress (“implies eventually”)
- Ordering (“only after”, “not before”)
- Invariant (“always”)

The example shown in Fig. 2 considers the following informal requirement of a car window controller:

“*If an obstacle is detected at least for 50 ms, the window down signal has to be activated for minimum time frame of 1 sec.* “.

After selecting the right “Interpretation” of the “Universal Pattern”, the user has to parametrize the “Trigger” and “Action” object regarding time duration and stable condition. For a “Semi-formal Specification”, 2 Macros are defined directly with a simple use interaction on the textual requirement:

1. “[..] obstacle is detected […]”-> `ObstacleDetected`
2. “[…] window down signal […]” -> `WindowDown`

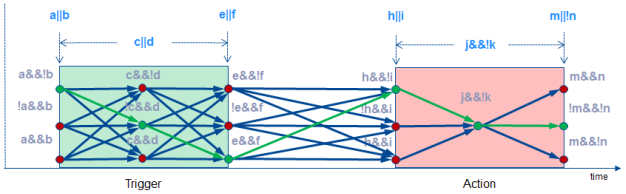
The Macro `ObstacleDetected` becomes the “Trigger Condition” and the Macro `WindowDown` is used to define the “Action Condition”. The timing definitions are defined directly with the specified time data: `50ms` (Trigger Duration), `10ms` (Scope Duration) and `1000ms` (Action Duration) as real-time definitions. In order to reach the level of a “Formal Specification” the 2 Macros have to be bound to real “Interface Description” signals like “input”, “output”, “local” and “calibration” variables of the system under verification. If this level is reached, the syntax as well as the semantics of the requirement specification is clearly defined.

C. Requirement Coverage

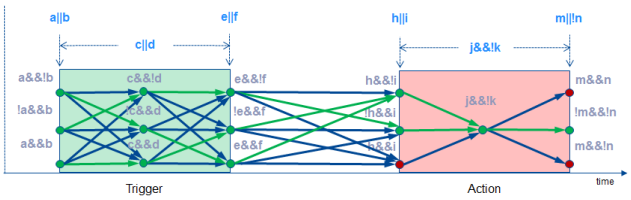
Based on the universal pattern specification method, a new definition of requirement coverage has been identified and has been mathematically defined in order to allow the measurement of requirement coverage in complete automated way. This is important for different reasons. Especially quality standards like IEC 61508, ISO 26262 and DO 178c always reference the term requirement coverage as the final goal of testing, beside other structural coverage criteria like model or code coverage. Additionally, this term is very important to identify test goals for any kind of automatic test generation activities. Generally the definition of requirement coverage has been done upon the so called trigger/action relationship, which is the basis for the universal pattern approach. The main idea of requirement coverage is the ratio between the reached relevant trigger combinations ($TriComb^{reached}$), which activate the corresponding action and all relevant trigger combinations ($TriComb^{all}$):

$$ReqCoverageRate = \frac{TriComb^{reached}}{TriComb^{all}} \%$$

As other coverage criteria, requirement coverage comes in different levels of coverage according to the desired test quality level to be reached (the higher the level the higher the number of test sequences to be covered):

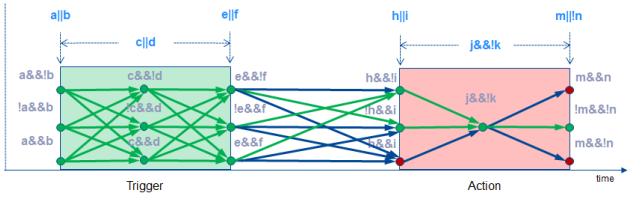


Level 0: Asks for reaching some of the goals. One trace suffices which satisfies (or violates) the action.

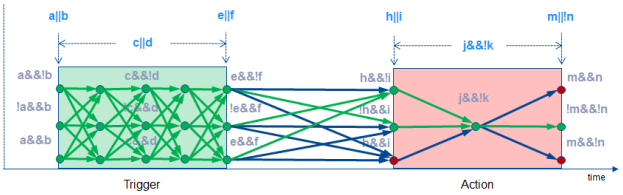


Level 1: Asks for reaching all goals.

:



Level 2: Asks for reaching all combinations of goals



Level 3: Asks for reaching all time-depending combinations of goals:

Fig. 3: Definition of levels for requirement coverage

D. Requirement Observers

After the user has specified requirements in a formal way, all semantical information is available to represent the property in a complete mathematical sense. This enables a complete automatic synthesis (generation) of so called requirement observers in an arbitrary computer language (e.g. c-code or

python). A requirement observer is defined over the name space of the interface of the system under verification, as the interface definition has been taken as one basis of the formal specification of the original requirement. This observer acts as a watch dog, which is part of the test harness of the system under verification to judge at any moment in time of execution, if the requirement is valid or not. In other words, it is an automatic verdict function of the system under verification in respect to the original specified requirement.

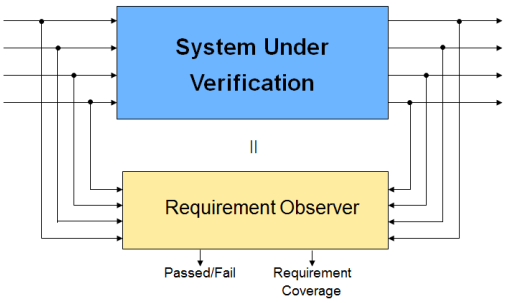


Fig. 4: Usage of requirement observers

The Figure above shows, that beside the verdict "Passed/Fail" analysis, the requirement observer additionally can calculate automatically the reached "Requirement Coverage" rates based on the defined coverage criterion definition. Even more, this requirement observers can be used for any Formal Verification activity or automatic test generation process, if the system under verification is parallel composed with the requirement observers. These use cases will be presented in more detail in the following section.

III. Using Formal Specifications in the Verification Process

A. *Automatic requirements-based test vector generation*

One of the most time consuming tasks in the testing and verification workflow is typically the creation of appropriate test cases, meaning to derive test cases from the textual requirements. These test cases are used on all levels along the v-cycle to show that a specific system-under-test correctly implements the corresponding requirements, from unit test on model- and code-level up the system tests and HIL testing. Such a test case, or test vector, typically consists of vectorised signals to be applied to the input variables of the system as well as a test verdict that allows to decide if the requirement is violated or not. But the manual analysis of textual requirements followed by the manual creation of test vectors is not only time consuming, it can also be error prone due to the interpretation of the requirement performed by the "human" engineer and depending on the quality and clarity of the requirements. In addition, textual requirements and manually created test vectors always lead to the question if a requirement has been completely tested and how many test cases are actually needed to completely cover a requirement.

In case a requirement has been formalized, these issues can be addressed quite efficiently in an automated way. As the formalized requirements are machine readable, automated analysis methods such as model checking can be used, to produce requirements-based test vectors automatically. For these vectors the notion of "requirements coverage" that has been introduced before also allows to make sure, that the requirements are completely covered.

While theoretically a complete set of formalized requirements for a system is sufficient to generate the corresponding test vectors, experience in real projects has shown that this ideal situation is almost never present. In practice, the system-under-test most likely contains behavior which is not fully described by the requirements. For example, we might have a requirement that only talks about output variables of the system, but the considered set of requirements does not fully describe how these outputs are connected with the inputs. Another typical issue is, that the process of formalization is often focused on the safety-critical requirements and is therefore rarely exercised for all requirements within a project. A practical solution to these challenges is, to analyze the requirement(s) together with the system-under-test for finding appropriate test data. In this case, a model checker can use the information from the system-under-test to find an appropriate set of test cases to drive the system to the state(s) described by the requirement and to fully cover a requirement. Thanks to the information contained in the system-under-test, this method can even be applied for a single formalized requirement. One pre-condition for this approach is of course the availability of the system-under-test in a form that can be used as an input for the model checker. Therefore, the method is especially suitable for the unit-test of embedded software, where the system-under-test is typically available as ANSI-C code.

The resulting test cases can then of course also be applied to other verification levels such as model-in-the-loop and processor-in-the loop.

B. *Simulation-based formal verification*

In a typical verification process, each test case is normally related to a requirement which is verified by executing this test and inspecting the result. However, if a test case fulfills "his requirement" but violates a different requirement, it would most likely be unnoticed.

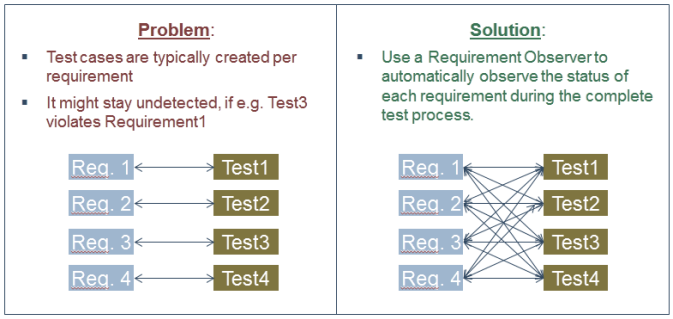


Fig. 5: Motivation for simulation-based formal verification

This issue can be efficiently addressed with formal specifications, allowing to automatically check all test data against a requirement. Assuming that a requirement has already been formalized, this so called "simulation-based formal verification" can be applied in the verification process with very little additional effort, while providing a high benefit regarding test depth and therefore quality. The method only requires access to the test data and a formal specification in executable form and can therefore be flexibly applied for any test level all along the V-Cycle, from unit test to hardware-in-the loop testing. Furthermore, in contrast to methods like model checking, the complexity and size of the system-under-test has no impact on the analysis effort, which makes this method also suitable for very large systems. The verification can either be performed "Online" by observing the system behavior during the test execution or "Offline" by analyzing recorded test data after the test execution.

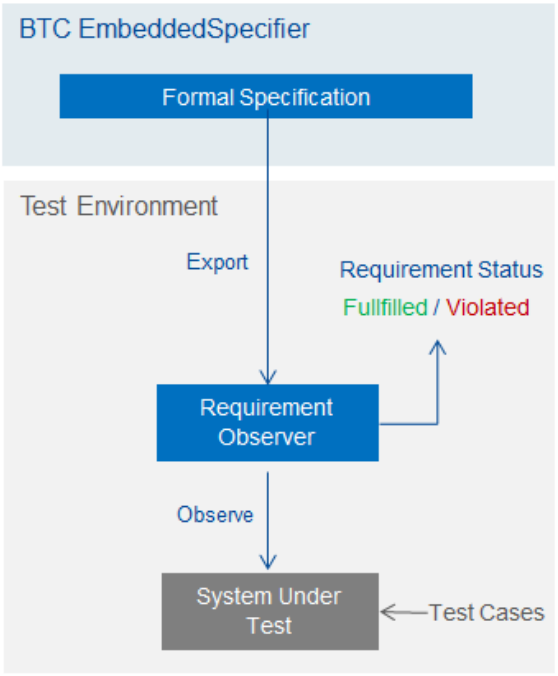


Fig. 6: Online Verification

For the "Online Verification", an executable requirement observer can be exported and then executed together with the system-under-test as shown in Fig. 6, for example as a real-time application on an HIL system (The export of a requirement observer from the specification tool has already been presented in section II-D). If the tests are performed interactively with an experiment tool, the user is able to monitor the status of each requirement in real time. In case a test automation is used to automatically execute a series of pre-defined test scenarios, the test automation tool can access the requirement status in the same way as it reads interface variables from the system under test. This makes it very easy to integrate the additional information about a requirement being passed or failed into a test report.

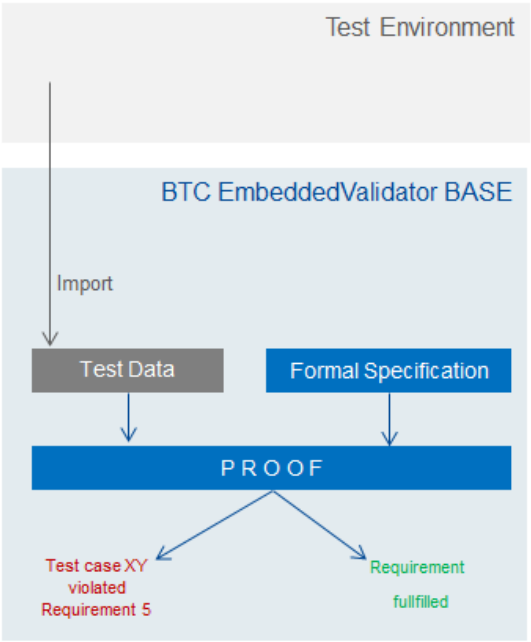


Fig. 7: Offline Verification

Sometimes it can be difficult to integrate and execute the requirement observer together with the system-under-test, e.g. when performing in-vehicle tests. In this case, the verification can also be performed "Offline" using recorded test data, as shown in Fig. 7. This data can be imported into an analysis tool which checks all test recordings for violations of requirements. After the analysis the tool can generate a report showing which requirements are always fulfilled and which requirements are violated by which test case. In addition, the notion of requirements coverage that has been presented previously allows to provide information about how good a requirement has been covered by the given set of test cases.

C. Formal verification using model checking

When executing one or more test cases on a system-under-test, each test case represents one run through the possible combination of system states over time. Since covering all possible runs would require an almost infinite number of test cases, it can be concluded that testing is never complete and therefore requires to establish criteria to decide when to stop testing. Examples for this kind of criteria are requirements coverage (informal or formal) or structural coverage criteria like MC/DC. However, none of this can guarantee an error free system.

One method allowing to analyze all possible combinations of system states over time is model checking. When a system-under-test is brought together with formalized requirements, a model checker can perform an automatic and complete proof showing, that no combination of input signals over time can bring the system into a state where the requirement is violated. However, if it is possible to bring the system into a state where the requirement is violated, the model checker will generate a corresponding test case as a counter example (see Fig. 6). Thanks to the analytical nature of model checking, the counter example will always be the shortest trace possible and will also only change a minimal set of interface variables. These characteristics typically facilitate an efficient and intuitive debugging.

This method obviously requires a system-under-test which can be fed into the model checker, e.g. a function realized as ANSI-C Code. In addition, it needs to be said the analysis effort grows exponentially with the system size, which means that in practice the method is in particular applicable as part of the verification of software units.

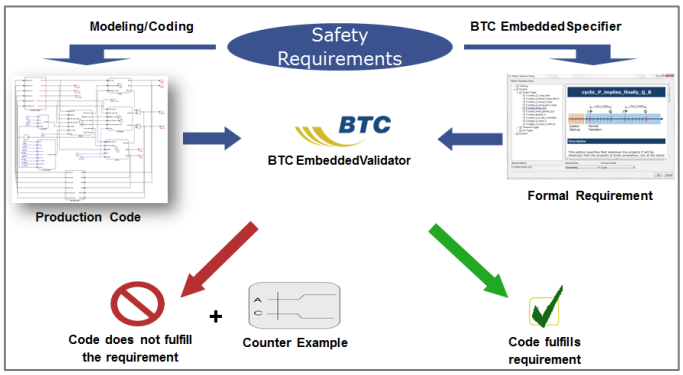


Fig. 8: Formal Verification using model checking

IV. CONCLUSION

The presented EmbeddedSpecifier method enables the engineer of embedded control software to specify requirements in a formal way and to fill the gap between informal textual requirements and machine-readable specifications. It provides an intuitive way of requirements specification, which directly addresses safety standard like ISO 26262 for functional safety for all safety integrity levels. Due to the computer-aided way of specification refinements, and due to the step-by-step way of structuring informal requirements, formal specification methods become usable even for non-experts. The automatic binding of existing testing architectures to the requirement specifications makes this method a very effective and efficient approach to enable automatic Formal Verification of any kind. The key-technology here is the automatic Requirement-Observer-Generation out of formal specifications, which builds the bridge between the operational world (executable specifications) and the declarative world (requirement statements). The described different levels of Formal Verification can be used very flexibly for different use cases and different test levels from unit test to system test. Even along the desired safety integrity level, different formal verification technologies can be selected by the user according to the given circumstances. The presented formal specification and formal verification methods in combination are one key of success for addressing three important aspects to cope with the time-to-market pressure of many industries: efficiency, quality and fulfilling safety standards.

The presented approach has been realized as a requirement specification environment called BTC EmbeddedSpecifier. It comes as a standalone tool environment to enable formal specification in general, but it is smoothly integrated into the existing BTC EmbeddedTester automatic testing tool-chain especially made for dSPACE TargetLink as part of a strategic partnership. This seamless verification tool chain currently is heavily used in the automotive industry by many OEMs and tire-one suppliers in the embedded software domains of motor control, chassis, body electronic, transmission, powertrain etc. in Europe and Japan. In the near future, a connection to dSPACE Hardware-in-the-Loop Systems will be introduced to the market, which allows benefiting from formal verification technology even on integration and system level.