Complete Test of Synthesised Safety Supervisors for Robots and Autonomous Systems*

Mario Gleirscher and Jan Peleska
Department of Mathematics & Computer Science, University of Bremen, Germany
{gleirscher,peleska}@uni-bremen.de

Verified controller synthesis uses world models that comprise all potential behaviours of humans, robots, further equipment, and the controller to be synthesised. A world model enables quantitative risk assessment, for example, by stochastic model checking. Such a model describes a range of controller behaviours some of which—when implemented correctly—guarantee that the overall risk in the actual world is acceptable, provided that the stochastic assumptions have been made to the safe side. Synthesis then selects an acceptable-risk controller behaviour. However, because of crossing abstraction, formalism, and tool boundaries, verified synthesis for robots and autonomous systems has to be accompanied by rigorous testing. In general, standards and regulations for safety-critical systems require testing as a key element to obtain certification credit before entry into service. This work-in-progress paper presents an approach to the complete testing of synthesised supervisory controllers that enforce safety properties in domains such as human-robot collaboration and autonomous driving. Controller code is generated from the selected controller behaviour. The code generator, however, is hard, if not infeasible, to verify in a formal and comprehensive way. Instead, utilising testing, an abstract test reference is generated, a symbolic finite state machine with simpler semantics than code semantics. From this reference, a complete test suite is derived and applied to demonstrate the observational equivalence between the synthesised abstract test reference and the generated concrete controller code running on a control system platform.

1 Introduction

In verified controller synthesis, world models are used that comprise all potential behaviours of humans, robots, further equipment, and the controller to be synthesised. A world model enables quantitative risk assessment, for example, by stochastic model checking. Such a model describes a range of controller behaviours some of which—when implemented correctly—guarantee that the overall risk in the actual world is acceptable, provided that the stochastic assumptions have been made to the safe side. The objective of the synthesis step is to select a *controller behaviour* from this range that meets requirements given as constraints, for example, to stay within an acceptable risk bound. Within such constraints, the synthesis can optimise further objectives, for example, maximal performance or minimal cost and risk. Because of crossing the boundaries between different abstractions, formalisms, and tools, verified controller synthesis for safety-critical systems naturally has to be accompanied by rigorous testing. Indeed, standards and regulations for safety-critical systems (e.g. [17, 16, 30, 31]) require testing as a key element to obtain certification credit before entry into service. Hence, a key methodological aim is to bridge the gap between verified controller synthesis and the generation of executable code that is being deployed on a control system platform and integrated into the wider system to be put into service.

*The second author is partially funded by the German Ministry of Economics, Grant Agreement 20X1908E.

M. Farrell & M. Luckcuck (Eds.):
Third Workshop on Formal Methods
for Autonomous Systems (FMAS2021)
EPTCS 348, 2021, pp. 101–109, doi:10.4204/EPTCS.348.7

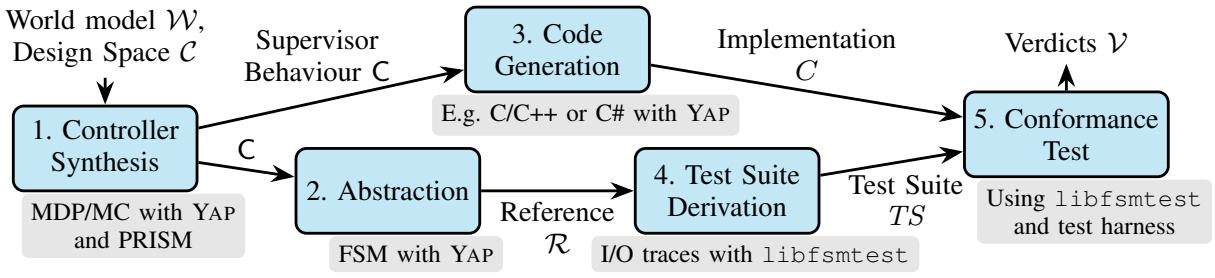

Figure 1: Workflow and artefacts of the proposed tool-supported approach to complete testing

Approach. Following this aim, we propose an *integrated formal approach to the complete testing of synthesised supervisory discrete-event controllers that enforce safety properties* in domains such as human-robot collaboration and autonomous driving. Our tool-supported approach works as follows.

1. Controller Synthesis. The verified synthesis step is based on policy synthesis for Markov decision processes [19, 18]. A conceptual *world model* $\mathcal{W}$ is constructed that defines all the behaviours of all the relevant actors (e.g. humans, robots, other equipment) and the controller under consideration. The range of controller behaviours are denoted as the controller *design space* $\mathcal{C}$. Then, the relevant temporal logic properties are formally verified of $\mathcal{C}$ and an appropriate (optimal) *controller behaviour* $\mathsf{C} \in \mathcal{C}$ is selected (synthesised) from $\mathcal{C}$. For this step, we adopt the approach described in [11, 10].

2. Abstraction. Then, the selected (verified) controller behaviour C is abstracted into a *test reference* model $\mathcal{R}$. This model is described as a *symbolic finite state machine (SFSM)* [22], where the control states are called *risk states*. Symbols correspond to subsets of $\mathcal{W}$'s state space. The *input* alphabet corresponds to the events monitored (observed) by the controller, the *output* alphabet to the signals that the controller can issue to $\mathcal{W}$ as the controlled process. An event is triggered by a *guard condition*, whose input valuation changes from `false` to `true`, so that a transition labelled with (or fulfilling) this guard can be taken. Transitions of $\mathcal{R}$ are labelled with such input/output (I/O) pairs and derived from C.

3. Code Generation. C is also translated into a software component C executable on the control system platform of a robotic or autonomous system. Following an embedded systems tradition, we use C/C++ as the target language for C, making the reasonable assumption that the used type of FSMs has a simpler semantics than the executable code. Abstraction and code generation are explained in Sect. 3.

4. Test Suite Derivation. Using the H-Method [6], in this step, a complete test suite TS for I/O conformance testing is derived for a finite state machine (FSM) abstraction of $\mathcal{R}$. This abstraction maps the SFSM guard conditions to atomic input labels; otherwise it adopts the SFSM structure without changes. It has been shown in [14, 15] that complete FSM test suites can be mapped to likewise complete suites on SFSMs, when the FSM input events e are considered as input equivalence classes of the SFSM, and each e is refined to a concrete SFSM input data tuple solving the equivalence class constraint (this is just a refined guard condition).

5. Conformance Test. Based on a generated test harness emulating the target platform, the test suite TS is run against C to record outputs and obtain a complete set of verdicts $\mathcal{V}$. A complete pass shown by the verdicts demonstrates the *observational equivalence* between the test reference $\mathcal{R}$ and the controller code C. Test suite derivation and conformance test execution are explained in Sect. 4. There, it is also explained how potential errors in the reference model $\mathcal{R}$, the test suite generator, or the test harness can be uncovered. This is required according to standards for safety-critical control applications (see, e.g. [30, 31]), because faulty tool chains might mask "real" errors in the software under test.

Related Work. In the rich body of literature on verified controller synthesis, the approaches in [21, 3] from collaborative robotics are perhaps closest to the one presented here as they include a platform deployment stage. While these authors focus on the synthesis of complete robot controllers, our approach focuses on safety supervisors but includes a testing step reassuring the correctness of platform code generation. The authors of [29] propose a general integration of quantitative model checking (with UPPAAL [1]) with model-based conformance testing and fault injection. Apart from using the switch cover method for test suite generation, their approach is highly similar to our Mealy-type test reference generation, conformance testing, and mutation approach for test suite evaluation. However, while their focus is more on cross-validation of UPPAAL and FSM models, we concentrate on code robustness tests, assuming that $\mathscr{W}$ has been validated and verified beforehand.

The investigation of complete testing methods is a very active research field [23]. The H-Method [6] applied for testing in this paper has been selected because (1) it produces far less test cases than the "classical" W-Method [5], but (2) it is still very intuitive with regard to the test case selection principles. This facilitates the qualification of the test case generator, as discussed in Section 4. If the main objective of a testing campaign was just to provide complete suites with a minimal number of test cases, then the SPYH-Method [26] should be preferred to the H-Method.

Whereas hazard- or failure-oriented testing [8, 20] and requirements falsification based on negative scenarios [28, 9, 27] are highly useful if no complete $\mathscr{R}$ is available or if $\mathscr{R}$ still needs to be validated and revised, our approach is complete once $\mathscr{R}$ is successfully validated. That is, any deviation from $\mathscr{R}$ detectable by these techniques is also uncovered by at least one test case generated by our approach. Moreover, our approach is usable to test controller robustness without a realistic simulator for $\mathscr{W}$.

Contribution. We propose a solution to the generation of well-defined test references used in techniques such as the H-Method [6]. In particular, we connect test reference generation with the H-Method to derive complete test suites and demonstrate that this form of robustness testing yields a correctness proof of a controller under certain assumptions. We provide tool support for both these steps. Our proof of concept indicates that complete test suites are a feasible and practically attractive means to verify correctness of implementations of the considered class of discrete-event control modules. In Sect. 2, we explain the safety supervisor concept by means of an example. In Sects. 3 and 4, we explain code and test reference generation and test suite derivation. We add concluding remarks in Sect. 5.

2 The Safety Supervisor Concept with an Illustrative Example

To illustrate our approach, we reuse our example from the domain of human-robot collaboration in industrial manufacturing as discussed in [10, 11]. In this example, a human operator collaborates with a robot on a welding and assembly *task* in a work cell equipped with a spot welder. This setting involves several actors performing potentially dangerous actions (e.g. robot arm movements, welding steps) and, thus, implies the reaching of hazardous states (e.g. operator near the active spot welder, *HC*, or operator and robot on the work bench, *HRW*). Such states need to be either avoided or reacted to in order to prevent accidents from happening or at least to reduce the likelihood of such undesired events.

This task of *risk mitigation* is, by design, put under the responsibility of a supervisory discrete-event controller *C*. This controller is supposed to enforce probabilistic safety properties of the kind "the probability of an accident a is less than pr_a" or "hazard h happens less likely than pr_h". The underlying conceptual controller behaviour C comprises (i) the detection of critical events, (ii) the performance of corresponding *mitigation* actions to react to such events and reach a *safe* risk state, and (iii) , avoiding a

paused task or degraded task performance, the execution of *resumption* actions to resolve the event and to return to a safe but productive risk state. For the sake of brevity, we call C a *safety supervisor*.

3 Derivation of the Software Module and the Test Reference

We summarise [11] on how to obtain the world model $\mathscr{W}$, the controller design space $\mathscr{C}$, and the controller behaviour C. Then, we describe in more detail the generation of the controller software component C (for deployment) and the abstraction into the test reference $\mathscr{R}$ (for test suite derivation, see Sect. 4).

The world model $\mathscr{W}$ is a Markov decision process (MDP), the result of a fixed-point application of actions given as probabilistic guarded commands to an initial state of $\mathscr{W}$ [19]. MDPs are models containing *uncertainties* about aspects not under control (or agency) or not to be modelled explicitly. The world state space is defined using a set V of finite-sorted variables. The MDP is a labelled transition system where the transition relation encodes non-deterministic and probabilistic choice in a compound manner and states are labelled with atomic propositions holding of V's valuations defining the states. Non-deterministic decisions encode freedom of choice of the actors in $\mathscr{W}$, in particular, the controller design space $\mathscr{C}$. This freedom can be resolved by picking an *appropriate policy*, a choice resolution for each state, with the result of obtaining a Markov chain (MC), a labelled transition system without indeterminacy in the controller (and the other considered actors). Policy appropriateness can be thought of as sub-setting $\mathscr{C}$ and is defined by constraints to be specified in probabilistic computation tree logic [19]. The resulting MC is verified against these constraints and includes the selected behaviour $\mathsf{C} \in \mathscr{C}$.

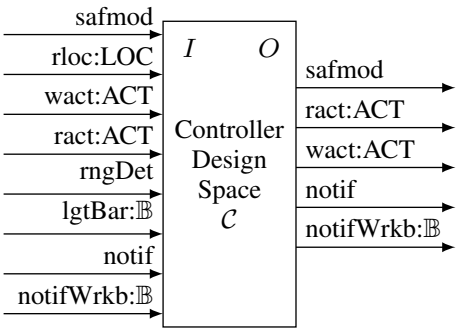

Figure 2: Interface between $\mathscr{W}$ and $\mathscr{C}$

Now, C has to be translated into the two forms C and $\mathscr{R}$. For this step, we define the variables $I \subseteq V$ to be monitored and the variables $O \subseteq V$ that can be controlled, resulting in what we call the *syntactic interface* (alphabet) $\Sigma \subseteq \mathsf{type}\, I \times \mathsf{type}\, O$[1] of $\mathscr{C}$ (see Figure 2) [4]. This interface defines the nature of the changes in $\mathscr{W}$ that any $\mathsf{C} \in \mathscr{C}$ can observe and perform.

The control states of both C and $\mathscr{R}$ are derived from the notion of *risk states* [12], which is defined over a set $F \subset V$ of $\mathbb{P}$-sorted variables modelling the critical events considered in $\mathscr{W}$ as *risk factors*. We require $(I \cup O) \cap F = \emptyset$. The sort $\mathbb{P} = \{0, a, m\}$ models life-cycle stages for handling a factor $f \in F$ (Figure 3), for example, from *inactive* (0), *active* (fa), and *mitigated* (fm) back to *inactive* [12]. In the example in Sect. 2, we consider three factors, hence $F = \{HS, HC, HRW\}$. Each C can then be associated with a control state space $S \subseteq \mathbb{P}^{|F|}$.

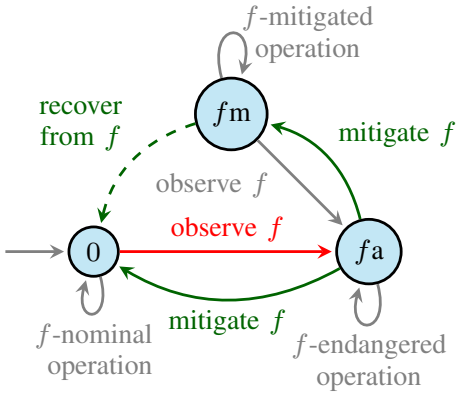

Figure 3: Phase transitions of factor f

We then translate the controller fragment C of the MC transition relation (resulting from policy synthesis over $\mathscr{W}$) into C++ code. Basically, every transition of C is translated into a guarded action $[a]\, i \wedge r\colon (o, r') \leftarrow \mathrm{STEP}(i, r)$ with $r, r' \in S, (i, o) \in \Sigma$ and action name a derived from F, r, and r'. For that, the source state of each transition is mapped into two parts: one corresponding to the input i (the

[1] $\mathsf{type}\, S$ of a set S of sorted variables returns the set of tuples in the Cartesian product of the sorts of the variables in S.

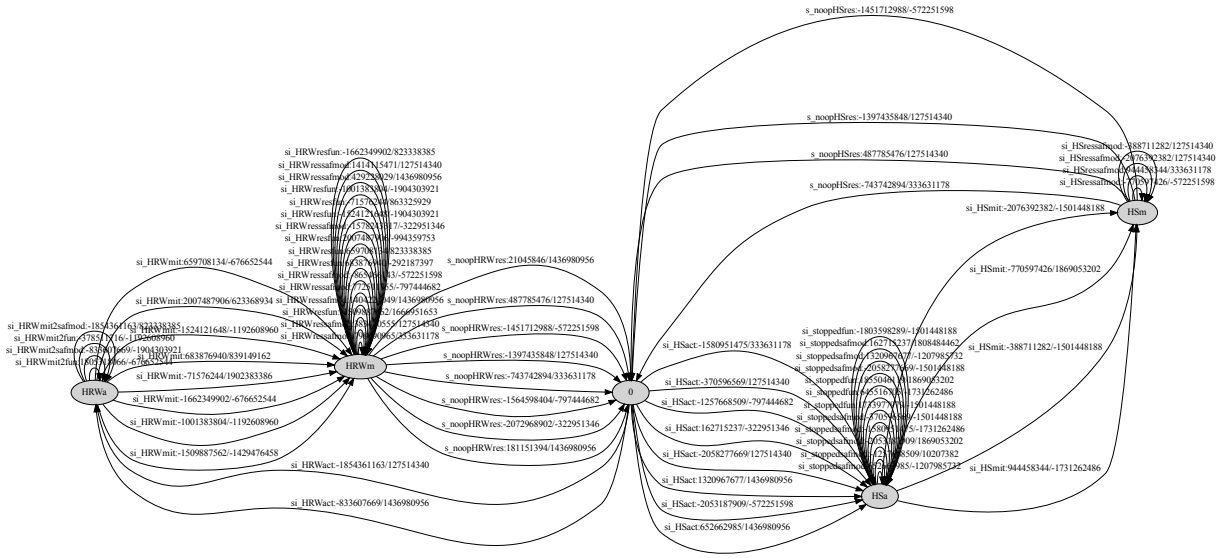

Figure 4: Visual representation of $\mathscr{R}$. Nodes are risk states in S, edges transitions in T. Because valuation expressions for $(i,o) \in \Sigma$ are too long, edge labels are hashed and prefixed with action names for readability, following the pattern $a : h(i)/h(o)$ with action name a and an integer-valued hash function h.

observed event) and one corresponding to a risk state r. The control and state updates o and r' to be associated with this action are derived from the difference in the controlled variables $O \cap F$ between source and target states. C implements Algorithm 1, intentionally simple (not using action names) and wrapped into platform-specific code (not shown) for data processing and communication.

In order to obtain $\mathscr{R}$, we then translate the C-fragment of this transition relation into a deterministic Mealy-type FSM $\mathscr{R} = (S, \Sigma, T, \bar{s})$ with the state space S, the transition relation $T \subseteq S \times \Sigma \times S$, and the initial state $\bar{s} \in S$ being congruent with the one in $\mathscr{W}$. Figure 4 shows $\mathscr{R}$ for the example in Sect. 2 and is an operation refinement of the composition of the factors (Figure 3) of F. The translations into C and $\mathscr{R}$ including the generation of the test harness are carried through with the YAP tool.[2]

Algorithm 1 Safety supervisor

```
1: procedure CTR(in Event i, out Signal o)
2:     r ← INIT                          ▷ init. control/risk state
3:     while true do                     ▷ should implement T
4:         []_{i∈I, r∈S} i ∧ r: (o, r') ← STEP(i, r)
```

Regarding the difference of C and $\mathscr{R}$, the semantics of C can vary significantly. $\mathscr{R}$ is converted into an input format suitable for test suite generation via `libfsmtest` [2]. Here, we consider a C++ component for a low-level real-time implementation, for example, an FPGA synthesised from VHDL or Verilog HDL[3] generated from C++. In [11], we consider a C# component used in a simulation of $\mathscr{W}$ in a "Robot Operating System"-enabled digital twinning environment. While the semantics of the C++ and C# implementations C_{C++} and $C_{C\#}$ may dramatically differ, $\mathscr{R}$ can be shared between the two. The only difference on the testing side is in the mappings used in the test harness (Sect. 4) to deliver the inputs to C and record the outputs of C.

[2]The discussed features are available in YAP version 0.8+, see `https://yap.gleirscher.at`.

[3]Field-programmable gate array (FPGA); VHSIC or Verilog hardware description language (VHDL or Verilog HDL)

4 Complete Test Strategy

In this section, we briefly describe the characteristics of complete test suites, sketch their derivation, outline the chosen recipe for test suite derivation (Sect. 4.1), and discuss typical error possibilities to be taken into account during standards-oriented controller and tool certification (Sect. 4.2).

4.1 Strategy Application

A test suite is complete if—under certain hypotheses—it guarantees that non-conforming implementations will fail in at least one test case, while equivalent implementations will always pass the suite [15]. Here, these hypotheses are (a) the number of control states implemented in C, and (b) assumptions about potential mutations of guards and output assignments. Since the verification of safety-critical systems requires code to be open source for analyses, these hypotheses can be checked using static analysis. We check that C has the same number of control states as $\mathscr{R}$, and it is checked that the guard conditions in $\mathscr{R}$ have been correctly translated to corresponding branching conditions in C (cf. Line 4 in Algorithm 1).

Since the reference $\mathscr{R}$ is modelled as an SFSM, we need a method to construct complete test suites for SFSMs. We follow the recipe from [14, 15] which allows us to use test generation strategies for the simpler class of FSMs and translate the resulting test suite to an SFSM suite as follows: (1) For the SFSM, input equivalence classes are calculated. This is performed by creating all conjuncts of positive and negated SFSM guard conditions which have at least one solution. (2) An FSM is created as an abstraction of $\mathscr{R}$. The input alphabet of this FSM consists of the identifiers for the input equivalence classes calculated in (1). Control states, output events, and transition arrows are directly adopted from the SFSM. (3) For this FSM, a complete test suite is created using the H-Method [6]. Its test cases consist of input traces, where each input is an identifier of an SFSM input equivalence class. The expected results are obtained by running this input trace against the FSM. (4) The FSM test suite is *refined* to an SFSM test suite by calculating concrete input representatives from the constrains specifying the referenced input classes. (5) The concrete SFSM test suite is executed in a *test harness*: this is an executable running the test cases one by one against C and checking its responses against the FSM test oracle.

The theory elaborated in [14, 15] confirms that the concrete SFSM test suite is also complete, if this holds for the abstract FSM test suite. Since we know that C has the same number of control states and the same guards as $\mathscr{R}$, passing the test suite is equivalent to *proving* observational equivalence between C and $\mathscr{R}$. For tool support, the `libfsmtest` library [2] is used which provides an implementation of the H-Method and a template for the test harness.

4.2 Verification of Verification Results

For automated verification/testing of safety-critical system components, applicable standards require a verification that the tool chain involved does not mask any errors inside C. This process is usually called *verification of the verification results*. We consider the possible errors in the testing environment one by one. (1) Error in the generation of $\mathscr{R}$: The complete test suite created as described above characterises $\mathscr{R}$ up to observational equivalence. By checking if the test suite is compatible with the computations of $\mathscr{W}$, it is shown that $\mathscr{R}$ is correct. (2) Error in the testing theory: It has been shown in [25] that methods of similar complexity as the H-Method can be mechanically verified using a proof assistant (e.g. Isabelle/HOL). (3) H-Method implementation error: Here, we have two options: in [24] it has been demonstrated that correct algorithms can be generated while proving a testing theory to be correct. Alternatively, the generated test suite can be checked automatically for completeness: from the specification

of the test cases required for the H-Method given in [13], a checking tool can be derived which verifies that the generated suite really contains the test cases required according to the theory. This checking algorithm would be *orthogonal* to the test generation algorithm. This means that it is highly unlikely that test generator and completeness checker could contain complementary errors masking each other out. (4) Test harness error: The test harness could execute the suite in a faulty way that masks an error in C. To ensure that this is not the case, we apply *mutation testing*. Using the `clang` compiler functions for static code analysis, a set $\mathscr{M}_C$ of mutants of C is created in an automated way. Then it is checked for each mutant in $\mathscr{M}_C$ if it is uncovered by the test suite, or if it is semantically equivalent to the original version of C.

5 Conclusions

We outlined an approach to the complete testing of synthesised discrete-event controllers that enforce safety properties in applications such as human-robot collaboration and autonomous driving. Our aim is to bridge the gap between verified controller synthesis and certified deployment of executable code. We illustrate our approach with a human-robot collaboration example where a safety supervisor makes autonomous decisions on when and how to mitigate hazards and resume normal operation. We check the specificity of the test reference $\mathscr{R}$ and the strength of the corresponding test suite TS by mutation of the generated code C, modulo semantic equivalence over $\mathscr{M}_C$. We contribute a preliminary synthesis-based test strategy that allows one to show total correctness of C under certain implementation assumptions. The presented approach is automated in a tool chain: YAP and a stochastic model checker (e.g. PRISM [19]) for MDP generation and verification, YAP for test reference and code generation, and `libfsmtest` [2] for test suite derivation. For testing the integrated system (robot, welding machine, safety supervisor and simulation of human interactions), the approach presented here is embedded into a more general methodology for verification and validation of robots and autonomous systems, starting at the module level considered here, and ending at the level of the integrated overall system [7].

References

[1] Gerd Behrmann, Alexandre David & Kim Guldstrand Larsen (2004): *A Tutorial on UPPAAL*. In: *SFM*, pp. 200–236, doi:10.1007/978-3-540-30080-9_7.

[2] Moritz Bergenthal, Niklas Krafczyk, Jan Peleska & Robert Sachtleben (2021): *libfsmtest – An Open Source Library for FSM-based Testing*. Available at `https://bitbucket.org/JanPeleska/libfsmtest`.

[3] Marcello M. Bersani, Matteo Soldo, Claudio Menghi, Patrizio Pelliccione & Matteo Rossi (2020): *PuRSUE – From specification of robotic environments to synthesis of controllers*. Formal Aspects of Computing 32(2-3), pp. 187–227, doi:10.1007/s00165-020-00509-0.

[4] Manfred Broy (2010): *A Logical Basis for Component-Oriented Software and Systems Engineering*. The Computer Journal 53(10), pp. 1758–82, doi:10.1093/comjnl/bxq005.

[5] Tsun S. Chow (1978): *Testing Software Design Modeled by Finite-State Machines*. IEEE Transactions on Software Engineering SE-4(3), pp. 178–186, doi:10.1109/TSE.1978.231496.

[6] Rita Dorofeeva, Khaled El-Fakih & Nina Yevtushenko (2005): *An Improved Conformance Testing Method*. In Farn Wang, editor: *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, Lecture Notes in Computer Science 3731, Springer, pp. 204–218, doi:10.1007/11562436_16.

[7] Kerstin Eder, Wen-ling Huang & Jan Peleska (2021): *Complete Agent-driven Model-based System Testing for Autonomous Systems*. In Matt Luckuck & Marie Farrell, editors: *Formal Methods for Autonomous Systems (FMAS), 3rd Workshop*. To appear in EPTCS.

[8] Mario Gleirscher (2011): *Hazard-based Selection of Test Cases*. In Antonia Bertolino, Howard Foster & J. Jenny Li, editors: *Automation of Software Test (AST), 6th ICSE Workshop*, ACM, Honolulu, HI, pp. 64–70, doi:10.1145/1982595.1982609.

[9] Mario Gleirscher (2014): *Behavioral Safety of Technical Systems*. Dissertation, Technical University of Munich. Available at `http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20141120-1221841-0-1`.

[10] Mario Gleirscher & Radu Calinescu (2020): *Safety Controller Synthesis for Collaborative Robots*. In Yi Li & Alan Liew, editors: *Engineering of Complex Computer Systems (ICECCS), 25th Int. Conf., Singapore*, ACM, pp. 83–92, doi:10.1109/ICECCS51672.2020.00017.

[11] Mario Gleirscher, Radu Calinescu, James Douthwaite, Benjamin Lesage, Colin Paterson, Jonathan Aitken, Robert Alexander & James Law (2021): *Verified Synthesis of Optimal Safety Controllers for Human-Robot Collaboration*. Working paper, University of York, University of Sheffield, and University of Bremen. Available at `https://arxiv.org/abs/2106.06604`.

[12] Mario Gleirscher, Radu Calinescu & Jim Woodcock (2021): *Risk Structures: A Design Algebra for Risk-Aware Machines*. *Formal Aspects of Computing* 33, pp. 763–802, doi:10.1007/s00165-021-00545-4.

[13] Wen-ling Huang, Sadik Özoguz & Jan Peleska (2019): *Safety-complete test suites*. *Software Quality Journal* 27(2), pp. 589–613, doi:10.1007/s11219-018-9421-y.

[14] Wen-ling Huang & Jan Peleska (2016): *Complete model-based equivalence class testing*. *Software Tools for Technology Transfer* 18(3), pp. 265–283, doi:10.1007/s10009-014-0356-8.

[15] Wen-ling Huang & Jan Peleska (2017): *Complete model-based equivalence class testing for nondeterministic systems*. *Formal Aspects of Computing* 29(2), pp. 335–364, doi:10.1007/s00165-016-0402-2.

[16] ISO 26262 (2011): *Road Vehicles – Functional Safety*. Standard, ISO/TC 22/SC 32. Available at `https://www.iso.org/standard/43464.html`.

[17] ISO/TS 15066 (2016): *ISO/TS 15066:2016 – Robots and robotic devices – Collaborative robots*. Standard, International Organization for Standardization, Geneva, CH.

[18] Marta Kwiatkowska, Gethin Norman & David Parker (2007): *Stochastic Model Checking*. In M. Bernardo & J. Hillston, editors: *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM)*, *LNCS* 4486, Springer, pp. 220–70, doi:10.1007/978-3-540-72522-0_6.

[19] Marta Kwiatkowska, Gethin Norman & David Parker (2011): *PRISM 4.0: Verification of Probabilistic Real-time Systems*. In G. Gopalakrishnan & S. Qadeer, editors: *23rd International Conference on Computer Aided Verification (CAV)*, LNCS, Springer, pp. 585–591, doi:10.1007/978-3-642-22110-1_47.

[20] Benjamin Lesage & Rob Alexander (2021): *SASSI: Safety Analysis using Simulation-based Situation Coverage for Cobot Systems*. In: *Computer Safety, Reliability, and Security (SAFECOMP), 40th Int. Conf., LNCS* 12852, Springer, pp. 195–209, doi:10.1007/978-3-030-83903-1_13.

[21] Andrea Orlandini, Marco Suriano, Amedeo Cesta & Alberto Finzi (2013): *Controller Synthesis for Safety Critical Planning*. In Judy Luo, editor: *Tools with Artificial Intelligence (ICTAI), IEEE 25th Int. Conf.*, IEEE, pp. 1–8, doi:10.1109/ictai.2013.54.

[22] Alexandre Petrenko (2016): *Checking Experiments for Symbolic Input/Output Finite State Machines*. In: *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*, IEEE Computer Society, pp. 229–237, doi:10.1109/ICSTW.2016.9. Available at `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7517740`.

[23] Alexandre Petrenko, Adenilso Simao & José Carlos Maldonado (2012): *Model-based Testing of Software and Systems: Recent Advances and Challenges*. *Int. J. Softw. Tools Technol. Transf.* 14(4), pp. 383–386, doi:10.1007/s10009-012-0240-3.

[24] Robert Sachtleben (2020): *An Executable Mechanised Formalisation of an Adaptive State Counting Algorithm*. In Valentina Casola, Alessandra De Benedictis & Massimiliano Rak, editors: *Testing Software and Systems - 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9-11, 2020, Proceedings*, *Lecture Notes in Computer Science* 12543, Springer, pp. 236–254, doi:10.1007/978-3-030-64881-7_15.

[25] Robert Sachtleben, Robert M. Hierons, Wen-ling Huang & Jan Peleska (2019): *A Mechanised Proof of an Adaptive State Counting Algorithm*. In Christophe Gaston, Nikolai Kosmatov & Pascale Le Gall, editors: *Testing Software and Systems - 31st IFIP WG 6.1 International Conference, ICTSS 2019, Paris, France, October 15-17, 2019, Proceedings*, *Lecture Notes in Computer Science* 11812, Springer, pp. 176–193, doi:10.1007/978-3-030-31280-0_11.

[26] Michal Soucha & Kirill Bogdanov (2018): *SPYH-Method: An Improvement in Testing of Finite-State Machines*. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, IEEE Computer Society, pp. 194–203, doi:10.1109/ICSTW.2018.00050.

[27] Viktoria Stenkova, Jennifer Brings, Marian Daun & Thorsten Weyer (2019): *Generic Negative Scenarios for the Specification of Collaborative Cyber-Physical Systems*. In: *Conceptual Modeling*, *LNCS* 11788, Springer, pp. 412–419, doi:10.1007/978-3-030-33223-5_34.

[28] Sebastian Uchitel, Jeff Kramer & Jeff Magee (2002): *Negative scenarios for implied scenario elicitation*. *ACM SIGSOFT Software Engineering Notes* 27(6), pp. 109–118, doi:10.1145/605466.605484.

[29] Emília Villani, Rodrigo Pastl Pontes, Guilherme Kisselofl Coracini & Ana Maria Ambrósio (2019): *Integrating model checking and model based testing for industrial software development*. *Computers in Industry* 104, pp. 88–102, doi:10.1016/j.compind.2018.08.003.

[30] RTCA SC-205/EUROCAE WG-71 (2011): *Software Considerations in Airborne Systems and Equipment Certification*. Technical Report RTCA/DO-178C, RTCA Inc, 1150 18^{th} Street, NW, Suite 910, Washington, D.C. 20036-3816 USA.

[31] RTCA SC-205/EUROCAE WG-71 (2011): *Software Tool Qualification Considerations*. Technical Report RTCA/DO-330, RTCA Inc, 1150 18^{th} Street, NW, Suite 910, Washington, D.C. 20036-3816 USA.