Efficient and Trustworthy Tool Qualification for Model-Based Testing Tools

Jörg Brauer[1], Jan Peleska[1,2,⋆], and Uwe Schulze[2,⋆⋆]

[1] Verified Systems International GmbH, Bremen, Germany
[2] Department of Mathematics and Computer Science, University of Bremen, Germany

Abstract. The application of test automation tools in a safety-critical context requires so-called tool qualification according to the applicable standards. The objective of this qualification is to justify that verification steps automated by the tool will not lead to faulty systems under test to be accepted as fit for purpose. In this paper we review the tool qualification requirements of the standards ISO 26262 (automotive domain) and the new RTCA DO-178C (avionic domain) and propose a general approach on how to qualify model-based testing tools according to these standards in an efficient and at the same time reliable way. Our approach relies on a lightweight error detection mechanism based on the idea of replaying test executions against the model. We further show how the error detection capabilities can be integrated into a convincing argument for tool qualification, going through the necessary verification activities step-by-step. We highlight the key steps for the RT-Tester Model-Based Test Generator, which is used in test campaigns in the automotive, railway and avionic domains. The approach avoids having to qualify several complex components present in model-based testing tools, such as code generators for test procedures and constraint solving algorithms for test data elaboration.

1 Introduction

In model-based testing, a test model is used to define the expected behavior of the system-under-test (SUT). From this formal specification of the desired system behavior, test cases are generated, which are then executed against the SUT[1]. The generation of test cases is frequently based on techniques such as abstract interpretation [3] and SMT solving [8], which exercise the semantic structure of the model to automatically calculate these test cases; such techniques

⋆ The author's contribution has been developed during the course of the project "Verifikation von Systemen synchroner Softwarekomponenten" (VerSyKo) funded by the German ministry for education and research (BMBF).

⋆⋆ The author's research is funded by the EU FP7 COMPASS project under grant agreement no.287829.

[1] It is important to note that correctness of model-based test generators relies on the consistency and completeness of test models, since test cases are derived directly from the models. It is thus assumed that the test models have undergone review.

B. Nielsen and C. Weise (Eds.): ICTSS 2012, LNCS 7641, pp. 8–23, 2012.

are also implemented in our tool-suite RT-Tester Model-Based Test Generator (RTT-MBT) [12]. The resulting test cases are then specified as sequences of input data — including timing constraints — that stimulate computations of the SUT conforming to the test case specifications. In addition to the input stimulations, RTT-MBT automatically generates test oracles that run concurrently with the SUT, checking the responses of the SUT against the test model on-the-fly. In combination, the stimulation component and the test oracles form a test procedure which is compiled and executed in a test execution environment (TE). The test execution environment then generates a so-called *execution log*, which contains the data observed and recorded during test case execution.

1.1 Model-Based Testing in Industry and Tool Qualification

The success of model-based testing in industry has been stimulated by the success of model-driven software development in general. Indeed, compared to conventional approaches, model-based testing has proven to increase both, quality and efficiency of test campaigns [10], which may explain the industrial interest in model-based methods, especially from domains such as automotive, avionics, and railway industry. However, all tools that automate process steps in the development and verification of safety-critical systems (e. g. code generators, compilers, model checkers, test automation tools) need to be qualified since they automate a life cycle activity so that a manual inspection of its outcome (e. g., generated source code, object code, verification or test results) is rendered superfluous. In this situation it has to be ensured that the tool performing this automation cannot inject errors into the artifacts produced; otherwise, this would induce the risk of a faulty system component to be accepted as fit for purpose.

The ISO 26262 [6] standard currently implemented in the automotive domain presents guidelines for the development of safety-related systems in road vehicles. This standard is an exemplar of a standard prescribing required properties for development and verification automation tools. The standard [6, Sect. 11.4] itself expresses the aim of tool qualification as follows:

> "*The objective of the qualification of software tools is to provide evidence of software tool suitability for use when developing a safety-related item or element, such that confidence can be achieved in the correct execution of activities and tasks required by ISO 26262.*"

The key steps of providing the required evidence are defined in the standard. First of all, qualifying a tool for a development process necessitates to determine the tool impact and its error detection capabilities. These two factors are combined to form an overall *tool confidence level*. Of course, the more severe the developed system component, the stricter the requirements imposed onto the tool. Yet, an interesting aspect of ISO 26262 [6, Sect. 11.4.4.1] is that tools with maximal *tool error detection capabilities* do not require qualification measures at all, as long as error detection is perfectly reliable. Hence, if a tool is capable of detecting its own malfunctioning, the entire tool qualification process for ISO 26262 can be simplified in a significant way.

1.2 Tool Qualification for ISO 26262

Malfunction of a test-case and test procedure generator such as RTT-MBT introduces two hazards which may result in a situation where a requirement allocated to a safety-related item[2] is violated, due to malfunction of this item:

- **Hazard 1: undetected SUT failures.** A deviation of the observed SUT behavior from its expected behavior specified in the test model may potentially remain undetected if the generator creates erroneous test oracles failing to detect this deviation during test executions.
- **Hazard 2: undetected coverage failures.** The test execution fails to meet the pre-conditions which are necessary in order to cover a given test case, but the test oracles indicate TEST PASSED because the observed execution is consistent with the model. If this situation remains undetected, it may be assumed that the SUT performs correctly with respect to the specified test case, while in fact the test procedure tested "something else".

The qualification goal required by ISO 26262 states that any possible hazard introduced by the tool will eventually be detected [6, Sect. 11.4.3.2]. The identification of components of model-based test generators relevant for qualification, as well as trustworthy, yet lightweight methods for satisfying this objective are topic of this paper. Of course, formal verification of RTT-MBT as a whole to prove the absence of defects is an unrealistic mission, as the state-of-the-art represents verifying functional correctness of systems that involve approximately 10,000 lines of C code [7], and RTT-MBT consists of approx. 250,000 lines of C/C++ code. To qualify RTT-MBT for use in the development of software of high quality assurance levels (according to ISO 26262 and other standards such as RTCA DO-178C), it is thus necessary to combine formal verification with testing and effective tool error detection. In the following, we discuss the verification strategies we applied to RTT-MBT and the tool error detection mechanism.

The key idea of our approach to tool qualification is simple: Rather than attempting *a priori* verification for the test generator by proving conformance of generated test procedures with the model, we focus on the *a posteriori* error detection capabilities of RTT-MBT, regarding the consistency of the test execution log with the model. This is performed by *replaying* the execution log on the test model. The key functionality for replay of test execution is defined as follows:

- A simulation of the test model is generated that uses exactly the input data to the SUT that was used during test procedure execution.
- The respective simulation contains the *expected* SUT outputs as calculated from the test model. These outputs are compared to the outputs *observed* during test procedure execution and documented in the test execution log. Any deviations are recorded in the replay verification results.
- The actual model and test coverage achieved by the simulation is recorded in the replay verification results, too.

[2] That is, a software or embedded HW/SW control system tested by means of procedures generated by the tool.

Using this strategy, the impact of errors in the test generator are localized. In essence, it suffices to show that replay detects any deviation of the concrete test execution from the test model. Establishing this correctness argument is strictly easier than proving correctness of the entire test generation functionality. In principle, replay could be performed on-the-fly, concurrently to the test execution. Yet, we describe it as an *a posteriori* activity, to be performed offline after the execution has been completed, since hard real-time test engines running HW/SW or system integration tests often do not have sufficient spare computing power to cope with additional model executions for replay purposes.

1.3 Contributions and Outline

In summary, this paper presents the following contributions:

- We present an analysis that relates properties of model-based test generators to requirements for tool qualification posed by ISO 26262.
- We identify classes of hazards introduced by test generators and provide an analysis of parts of a test generator that are relevant to qualification.
- We introduce a lightweight framework for replaying concrete test executions with the aim of identifying erroneous test case executions.
- We show how the different verification activities, consisting of design analyses, formal verification, structural testing and tool error detection are combined to form a convincing case in point for customers and certification authorities.

The exposition of this paper is laid out as follows: First, Sect. 2 presents an impact analysis that connects the test generator to the demands posed by the ISO 26262. Then, Sect. 3 identifies those parts of a test generator that are relevant for tool qualification and discusses the details of the verification strategies applied. Following, Sect. 4 studies properties of the replay and constructs a correctness argument from the architecture, before Sect. 5 discusses differences between ISO 26262 and RTCA DO-178C w.r.t. test-case generators. Finally, the paper concludes with a survey of related work in Sect. 6 and a discussion in Sect. 7.

2 Tool Classification According to ISO 26262

To assess the qualification requirements for a model-based test-case generator, it is necessary to analyze hazards potentially inflicted by the tool, as well as the impact of these hazards. The second important aspect are the tool error detection capabilities of a tool with respect to its own malfunction. Based on this analysis, particular requirements are imposed onto the tool, which are discussed in the remainder of this section.

2.1 Impact Analysis

ISO 26262 defines two different *tool impact (**TI**) levels* for software tools of any kind: **TI0** is applicable iff malfunctioning of the tool can under no circumstances introduce a hazard; otherwise, **TI1** shall be chosen [6, Sect. 11.4.3.2].

Assuming correctness and completeness of the test models[3] potential malfunction of a model-based test-case generator introduces the hazards 1 (undetected SUT failures) and 2 (undetected coverage failures) introduced in Sect. 1. These hazards clearly imply tool impact level **TI1** for model-based test-case generators.

2.2 Tool Error Detection Capabilities

The probability of preventing or detecting an erroneous tool output is expressed by *tool error detection classes* ***TD*** [6, Sect. 11.4.3.2]. If there is high confidence in the ability of a tool to detect its own malfunctioning, then **TD1** is the appropriate classification. Lower classes such as **TD3** or **TD4** are applicable if there is low confidence in the tool's error detection facilities, or if no such mechanism exists. To achieve **TD1**, we specify one tool-external measure — that is, a guideline to be respected by the test engineers applying the tool — and three tool-internal measures, that is, measures implemented in software.

External Measure. Every test execution of test procedures generated by RTT-MBT shall be replayed.

Independently of the specific functionality of RTT-MBT, this external measure is mandatory to enforce anyway, since every testing activity for safety-critical systems requires a verification of the test results. The replay function as introduced in Sect. 1 can be considered as a tool-supported review of this kind, because it verifies whether the test execution observed complies with the model and whether the intended test cases have really been executed. To enable the external measure above, or, equivalently, to achieve **TD1**, we implement three tool-internal measures with the following objectives:

Internal Measure #1. Every change of input data to the SUT is correctly captured by logging commands in the test stimulator of the test procedure.

Internal Measure #2. Every change of SUT output data is correctly captured by logging commands in the test oracles of the generated test procedure.

Internal Measure #3. The replay mechanism detects (1) every deviation of the SUT behavior observed during test execution from the SUT behavior expected according to the test model, and (2) every deviation of the test cases actually covered during the test execution from the test cases planned to be covered according to the test generator.

[3] Since models represent abstractions of a real system, correctness and completeness is usually defined according to some conformance relation between model and SUT. In our case – since the SUT never blocks inputs – conformance means that for any given timed trace of inputs (1) the SUT produces the same observable outputs as the model, modulo tolerances regarding timing and floating point values, and (2) that the ordering of inputs and outputs, when restricted to a sequential sub-component of the model, is the same for model and SUT (partial ordering of observable I/Os).

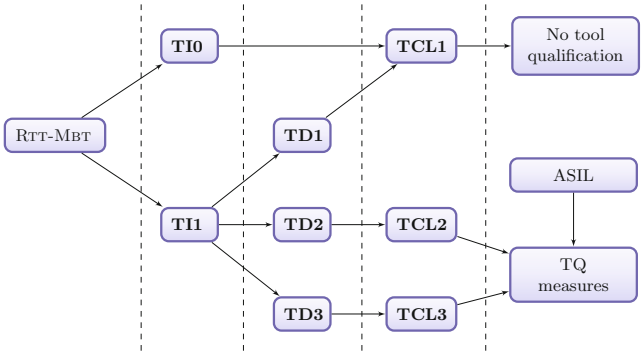

Fig. 1. Relation between tool impact (**TI**), error detection (**TD**) and automotive safety integrity level (**ASIL**)

The effectiveness of internal measures 1 — 3 clear implies the desired **TD1** capability, as long as test engineers comply with the external measure above.

For the tool-internal measures to be effective, two prerequisites of the overall test system need to be satisfied: (1) All logging commands in the test procedures are correctly executed and recorded in the execution log, and (2) the test model is complete and correct. Clearly, the first prerequisite is delegated to the test execution environment, a component which is independent of the functionality discussed in this paper and has to be qualified on its own. Satisfying the second prerequisite is the duty of test engineers that use a model-based test generator.

2.3 Tool Confidence Level

Pairing the tool impact **TI** with the appropriate tool error detection capability **TD** yields the associated *tool confidence level* ***TCL*** [6, Sect. 11.4.3.2]. It is remarkable that for a tool with impact level **TI1** and error detection class **TD1** (such as RTT-MBT), no qualification whatsoever is required according to [6, Sect. 11.4.4.1]. Otherwise, tool qualification measures have to be adopted according to the automotive safety integrity level (**ASIL**) of the system under test. This situation is depicted in Fig. 1. High confidence in the error detection capabilities of a tool thus eases the tool qualification significantly, which motivates the desire for trustworthy, lightweight error detection integrated into the tool.

3 Verification for Qualification

The overall architecture of RTT-MBT is depicted in Fig. 2. A parser component translates an input model written in UML or SysML, which is given as an XML export of some modeling tool, into an intermediate model representation (IMR, the abstract syntax tree of the model) that is used by both the test-case generation and the replay facilities. The test-case generator uses techniques such as SMT solving, abstract interpretation and code generation in order to generate test inputs and the corresponding test procedures. In contrast to this, the

Table 1. Functionality in RTT-MBT affected by tool-internal measures for error detection

Measure	Affected Functionality
#1	The method that creates the log commands associated with SUT inputs.
#2	The method that creates a conditional logic operation that shall be executed whenever an SUT output observed is changed. The method that creates the test oracles and inserts a log operation to be executed whenever an SUT output observed by this test oracle is changed. The method that stores the conditional log operations. The method that creates the log operation in the syntax of the test oracle.
#3	The method that parses an execution log. The interaction between the replay log and the simulator. The methods that simulate an input from the replay log. The method comparing the observed SUT outputs with the expected SUT outputs calculated by the simulator The methods that manage the internal memory during simulation. The methods that determine whether the test cases covered during replay are identical to those observed during a concrete execution.

sole purpose of the replay mechanism is to simulate an execution of a concrete execution log on the test model. Test-case generation and replay are thus independent components that have also been developed with independence. The test generator, however, is responsible for the generation of log commands in test procedures, and the replay mechanism depends on their correct generation (recall the tool-internal measures discussed in Sect. 2.2). Therefore this part of the generator (it is a sub-component of the test procedure generator) receives increased attention during verification.

3.1 Identification of Relevant Components

The parser is responsible for translating an input model into an IMR. It has thus to be qualified as errors in this component may mask failures of the test generator and the replay. To identify classes and methods reachable from the parser, we use data-flow analysis and code inspection. Apart from the parser itself, the relevant classes most notably include the IMR used within RTT-MBT.

As argued in Sect. 2.2, a replay that implements the tool-internal measures eliminates the need to qualify test generation. This strategy entails that the replay needs to be qualified, as tool error detection is delegated to the replay component. It is noteworthy that test generation is much more complex than replay. Complementing test generators by replay mechanisms thus reduces the workload for qualification significantly: the existence of tool error detection turns test generation — with the exception of the log command generator — into a component whose outputs need not be verified. As before, we apply data-flow analysis to identify classes and methods that are involved during replay. The result of this analysis includes, most notably, the IMR, the memory model storing states during model simulation, the simulator, and the parser for the test execution log.

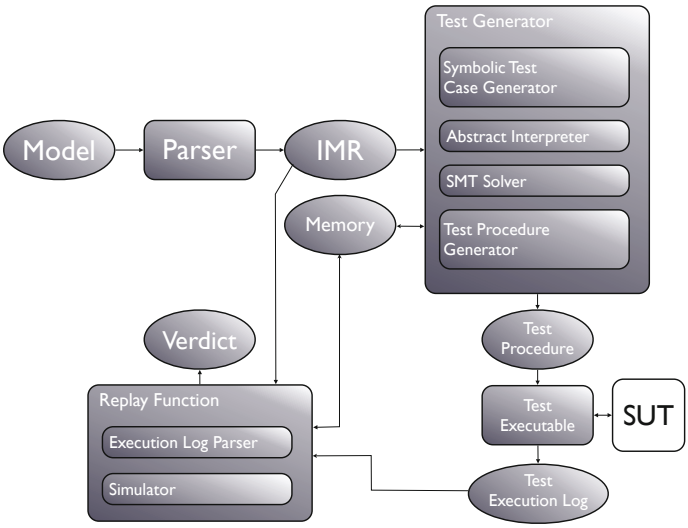

Fig. 2. High-level architecture of RTT-MBT

3.2 Identification of Tool-Internal Measures

Additionally, we performed a design review to identify those parts of RTT-MBT implementing the tool-internal measures 1—3 for error detection. The results of this analysis are given in Tab. 1. These parts must under any circumstances be implemented correctly. Special attention should thus be paid during the verification of the methods highlighted in Tab. 1.

3.3 Verification Activities

To verify the correctness of each involved component, we apply different verification activities. The main verification activity is (software integration) testing, though this process is augmented with formal verification for the most critical software parts. The combination of methods applied conforms to the requirements of RTCA DO-178B for developing software of the highest criticality (Level A): (1) The development process for the replay component has been controlled according to [15, Tab. A-1,A-2]. (2) High-level and low-level requirements specified for the replay mechanism have been verified with independence according to the approach defined in [15, Tab. A-3,A-4]. (3) The source code has been inspected, formally verified and tested according to [15, Tab. A-5, A-6, A-7]. (4) The configuration management and software quality assurance processes have been set up for the whole RTT-MBT product in a way conforming to [15, Tab. A-8,A-9], respectively.

Requirements-based testing. For each high-level and low-level requirement of the test generator, we provide normal behavior tests as well as robustness tests that investigate the stability of RTT-MBT.

Table 2. Verification activities for affected components

Component	Requirements testing	Structural testing	Formal Verification
Measure #1	yes	yes	yes
Measure #2	yes	yes	yes
Measure #3	yes	yes	no
Parser	yes	yes	no
IMR	yes	yes	no
Replay	yes	yes	no
Generation	no	no	no

Structural testing. We provide a collection of test cases that achieve MC/DC coverage for the reachable parts of RTT-MBT.

Formal Verification. For certain critical components, we perform additional formal verification and documentation of the verification results. An example of functionality for which proofs are provided are the logging facilities. Formal verification is performed for small isolated functions specified by pre- and post-conditions. The verification is performed manually using Hoare Logic. The small size of the verified functions indicated that manual proofs were acceptable and could be checked independently by a verification specialist.

A summary of the verification activities for each component is given in Tab. 2. We define the following general criteria for the testing process of the involved components, before discussing the details for the strategy applied to the parser.

Pass/fail criteria. The test passes if the expected results are produced without deviation; otherwise, it fails.

Test end criteria. The test ends when all test cases have passed and the test suite resulted in a 100% MC/DC coverage for the involved methods.

General test strategy. Each method shall be tested as follows:

- The test cases for verifying the methods are elaborated.
- A test model is selected which is suitable for the test cases under consideration. Test cases shall cover both normal behavior and robustness.
- A replay file is selected which is based on a test execution whose test procedure was generated with the selected model and which is suitable for the test cases under consideration.
- The replay function is activated with the model and replay file as inputs.
- The replay results consist of the pass/fail results achieved during the replay, and the list of test cases covered during the replay.
- The replay results are verified with respect to the expected results. It is checked that the replay result is pass if and only if the replay file corresponds to a correct model computation. Evaluating the replay file against the model, it is checked against the model that the list of covered test cases produced during replay is correct and complete.
- The MC/DC coverage achieved during the execution of the test suite is checked whether it results in 100% for the methods identified above.

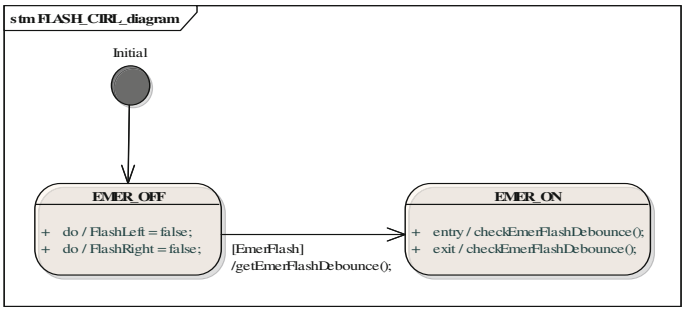

Fig. 3. A simple state machine

– If some code portions could not be covered by the test suite designed according to the guidelines listed here, it is admissible to perform unit tests on the methods not completely covered so far. The unit tests shall check the expected results by means of post conditions. The associated unit test cases specify the data needed to cover the missing code portions as test condition, and the post condition as expected result.

Test Strategy for the Parser Component. The IMR of a model is the basis for all further functionality of RTT-MBT. Errors in the IMR can therefore mask failures in the test data generation as well as in the replay. Thus the parser component setting up the IMR has to be qualified. The IMR is a representation of the abstract syntax tree (AST) of the model restricted to those UML/SysML elements which are supported by RTT-MBT. An example of a simple state machine, which is part of a test model, is shown in Fig. 3. The corresponding IMR is shown in Fig. 4. In addition to the IMR, a model of the internal memory representation is generated, which is used during simulations to store the values of variables and control states. Both the AST and the memory model are used as inputs to the replay and must thus be qualified.

For requirements-based tests of the parser, models serve as input that have been designed to specifically test one or more syntactic UML/SysML feature supported by the parser. Their respective XML representations are the test inputs to the parser, which generates both, the IMR and the memory model, from the test model. The IMR is then verified to be a valid representation of the model. For each test model, we handcraft an AST corresponding to the expected IMR. The result generated by the parser is then checked against this expected AST.

The memory model is an algorithmically simple component. Qualification amounts to verifying that for each control state and each variable, an appropriate entry in the internal memory is initialized. Whenever a value is stored in the memory, its internal state shall correctly reflect the update. It also needs to be ensured that support for data types is exhaustive. These properties are checked using a combination of requirements-based integration tests and unit tests which, e.g., systematically probe all data types. The inputs for the requirements-based

test procedures form equivalence classes. The structural coverage tests refine these equivalence classes so that MC/DC coverage is eventually achieved.

As explained above, the verification strategy for the parser merely analyzes the correct parsing and transformation of a pre-defined class of language patterns. This approach dovetails with strategies applied to non-optimizing compilers [4, Sect. 2.1]. There, the strategy is to verify the correct translation of each supported symbol in a high-level programming language into the corresponding assembly fragment. Since RTT-MBT does not optimize the AST, such a direct mapping from expected inputs to outputs can also be established for the model parser.

4 Error Detection Using Replay

Objectives. In this section we elaborate a formal argument to show that replay — if correctly implemented — enforces the tool error detection class **TD1** as required. Let us briefly recall the scenarios that may occur once a generated test case is executed: (1) If test execution fails, manual investigation is required to justify the deviations, identify an erroneous test case, fix the SUT, or refine the test model. (2) If test execution passes, this may be due to one of the following reasons: (a) The test cases were generated correctly from the test model and the SUT conforms these test cases, or (b) some test cases were incorrectly generated[4] from the test model, but the SUT behavior still conforms to these faulty cases. Tool error detection thus needs to classify only test case executions that pass. In the following, we build towards a correctness argument for the replay.

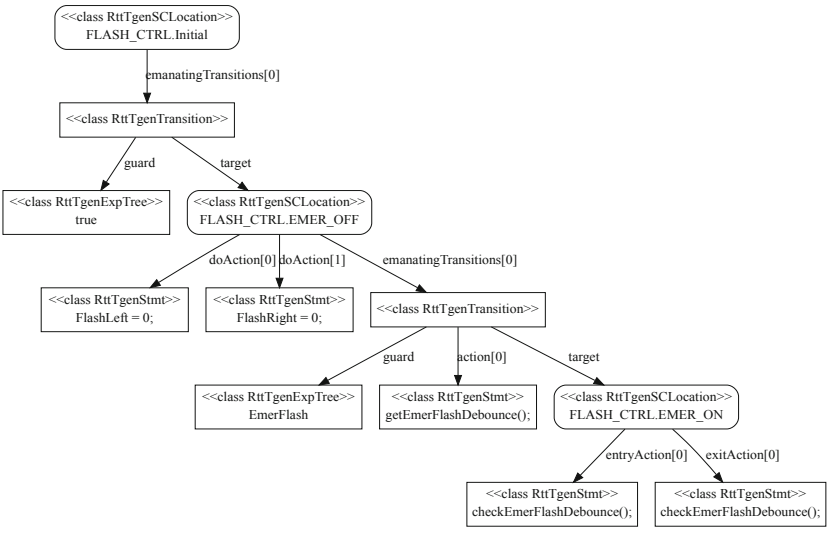

Fig. 4. IMR of the state machine in Fig. 3

[4] This means that either the test data is inappropriate for the test objective, or the check of expected results is faulty.

Prerequisites. The SUT $\mathbf{S}$ communicates with the test environment through finite sets of input signals $\mathcal{I} = \{i_1, \ldots, i_m\}$ and output signals $\mathcal{O} = \{o_1, \ldots, o_n\}$ such that $\mathcal{I} \cap \mathcal{O} = \varnothing$. We denote the set of overall signals by $\mathcal{S} = \mathcal{I} \cup \mathcal{O}$. Further, each signal $s \in \mathcal{S}$ is assigned a type drawn from a finite class $\mathbb{T} = \{\mathbb{N}, \mathbb{B}, \mathbb{Q}, \ldots\}$ of types using a map $\sigma : \mathcal{S} \to \mathbb{T}$. An input to the SUT is then given as a triple $\langle t, i, v \rangle$ where $t \in \mathbb{R}$ is a time-stamp, $i \in \mathcal{I}$ is an input signal and $v \in \sigma(i)$ is an assignment to i. Likewise, an output from the SUT to the test environment is a triple $\langle t, o, v \rangle$ with $t \in \mathbb{R}$, $o \in \mathcal{O}$, and $v \in \sigma(o)$.

Model Semantics. The desired behavior of the SUT is specified by means of a (deterministic) test model $\mathbf{M}$. The model $\mathbf{M}$ is syntactically reproduced by a parser that has been qualified. Syntactic correctness of $\mathbf{M}$ can thus be assumed. Semantically, $\mathbf{M}$ can be interpreted as the (possibly infinite) set of (infinite) computation paths it defines, which we denote by $[\![\mathbf{M}]\!]$. Since $\mathbf{M}$ is deterministic, a path $\pi \in [\![\mathbf{M}]\!]$ is uniquely determined through its observable input-output behavior, i.e., partial (timed) assignments of the above form to the signals in $\mathcal{S}$.

Test Generation. Formally, a test generator is a function that computes a finite set of finite traces from $\mathbf{M}$, which we denote $[\![\mathbf{M}]\!]_{\mathsf{TC}}$. If correct, each $\pi_{\mathsf{TC}} \in [\![\mathbf{M}]\!]_{\mathsf{TC}}$ is the finite prefix of a computation path π through the test model (the prefix relation is denoted by $\pi_{\mathsf{TC}} \prec \pi$). Since the test generator is not qualified, and is thus an untrusted component, we have to assume that such a corresponding path does not necessarily exist. We define:

Definition 1. *Let* $\pi_{\mathsf{TC}} \in [\![\mathbf{M}]\!]_{\mathsf{TC}}$. *The predicate* $\mathsf{correct}$ *on* π_{TC} *is defined as:*

$$\mathsf{correct}(\pi_{\mathsf{TC}}) \Leftrightarrow \exists \pi \in [\![\mathbf{M}]\!] : (\pi_{\mathsf{TC}} \prec \pi)$$

Test Execution. Given $\pi_{\mathsf{TC}} \in [\![\mathbf{M}]\!]_{\mathsf{TC}}$, the test procedure that executes π_{TC} generates an execution log that can be interpreted as a finite trace π_{exec}. As before, we interpret the semantics $[\![\mathbf{S}]\!]$ of an SUT $\mathbf{S}$ as the set of its feasible execution traces; clearly, $\pi_{\mathsf{exec}} \in [\![\mathbf{S}]\!]$. Observe that π_{exec} is not required to be identical to π_{TC}: timings and floating point values may deviate within specified limits, and only a partial ordering of I/Os has to be observed as explained in Sect. 2.1. However, if π_{exec} conforms to π_{TC} the test passes, which we denote by $\mathsf{pass}_{\pi_{\mathsf{TC}}}(\pi_{\mathsf{exec}})$. Since the execution log is compared to a trace from an untrusted generator, we cannot infer correctness of the test execution with respect to $\mathbf{M}$.

Proposition 1. *Let* $\pi_{\mathsf{TC}} \in [\![\mathbf{M}]\!]_{\mathsf{TC}}$ *and* $\pi_{\mathsf{exec}} \in [\![\mathbf{S}]\!]$. *Then*

$$\big(\mathsf{correct}(\pi_{\mathsf{TC}}) \wedge \mathsf{pass}_{\pi_{\mathsf{TC}}}(\pi_{\mathsf{exec}})\big) \Rightarrow (\exists \pi \in [\![M]\!] : \mathsf{pass}_{\pi}(\pi_{\mathsf{exec}}))$$

Replay. Formally, the replay mechanism can be interpreted as a predicate replay:

Definition 2. *Let* $\pi_{\mathsf{exec}} \in [\![\mathbf{S}]\!]$ *denote an execution of* $\pi_{\mathsf{TC}} \in [\![\mathbf{M}]\!]_{\mathbf{TC}}$ *such that* $\mathsf{pass}_{\pi_{\mathsf{TC}}}(\pi_{\mathsf{exec}})$. *The predicate* $\mathsf{replay} : [\![\mathbf{S}]\!] \to \mathbb{B}$ *is defined as:*

$$\mathsf{replay}(\pi_{\mathsf{exec}}) \Leftrightarrow \exists \pi \in [\![\mathbf{M}]\!] : (\pi_{\mathsf{TC}} \prec \pi) \wedge \mathsf{pass}_{\pi_{\mathsf{TC}}}(\pi_{\mathsf{exec}})$$

Since correctness of the implementation of the replay mechanism is ensured, we safely assume that $\mathsf{replay}(\pi_{\mathsf{exec}}) = \mathsf{true}$ iff π_{exec} is the prefix of a path $\pi \in [\![\mathbf{M}]\!]$ such that $\mathsf{pass}_{\pi}(\pi_{\mathsf{exec}})$. We thus obtain the correctness argument:

Proposition 2. *Let* $\pi_{\mathsf{exec}} \in [\![\mathbf{S}]\!]$ *denote an execution of* $\pi_{\mathsf{TC}} \in [\![\mathbf{M}]\!]_{\mathbf{TC}}$ *such that* $\mathsf{pass}_{\pi_{\mathsf{TC}}}(\pi_{\mathsf{exec}})$. *Then,* $\mathsf{correct}(\pi_{\mathsf{TC}}) \Leftrightarrow \mathsf{replay}(\pi_{\mathsf{exec}})$.

In consequence, given a generated test $\pi_{\mathsf{TC}} \in [\![\mathbf{M}]\!]_{\mathsf{TC}}$ and an execution $\pi_{\mathsf{exec}} \in [\![\mathbf{S}]\!]$ that passes, π_{TC} is the finite prefix of a path $\pi \in [\![\mathbf{M}]\!]$ iff $\mathsf{replay}(\pi_{\mathsf{exec}})$ as desired, which ultimately provides a proof of correctness of the mechanism.

5 Compatibility with RTCA DO-178C

The avionic software development standard RTCA DO-178B [15] was one of the first standards to explicitly address tool qualification. The requirements are less strict than the ones imposed by ISO 26262 discussed so far in this paper. The updated standard RTCA DO-178C [16], however, devotes a whole companion standard [17] to tool qualification. When planning to use test and verification tools in both the automotive and the avionic domain it is thus useful to elaborate a consolidated qualification strategy that is consistent with both standards.

Project-specific tool qualification. The avionic standards emphasize that tool qualification cannot be unconditionally granted for any development or verification tool, but has to be performed with respect to a specific project [17, p. 5]. Practically speaking, certain tool components may only be universally qualified if it can be justified that these components' behavior does not depend on a specific project or target system to be developed. All other components have to be re-qualified for each development and verification campaign. For test automation tools this implies that the interfaces between test tool and SUT have to be specifically qualified, because correctly calculated test data may be passed along the wrong interfaces to the SUT. Conversely, erroneous SUT outputs may be passed along a wrong interface where the data appears to be correct. Moreover, it has to be verified that the interface-specific refinements and abstractions performed by the test automation tool (e. g., transforming abstract values used on model level to concrete communication telegrams and vice versa) are correct.

Tool criticality assessment. Both ISO26262 and RTCA DO-178C / DO-330 require that the qualification effort to be invested shall depend on the impact that tool malfunctions could have on the target system under consideration. The automotive and the avionic standards, however, differ in one crucial aspect. ISO26262 classifies the criticality of a tool alone on the basis whether erroneous tool behavior may result in erroneous target system behavior, and whether tool malfunction can be detected with high or low confidence [6, Sect. 11.4.3.2]. RTCA DO-178C distinguishes between development tools whose outputs become part of the airborne software and verification tools (including test automation tools) whose malfunction could only lead to an error in the target system remaining undetected (criteria 1 and 2 in [16, Sect. 12.2.2]).

As a consequence, DO-178C assigns only *tool qualification level* **TQL4** to tools that automate verification and test of software of the highest criticality (i.e., Level A) [16, Tab. 12-1]. This requires the elaboration of operational and functional requirements and their verification, the verification of protection mechanisms, and requirements-based testing [16, Tab. T-0 – T-7]. Yet, neither tests against the detailed design, nor code coverage of any measure are required.

By way of contrast, ISO26262 assigns **TCL1** to tools classified by **TI1** and **TD1**, as the one considered in this paper [6, Sect. 11.4.3.2]. Since no qualification whatsoever is required for **TCL1**, no requirements-based testing is is needed. It remains to prove, however, that the tool indeed fulfills **TD1** (*"there is a high degree of confidence that a malfunction or an erroneous output from the software tool will be prevented or detected"*, [6, Sect. 11.4.3.2, b)]). It is remarkable, though that the standard does not elaborate on how error detection should be verified. From the general qualification requirements [6, Sect. 11.], however, we conclude that this should be done with the highest possible effort associated with the target system's criticality. For the highest criticality level (denoted ASIL D) the standard requires alternatively the evaluation of the development process in combination with a comprehensive validation or — this is the procedure applied for RTT-MBT in this paper — development in compliance with a safety standard, such as RTCA DO-178B [6, p. 23, Tab. 2, Ex. 3]. This implies, e.g., that detailed tests to achieve MC/DC code coverage have to be performed for the replay.

We conclude that the tool qualification requirements of ISO26262 and RTCA DO-178C / DO-330 are complementary in the sense that the former put emphasis on an in-depth verification of the error detection mechanism with highest confidence, while the latter requires comprehensive requirements-based testing.

6 Related Work

The idea of replaying a test execution in a simulator is, of course, not new. The overall approach is frequently referred to as the *capture and replay* paradigm, and has long been studied in different contexts such as testing of concurrent programs [2]. However, the classical approach of capture/replay testing is to capture user-interaction and then replay the recorded inputs within test cases, as opposed to automatic test-case generation. This paradigm differs from the one implemented in RTT-MBT, although we integrate a replay function into our work to detect deviations of a generated test case from the test model. To our best knowledge, our approach is the first to combine replay with model-based methods for error detection within a test-case generator. Our contribution is not a theoretical one, but comes from an industrial perspective. Tool qualification is compulsory for software that is applied in development processes of safety-critical systems. For ISO 26262, the tool qualification requirements in a general context were recently studied by Hillebrand et al. [5]. Most relevant to our work, the authors study verification measures for error detection, which are classified as *prevention*, *review*, and *test* [5, Sect. 4.6]. According to this classification, our approach falls into the categories *review* and *test*, as the results of the test-case generator undergo review and are automatically tested against the test model.

Recently, there has been impressive progress on verified compilers [9,11] that, in theory, do not require further qualification measures. França et al. [4] report on their experiences of introducing the verified CompCert compiler into development processes for airborne software. Yet, it is important to note that even a verified compiler contains non-verified components, which may introduce bugs. Indeed, Regehr [14] discovered defects in the CompCert front-end responsible for type-checking, thereby showing that proofs of functional correctness of core components do not provide sufficient evidence to construct an overall correctness argument. Our approach can be seen as a practical response to this situation since we evaluate the correctness of the outputs on a *per test* basis.

A notable difference between ISO 26262 and RTCA DO-178C is that the latter standard distinguishes between tools that mutate the software (such as code generators) and those that only analyze it (such as stack analyzers). The qualification measures imposed onto analyzers or test-case generators are less strict in RTCA DO-178C. Further details on this issue in the context of RTCA DO-178C (that are likewise applicable to DO-178B), and also additional information about the qualification process for formal verification tools, are given by Souyris et al. [18]. This recent paper can be seen as a wrap-up of a paper by the same authors that studied the same problem more than a decade ago [13], and also contains details about how different formal methods tools are used within Airbus. Qualification for RTCA DO-178B was mentioned earlier by Blackburn and Busser [1]. There, the authors describe the tool T-Vec, which is used in the qualification process to test itself. However, rather than using automatic replay, they manually define expected outputs and compare them to derived test procedures [1, Sect. 5]. By way of contrast, our approach delegates the manual specification of expected outputs to the design of the test model — a step that is necessary in model-based testing anyway — and the execution of test procedures.

7 Concluding Discussion

This paper advocates the use of replay for tool error detection in model-based test generators as a key mechanism for qualifying such a tool according to ISO 26262, since it provides trustworthy, yet simple and cost effective, error detection. By providing full tool error detection capabilities, the approach thus smoothly integrates with the requirements of ISO 26262 for the highest tool confidence level. It is noteworthy, however, that the mechanism does not ensure the absence of errors in the entire tool, but only in the functionality that is indeed used.

We have to point out, however, that the qualification achieved for ISO 26262 cannot be directly used to gain qualification credit according to RTCA DO-178C: the latter standard requires comprehensive requirements-based testing of *all* tool capabilities, while being less strict with respect to the verification of error detection mechanisms, which are considered simply as tool capabilities to be verified by means of robustness tests. The verification techniques to achieve this are the same as the ones used for qualification according to ISO 26262.

References

1. Blackburn, M.R., Busser, R.D.: T-VEC: A Tool for Developing Critical Systems. In: Compass, pp. 237–249. IEEE Computer Society Press (1996)
2. Carver, R.H., Tai, K.C.: Replay and Testing for Concurrent Programs. IEEE Software 8(2), 66–74 (1991)
3. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL, pp. 238–252. ACM Press (1977)
4. França, R.B., Favre-Felix, D., Leroy, X., Pantel, M., Souyris, J.: Towards Formally Verified Optimizing Compilation in Flight Control Software. In: PPES. OASICS, vol. 18, pp. 5–9–68. Schloss Dagstuhl (2011)
5. Hillebrand, J., Reichenpfader, P., Mandic, I., Siegl, H., Peer, C.: Establishing Confidence in the Usage of Software Tools in Context of ISO 26262. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) SAFECOMP 2011. LNCS, vol. 6894, pp. 257–269. Springer, Heidelberg (2011)
6. International Organization for Standardization. ISO 26262 - Road Vehicles - Functional Safety - Part 8: Supporting Processes. ICS 43.040.10 (2009)
7. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal Verification of an Operating-System Kernel. Commun. ACM 53(6), 107–115 (2010)
8. Kroening, D., Strichman, O.: Decision Procedures. Springer (2008)
9. Leroy, X.: Formal Verification of a Realistic Compiler. Commun. ACM 52(7), 107–115 (2009)
10. Löding, H., Peleska, J.: Timed Moore Automata: Test Data Generation and Model Checking. In: ICST, pp. 449–458. IEEE Computer Society (2010)
11. Myreen, M.O.: Verified Just-in-Time Compiler on x86. In: POPL, pp. 107–118. ACM (2010)
12. Peleska, J., Vorobev, E., Lapschies, F.: Automated Test Case Generation with SMT-Solving and Abstract Interpretation. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 298–312. Springer, Heidelberg (2011)
13. Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., Schoen, D.: Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1798–1815. Springer, Heidelberg (1999)
14. Regehr, J.: The Future of Compiler Correctness (2010), `http://blog.regehr.org/archives/249`
15. RTCA SC-167/EUROCAE WG-12. Software Considerations in Airborne Systems and Equipment Certification. Number RTCA/DO-178B. RTCA, Inc., 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036 (December 1992)
16. RTCA SC-205/EUROCAE WG-71. Software Considerations in Airborne Systems and Equipment Certification. Number RTCA/DO-178C. RTCA, Inc., 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036 (December 2011)
17. RTCA SC-205/EUROCAE WG-71. Software Tool Qualification Considerations. Number RTCA/DO-330. RTCA, Inc. (December 2011)
18. Souyris, J., Wiels, V., Delmas, D., Delseny, H.: Formal Verification of Avionics Software Products. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 532–546. Springer, Heidelberg (2009)