@inproceedings{Briand2016,

author = {L. Briand and S. Nejati and M. Sabetzadeh and D. Bianculli},

title = {{Testing the Untestable: Model Testing of Complex Software-intensive Systems}},

booktitle = {Proceedings of the 38th International Conference on Software Engineering (ICSE) - Visions of 2025 and Beyond Track},

year = {2016},

pages = {789-792},

address = {Austin, TX, USA},

doi = {http://doi.acm.org/10.1145/2889160.2889212},

publisher = {ACM},

summary = {\citet{Briand2016} (P0003) presented challenges and potential solutions to the difficulty of testing ``untestable'' systems, that is, systems that continuously interact with the environment (including software and hardware). They suggested \emph{model testing}, which addresses executable models, including both behavior and properties, and its environment, allowing testers to completely automate the execution of many tests. Their concept combined model checking with model-based testing in a way that reduced problems with scalability. They proposed evolutionary algorithms to generate tests for critical and risky scenarios at the model level, then run on the deployed system. This speculative study did not provide details on how this approach would work and did not address model-to-code traceability matters.}

}

@inproceedings{Camus2016,

author = {J.-L. Camus and C. Haudebourg and M. Schlickling and J. Barrho},

title = {{Data Flow Model Coverage Analysis: Principles and Practice}},

booktitle = {Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS)},

year = {2016},

pages = {1-10},

address = {Toulouse, France},

doi = {https://hal.archives-ouvertes.fr/hal-01262411/},

publisher = {Centre pour la Communication Scientifique Directe},

summary = {\citet{Camus2016} (P0004) proposed a way to apply data flow coverage to support requirements-based testing as required by the DO-178C/DO-331 standard~\cite{DO-178C, DO-331}. Models were described using the SCADEdata flow language. They employed their Model Test Coverage (MTC) tool to run tests and collect model coverage data. They also analyzed structural code coverage analysis on the code, using the DS and MCDC criteria. In the study, the tool handled complex software with 3,000 user operators, 100,000 model coverage points, and 50,000 code coverage points.}

}

@inproceedings{Eriksson2016,

author = {A. Eriksson and B. Lindström},

title = {{UML Associations: Reducing the Gap in Test Coverage Between Model and Code}},

booktitle = {Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)},

year = {2016},

pages = {589-599},

address = {Rome, Italy},

doi = {http://ieeexplore.ieee.org/document/7954409/},

publisher = {IEEE},

summary = {\citet{Eriksson2016} (P0005) continued the work of \citet{Eriksson2012} by generating logic-based tests at the platform-independent level using xtUML. Although many tools exist to measure test coverage at the code level, the authors argue that coverage tools are not available at the model level. Thus, to measure test coverage, the model first has to be transformed to code, which takes time and distracts the engineer from the model level. Thus, \citet{Eriksson2016} presented an innovative way to measure coverage at the model level. Since code introduces predicates that are not in the model, they first created model-level predicates that capture the predicates that would appear during transformation. This allows test coverage to be measured at the model level. Based on one of the six applications addressed in their previous study \cite{Eriksson2012}, they demonstrated that coverage measured at the model level can accurately predict coverage at the code level. This is particularly important for logic-based testing, since coverage at the code level is often required.}

}

@article{Shokry2009,

author = {H. Shokry and M. Hinchey},

title = {{Model-Based Verification of Embedded Software}},

journal = {IEEE Computer},

year = {2009},

volume = {42},

number = {2},

pages = {53-59},

doi = {https://doi.org/10.1109/MC.2009.125},

publisher = {IEEE},

summary = {When verifying models, simulations are often run until the software engineers are satisfied, then code is generated, tested, and deployed. \citet{Shokry2009} (P0010) suggested that automatic code generation from models could lead to completely reliable software. They questioned whether tests designed for models could satisfy stringent code-level criteria such as MCDC on complex auto-generated code. They suggested solving the problem by decomposing the model into smaller units and by developing code generators that produce simple code. A preliminary study found that randomly generated model level tests only covered about 32\% of the statements in the code.}

}

@article{Matinnejad2015,

author = {R. Matinnejad and S. Nejati and L. Briand and T. Bruckmann and C. Poull},

title = {{Search-based Automated Testing of Continuous Controllers: Framework, Tool Support, and Case Studies}},

journal = {Information and Software Technology},

year = {2015},

volume = {57},

pages = {705-722},

doi = {https://doi.org/10.1016/j.infsof.2014.05.007},

publisher = {Elsevier},

summary = {\citet{Matinnejad2015} (P0011) addressed model-level test generation for embedded software, specifically, continuous controllers of automotive systems. They framed their work as an \textit{X-in-the-loop} process, where \textit{X} can be $M$odel (MIL), $H$ardware (HIL), $S$oftware (SIL), or $P$rocessor (PIL). Model-in-the-loop (MIL) testing generates tests to evaluate the model and to support SIL, PIL, and HIL tests. \citet{Matinnejad2015} applied search-based techniques to identify worst-case scenarios that are derived from non-functional requirements (liveness, stability, smoothness, and responsiveness) to generate stress tests that violate the requirements. They fed Simulink models to a tool they developed, Continuous Controller Tester (CoCoTest). They found that such tests could reveal faults that tests created by domain experts might not find, and that their tests revealed more faults with 27\% fewer tests that randomly generated tests.}

}

@inproceedings{Eriksson2013,

author = {A. Eriksson and B. Lindström and J. Offutt},

title = {{Transformation Rules for Platform Independent Testing: An Empirical Study}},

booktitle = {Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST)},

year = {2013},

pages = {202-211},

address = {Luxembourg City, Luxembourg},

doi = {https://doi.org/10.1109/ICST.2013.28},

publisher = {IEEE},

summary = {\citet{Eriksson2013} (P0045) further explored the indications that 100\% coverage at the model level translates to less than 100\% coverage at the code level \cite{Pretschner2005, Amalfitano2015}, especially when logic criteria are applied \cite{Eriksson2012}. This effect is largely due to implicit semantic behaviors in the model, that is, conditional behaviors that are not explicitly expressed as predicates in the model. To address this problem \citet{Eriksson2013} developed model to model transformation rules to turn implicit predicates into explicit predicates in the model. These rules were applied to xtUML, allowing the new conditions to become part of the test requirements. These transformations resulted in tests developed at the model level to achieve 100\% coverage at the code level, an important requirement of FAA standards \cite{DO-178C}.}

}

@article{Kirner2009,

author = {R. Kirner},

title = {{Towards Preserving Model Coverage and Structural Code Coverage}},

journal = {EURASIP Journal on Embedded Systems},

year = {2009},

volume = {2009},

pages = {1-16},

doi = {https://link.springer.com/article/10.1155/2009/127945},

publisher = {Springer},

summary = {\citet{Kirner2009} (P0056) addressed the problem of preserving structural code coverage after automatic code generators and compilers apply transformations. These transformation include instruction reordering, copying, and moving. The key idea is that program properties must be maintained when program $P\_1$ is transformed into program $P\_2$, so that the structural coverage on $P\_1$ is preserved in $P\_2$ with the same test data. This can apply to both model-to-text (M2T) and text-to-text (T2T) transformations. The author defined formal rules based on coverage criteria (statement coverage, decision coverage, and MCDC), a set of coverage preservation rules, and a set of code optimizations. A coverage profile is created and integrated into a code transformer. This study was theoretical and did not present a tool or experimental results. Several examples were shown, but effectiveness and cost were not addressed.}

}

@inproceedings{Eriksson2012,

author = {A. Eriksson and B. Lindström and S. Andler and J. Offutt},

title = {{Model Transformation Impact on Test Artifacts: An Empirical Study}},

booktitle = {Proceedings of the 9th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)},

year = {2012},

pages = {5-10},

address = {Innsbruck, Austria},

doi = {http://doi.acm.org/10.1145/2427376.2427378},

publisher = {ACM},

summary = {\citet{Eriksson2012} (P0059) quantitatively evaluated the impact on the number of test requirements when design models specified in executable and translatable UML (xtUML) are transformed into C++ code. They analyzed the predicates and clauses (\emph{test artifacts}) that appeared in both model and code. They mapped C++ code to model elements to establish traceability. An experiment with six avionics applications showed that the code had 67\% more test artifacts than the model did. The additional test artifacts represented implicit decisions in the design and were easily predictable. In the experiment, models were created using the BridgePoint UML modeling environment, and C++ code was analyzed by using the LLVM compiler framework. Tailor-made plugins were developed for both tools (BridgePoint and LLVM) to count the number of predicates and clauses.}

}

@inproceedings{Sturmer2005,

author = {I. Stürmer and D. Weinberg and M. Conrad},

title = {{Overview of Existing Safeguarding Techniques for Automatically Generated Code}},

booktitle = {Proceedings of the 2nd International Workshop on Software Engineering for Automotive Systems},

year = {2005},

pages = {1-6},

address = {St. Louis, MO, USA},

doi = {http://doi.acm.org/10.1145/1082983.1083192},

publisher = {ACM},

summary = {}

}

@article{Tekcan2012,

author = {T. Tekcan and V. Zlokolica and V. Pekovic and N. Teslic and M. Gündüzalp},

title = {{User-driven Automatic Test-case Generation for DTV/STB Reliable Functional Verification}},

journal = {IEEE Transactions on Consumer Electronics},

year = {2012},

volume = {58},

number = {2},

pages = {587-595},

doi = {https://doi.org/10.1109/TCE.2012.6227464},

publisher = {IEEE},

summary = {\citet{Tekcan2012} (P0071) presented a user-driven approach to generate test cases to test Digital TV receivers (DTV) and Set-Top Boxes (SBT). DTVs and SBTs are complex systems that deal with increasing amounts of information and are frequently updated. \citet{Tekcan2012} generated abstract XML test scripts from state transition models, then automatically translated them to Python test scripts. The abstract tests were created from usage profiles of end user scenarios and DTV design specifications. The XML test scripts were written in MaTeLo. To validate the approach, the authors created two test sets with 200 test cases apiece, one auto-generated and another manually created. The auto-generated tests achieved 85\% coverage and the manual tests achieved 50\% coverage. The study did not define what type of coverage was achieved.}

}

@inproceedings{Li2011,

author = {G. Li and R. Zhou and R. Li and W. He and G. Lv and T. J. Koo},

title = {{A Case Study on SDF-based Code Generation for ECU Software Development}},

booktitle = {Proceedings of the 3rd International Workshop on Component-Based Design of Resource-Constrained Systems (CORCS)},

year = {2011},

pages = {211-217},

address = {Munich, Germany},

doi = {https://doi.org/10.1109/COMPSACW.2011.45},

publisher = {IEEE},

summary = {\citet{Li2011} (P0086) reported on a case study that used model-based design to develop an Electronic Control Unit (ECU) for an automatic vehicle climate controller. The controller was modeled in Simulink and Synchronous Data-Flow (SDF) and then processed by a chain of tools to automatically generate code. The Simulink model represents the system as a set of interacting components, and the SDF model describes computation through the interactions. Both models are processed by a tool called MoDAL to produce a file that is then used by Ptolemy II to generate C code. The authors presented only results that are visually the same (based on charts derived from model-in-the-loop and software-in-the-loop simulations), but does not give details of the results.}

}

@inproceedings{Li2015,

author = {N. Li and J. Offutt},

title = {{A Test Automation Language Framework for Behavioral Models}},

booktitle = {Proceedings of the 11th Workshop on Advances in Model Based Testing (A-MOST)},

year = {2015},

pages = {1-10},

address = {Graz, Austria},

doi = {https://doi.org/10.1109/ICSTW.2015.7107402},

publisher = {IEEE},

summary = {presented a test framework, Structured Test Automation Language framEwork (STALE), to automatically transform abstract tests on a model to concrete tests on code. Concrete tests often share test components, which wind up being created many times in a manual test generation process. This creates duplication of effort (human cost), as well as errors. As opposed to other studies summarized in this paper, STALE works with a non-executable model, UML state machines, and models are translated to code by hand. STALE translates state machines into generic graphs, then identifies test paths through the graphs to satisfy graph coverage criteria (node, edge, and edge-pair coverage). The tester defines test components by describing mappings from abstract elements at the model level to concrete elements at the code level according to the novel Structured Test Automation Language (STAL). STALE uses the state machine and the STAL mappings to generate concrete tests in JUnit. The mappings provide full traceability from model to code. Experimental results showed 70\% reduction in effort for translating abstract tests into concrete tests and 100\% correctness in the translation process.}

}

@inproceedings{Baresel2003,

author = {A. Baresel and M. Conrad and S. Sadeghipour and J. Wegener},

title = {{The Interplay between Model Coverage and Code Coverage}},

booktitle = {Proceedings of the 10th EuroSTAR Software Testing Conference},

year = {2003},

pages = {1-14},

address = {Amsterdam, The Netherlands},

doi = {},

publisher = {Qualtech Group},

summary = {\citet{Baresel2003} (P0223) explored the relationship between requirements and structural coverage at the model and code levels. They generated model-level tests by applying the Stateflow tool within a Simulink model, then augmented the tests with the support of a search-based algorithm to be able to test the auto-generated C code. They reported coverage results for models and the code for three functional modules of an automotive system. The tests were able to reach internal system states, resulting in automatic generation of test suites with high coverage for model (ranging from 54\% to 100\%) as well as code (58\% to 100\%).}

}

@article{Mohalik2014,

author = {S. Mohalik and A. A. Gadkari and A. Yeolekar and K. C. Shashidhar and S. Ramesh},

title = {{Automatic Test Case Generation from Simulink/Stateflow Models using Model Checking}},

journal = {Software Testing, Verification and Reliability},

year = {2014},

volume = {24},

number = {2},

pages = {155-180},

doi = {https://doi.org/10.1002/stvr.1489},

publisher = {Wiley},

summary = {\citet{Mohalik2014}~(P0158) described a tool, AutoMOTGen, that generates model-level tests using the Stateflow tool within a Simulink model. AutoMOTGen translates Stateflow models to the SAL language, which underlies the generation of tests. Test coverage requirements are encoded as goals in SAL to establish traceability, and a model-checking engine is utilized to generate tests from counter-example traces. AutoMOTGen generates tests to satisfy block coverage, condition coverage, decision coverage, and MCDC. The study compared AutoMOTGen's tests with tests from a commercial tool that generates tests from inputs and guided simulation-based techniques, Reactis, on industrial software. The authors concluded that AutoMOTGen and Reactis produced complementary tests in the sense that the test suites from these tools can be combined to obtain better coverage. }

}

@inproceedings{Abade2015,

author = {A. Abade and F. Ferrari and D. Lucrédio},

title = {{Testing M2T Transformations: A Systematic Literature Review}},

booktitle = {Proceedings of the 17th International Conference on Enterprise Information Systems (ICEIS)},

year = {2015},

pages = {177-187},

address = {Barcelona, Spain},

doi = {https://doi.org/10.5220/0005378501770187},

publisher = {SCITEPRESS Digital Library},

summary = {}

}

@article{Sturmer2007,

author = {I. Stürmer and M. Conrad and H. Dörr and P. Pepper},

title = {{Systematic Testing of Model-Based Code Generators}},

journal = {IEEE Transactions on Software Engineering},

year = {2007},

volume = {33},

number = {9},

pages = {662-634},

doi = {https://doi.org/10.1109/TSE.2007.70708},

publisher = {IEEE},

summary = {\citet{Sturmer2007} (P0253) test code generators whose inputs and outputs are executable models. Test cases comprise a test model in Simulink and input values are called test vectors. Input values are used to check the functional equivalence between the model under test and the auto-generated C code. This approach assumes that both the inputs and outputs are executable. \citetauthor{Sturmer2007} built a tool to map model elements to code so tests can be executed in both artifacts, allowing for optimizations during code generation, as long as the optimizations are clearly specified. This allows the model elements to be traced to code, including changes by the optimizer. The study does not give technical details on how Simulink and Stateflow models are turned into code, or how test vectors are translated into code.}

}

@article{Conrad2009,

author = {M. Conrad},

title = {{Testing-based Translation Validation of Generated Code in the Context of IEC 61508}},

journal = {Formal Methods in System Design},

year = {2009},

volume = {35},

number = {3},

pages = {389-401},

doi = {https://doi.org/10.1007/s10703-009-0082-0},

publisher = {Springer},

summary = {\citet{Conrad2009} (P0259) described a process to run tests to demonstrate behavioral equivalence between model and code. Model-level testing was called \textit{Model-in-the-Loop} and code-level testing was called \textit{Processor-in-the-Loop}. Tests were run at both levels and compared with a tolerance to allow for unimportant differences. Tests were generated to cover model elements according to a particular coverage criterion. The workflow also includes traceability review to ensure unintended functionality was not added to the code. If code coverage was lower than model coverage, or if code elements cannot be traced back to the model, then the translation may have introduced unintended functionality. The study did not include empirical results.}

}

@inproceedings{Pretschner2005,

author = {A. Pretschner and W. Prenninger and S. Wagner and C. Kühnel and M. Baumgartner and B. Sostawa and R. Zölch and T. Stauner},

title = {{One Evaluation of Model-based Testing and Its Automation}},

booktitle = {Proceedings of the 27th International Conference on Software Engineering (ICSE)},

year = {2005},

pages = {392-401},

address = {St. Louis, MO, USA},

doi = {https://doi.org/10.1145/1062455.1062529},

publisher = {ACM},

summary = {\citet{Pretschner2005}~(P0313) investigated the effectiveness of model-based testing and automation by evaluating test sets in terms of fault detection, model coverage, and code coverage. Some of their tests were generated automatically, some from models and some from code, some randomly, some with functional testing criteria, and some by hand. They found that tests generated from models found more faults, especially faults that resulted in changes to requirements.}

}

@article{Conrad2005,

author = {M. Conrad and S. Sadeghipour and H-W. Wiesbrock},

title = {{Automatic Evaluation of ECU Software Tests}},

journal = {SAE Transactions},

year = {2005},

volume = {114},

pages = {583-592},

address = {Warrendale, PA, USA},

doi = {https://www.jstor.org/stable/44682468},

publisher = {SAE International},

summary = {\citet{Conrad2005} (P0321) presented a testing technique to validate model-to-code translations. Tests are created to satisfy decision coverage on the model, then run on the code. This establishes behavioral equivalence between model and code. A subsequent step is a manual traceability review to ensure that all parts of the generated code can be traced back to the model. If the code coverage is lower than the model coverage, or if it is not possible to trace code elements back to the model, then the translation may have introduced unintended functionality.}

}

@inproceedings{Amalfitano2015,

author = {D. Amalfitano and V. {De Simone} and A. R. Fasolino and V. Riccio},

title = {{Comparing Model Coverage and Code Coverage in Model Driven Testing: An Exploratory Study}},

booktitle = {Proceedings of the 6th International Workshop on Testing Techniques for Event BasED Software (TESTBEDS)},

year = {2015},

pages = {70-73},

address = {Lincoln, NE, USA},

doi = {https://doi.org/10.1109/ASEW.2015.18},

publisher = {IEEE},

summary = {Model Driven Testing (MDT) generates model-level, or abstract, tests from models, which are then applied to code as concrete tests. Ideally, both the code and the concrete tests are generated automatically from the model-level artifacts. Since the tests are designed to achieve coverage on the model, an important question is whether the concrete tests achieve the same level of coverage on the code. \citet{Amalfitano2015}~(P0362) investigated this question by finding factors that create differences in coverage. They explored differences for four state machine models and eight test suites, and found significant differences. Tests that achieved 100\% edge and node coverage at the model level, ranged from 48\% to 75\% for statement coverage on the code and 25\% to 52\% for branch coverage.}

}

@article{Li\_a\_2016,

author = {N. Li and J. Offutt},

title = {{Test Oracle Strategies for Model-Based Testing}},

journal = {IEEE Transactions on Software Engineering},

year = {2016},

volume = {43},

number = {4},

pages = {372-395},

doi = {https://doi.org/10.1109/TSE.2016.2597136},

publisher = {IEEE},

summary = {\citet{Li\_a\_2016} (P0374) investigated the need for better test oracle strategies tailored to MBT. In automated tests, \textit{test} oracles determine whether tests pass by comparing expected with actual output, usually in the form of assertions. Since it is usually too expensive to check the entire output state of a program, test oracles tend to target certain output variables or behaviors. \citeauthor{Li\_a\_2016} defined 10 test oracle strategies (OSes), asking how many more failures are revealed by looking at more of the output state, and by looking at intermediate states more frequently. They found that OSes that checked states only after execution terminates, and that checked only state invariants from the model, revealed failures almost as often as OSes that checked states more frequently and that looked at the complete output state. The abstract tests were generated with STALE, and translated to concrete tests in the form of calls to methods in the software. Testers provided mappings from the model level to the code level, thus the OSes are only partially automated for traceability purposes.}

}

@inproceedings{Li\_b\_2016,

author = {N. Li and A. Escalona and T. Kamal},

title = {{Skyfire: Model-Based Testing with Cucumber}},

booktitle = {Proceedings of the 9th International Conference on Software Testing, Verification and Validation (ICST) - Testing Tool Papers},

year = {2016},

pages = {393-400},

address = {Chicago, IL, USA},

doi = {https://doi.org/10.1109/ICST.2016.41},

publisher = {IEEE},

summary = {\citet{Li\_b\_2016} (P0375) extended STALE to create skyfire, which reads behavioral diagrams (UML state machines), finds all transitions, and generates test cases to satisfy graph coverage criteria. Skyfire then creates Cucumber scenarios, which provides mappings to translate model-level tests to concrete tests.}

}

@inproceedings{Lamancha2011,

author = {B. P. Lamancha and P. Reales and M. Polo and D. Caivano},

title = {{Model-driven Testing - Transformations from Test Models to Test Code}},

booktitle = {Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)},

year = {2011},

pages = {121-130},

address = {Beijing, China},

doi = {https://www.semanticscholar.org/paper/Model-driven-Testing-Transformations-from-Test-to-Lamancha-Mateo/43db2ba488c36ab03da1b719fc34c186095dcf1e},

publisher = {SCITEPRESS Digital Library},

summary = {\citet{Lamancha2011} (P0381) extended their prior framework to derive abstract tests from UML diagrams to generate concrete tests. The framework first does a model-to-model transformation from UML to UML Testing Profile models, then uses the MofSCript tool to transform the abstract tests to JUnit and NUnit test methods. The framework does not include any traceability support from model to code.}

}

@inproceedings{Fraternali2010,

author = {P. Fraternali and M. Tisi},

title = {{Multi-level Tests for Model Driven Web Applications}},

booktitle = {Proceedings of the 10th International Conference on Web Engineering (ICWE)},

year = {2010},

pages = {158-172},

address = {Vienna, Austria},

doi = {https://doi.org/10.1007/978-3-642-13911-6\_11},

publisher = {Springer},

summary = {When models change, tests that are derived from the models but applied to the code must also be changed. This is particularly problematic when test generation is manually done or when abstract tests are transformed to concrete tests by hand. \citet{Fraternali2010} (P0383) developed a multi-level test generation approach, and a transformation framework to align two streams of transformation, from computation independent models (CIM) to code, and from computation independent test (CIT) specifications to executable test scripts. The test scripts are updated by mappings that can be applied after changes take place.}

}

@inproceedings{Aniculaesei2019,

author = {A. Aniculaesei and A. Vorwald and A. Rausch},

title = {{Using the SCADE Toolchain to Generate Requirements-Based Test Cases for an Adaptive Cruise Control System}},

booktitle = {Proceedings of the 16th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)},

year = {2019},

pages = {503-513},

address = {Munich, Germany},

doi = {https://doi.org/10.1109/MODELS-C.2019.00079},

publisher = {IEEE},

summary = {\citet{Aniculaesei2019} (P0411) sought to explore model-checking for the automatic generation of test cases based on requirements test cruise control system for the automotive industry. The authors compared the fault revealing capability of test sets automatically generated with a commercial tool (the SCADE toolchain) and an academic, open-source tool (NuSMV) that applied the model-checking approach. Both tools turn models and test cases generated at the model level into C code, and the study assesses the effectiveness of the test suites based on their mutation scores.}

}

@inproceedings{Durak2018,

author = {U. Durak and D. Müller and F. Möcke and C. B. Koch},

title = {{Modeling and Simulation based Development of an Enhanced Ground Proximity Warning System for Multicore Targets}},

booktitle = {Proceedings of the 2018 International Symposium on Model-driven Approaches for Simulation Engineering (Mod4Sim)},

year = {2018},

pages = {1-12},

address = {Baltimore, MD, USA},

doi = {https://dl.acm.org/doi/10.5555/3213214.3213218},

publisher = {ACM},

summary = {\citet{Koch2018} (P0448) presented a tool named Scilab/Xcos XTG, which was designed to support the X-in-the-loop testing pipeline for model-based development of parallel real-time software that runs on multicore processor architectures tailored to the avionics industry, devised by \citet{Durak2018} (P0443). Scilab/Xcos XTG extends the Scilab/Xcos tool by creating \emph{toolboxes} that include scenario file templates, system interfaces, and model components. In this context, a toolbox enables back-to-back testing by injecting automatically generated code into the model elements, allowing enhanced simulations to be carried out at the model level. The tool also generates input test data and expected output that can be used to exercise the model and the code at various phases of the model-based testing workflow. However, similar to P0443, no empirical evidence supporting the effectiveness of the tool was provided: a single example was outlined.}

}

@inproceedings{Koch2018,

author = {C. B. Koch and U. Durak and D. Müller},

title = {{Simulation-based Verification for Parallelization of Model-based Applications}},

booktitle = {Proceedings of the 50th Computer Simulation Conference (SummerSim)},

year = {2018},

pages = {1-10},

address = {Bordeaux, France},

doi = {https://dl.acm.org/doi/10.5555/3275382.3275392},

publisher = {ACM},

summary = {\citet{Koch2018} (P0448) presented a tool named Scilab/Xcos XTG, which was designed to support the X-in-the-loop testing pipeline for model-based development of parallel real-time software that runs on multicore processor architectures tailored to the avionics industry, devised by \citet{Durak2018} (P0443). Scilab/Xcos XTG extends the Scilab/Xcos tool by creating \emph{toolboxes} that include scenario file templates, system interfaces, and model components. In this context, a toolbox enables back-to-back testing by injecting automatically generated code into the model elements, allowing enhanced simulations to be carried out at the model level. The tool also generates input test data and expected output that can be used to exercise the model and the code at various phases of the model-based testing workflow. However, similar to P0443, no empirical evidence supporting the effectiveness of the tool was provided: a single example was outlined.}

}

@article{Amalfitano2019,

author = {D. Amalfitano and V. {De Simone} and R. R. Maietta and S. Scala and A. R. Fasolino},

title = {{Using Tool Integration for Improving Traceability Management Testing Processes: An Automotive Industrial Experience}},

journal = {Software: Evolution and Process},

year = {2019},

volume = {31},

number = {6},

pages = {1-20},

doi = {https://doi.org/10.1002/smr.2171},

publisher = {Wiley},

summary = {\citet{Amalfitano2019} (P0451) reported on a relatively simple experiment to probe into the difference between model and code coverage for four different state machine models and eight test suites. According to their results, coverage was 100\% at the model level (by design); statement and branch coverage, however, were considerably different. The authors pointed out three possible reasons for such a difference: \emph{(i)} code was added by the code generator for exception handling and debugging purposes, which was not present in the model, \emph{(ii)} model coverage was not enough to guarantee coverage of code, and \emph{(iii)} the style of design of the system model influenced coverage. The authors did not elaborate on how to avoid problems \emph{(ii)} and \emph{(iii)}.}

}

@inproceedings{Vanhecke2019,

author = {J. Vanhecke and X. Devroey and G. Perrouin},

title = {{AbsCon: A Test Concretizer for Model-Based Testing}},

booktitle = {Proceedings of the 15 Workshop on Advances in Model Based Testing (A-MOST)},

year = {2019},

pages = {15-22},

address = {Xi'an, China},

doi = {https://doi.org/10.1109/ICSTW.2019.00027},

publisher = {IEEE},

summary = {\citet{Vanhecke2019} (P0463) described the AbsCon (Abstract test case Concretizer) tool, which generates executable test cases from abstract definitions. AbsCon utilizes mappings for the interface, actions, and assertions of the system under test. Rather than involving a model-based testing transformation chain, AbsCon bridges the gap from model to code in test case generation by using a combination of transformation and adaptation. Abstract test cases are initially defined in an XML file in which each test case is described as a sequence of actions and assertions regarding the system under test. Concrete test cases are generated as Python scripts that execute the verification steps and sequences of assertions.}

}

@article{Kalaee2019,

author = {A. Kalaee and V. Rafe},

title = {{Model-based Test Suite Generation for Graph Transformation System Using Model Simulation and Search-based Techniques}},

journal = {Information and Software Technology},

year = {2019},

volume = {108},

pages = {1-29},

doi = {https://doi.org/10.1016/j.infsof.2018.12.001},

publisher = {Elsevier},

summary = {\citet{Kalaee2019} (P0474) devised an approach that turns the problem of generating test suites into an optimization problem. In this context, the initial configuration of the system under test is a graph and transformation rules specify method and service invocations. Thus, each sequence of rule transitions in the state space is considered a test case, which at code level can be translated as a sequence of method invocations. The proposed approach is fed with randomly generated test cases that encode one finite path of state space, then search-based algorithms are used to satisfy the coverage criterion all def-uses. According to the authors, their study is the first research effort devoted to applying search algorithms to generate test suites from graph transformation specifications.}

}

@inproceedings{Veanes2008,

author = {M. Veanes and C. Campbell and W. Grieskamp and W. Schulte and N. Tillmann and L. Nachmanson},

title = {{Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer}},

booktitle = {Proceedings of the 2008 Formal Methods and Testing Workshop (FORTEST)},

year = {2008},

pages = {39-76},

address = {},

doi = {https://doi.org/10.1007/978-3-540-78917-8\_2},

publisher = {Springer},

summary = {\citet{Veanes2008} provide a in-depth discussion of Spec Explorer, which is a tool for testing reactive, object-oriented software systems developed by the Foundations of Software Engineering group in Microsoft Research. Spec Explorer provides an integrated tool environment that allows testers to develop, examine, and validate models. Additionally, Spec Explorer is able to generate tests from models and execute tests against a system under test. In the context of Spec Explorer, the system's behavior is described by models written in the language Spec# (an extension of C#) and AsmL. Fundamentally, a model in Spec# defines the state variables and update rules of an abstract state machine. Spec Explorer employs algorithms similar to those of explicit state model checkers to explore the machine's states and transitions, which results in a finite graph containing a subset of model states and transitions. Spec Explorer also generates test cases for the explored behavior, which can be run against the system under test in an offline (i.e., when test generation and execution are seen as two independent phases) or online (i.e., in which the results from test execution are used to prune the generation process) fashion. The resulting test cases can be saved either as C# or Visual Basic (VB) programs.

}

}

@article{Drave2019,

author = {I. Drave and S. Hillemacher and T. Greifenberg and S. Kriebel and E. Kusmenko and M. Markthaler and P. Orth and K. S. Salman and J. Richenhagen and B. Rumpe and C. Schulze and M. von Wenckstern and A. Wortmann},

title = {{SMArDT Modeling for Automotive Software Testing}},

journal = {Software: Practice and Experience},

year = {2019},

volume = {49},

number = {2},

pages = {301-328},

doi = {https://doi.org/10.1002/spe.2650},

publisher = {Wiley},

summary = {\citet{Drave2019} developed a method to manage requirements, design, and test in automotive industry. The specification method for requirements, design, and test (SMArDT) leverages model-based software engineering techniques with the aim of mitigating the deficiencies of the established V-Model. The method is based on the premise that consistency checking between layers and test case generation (and regeneration) helps developers and testers cope with the bureaucracy imposed by the classical V-Model. As a result, the method emphasizes the technical aspects of the models in the different layers in the V-Model in hopes of ensuring traceable, verifiable, and testable artifacts throughout the development process. Moreover, the authors posit that consistency among specification artifacts between layers enables automatic transformation of tests to lower levels. To realize the method in a modeling environment agnostic fashion, the authors put together a configurable tool chain that can turn functional requirements modeled using activity diagrams, state charts, sequence diagrams, and internal block diagrams from various formats into executable test cases for various output formats. The authors carried out a case study to compare model-based tests derived in the context of the proposed approach and manually created tests. According to the results from such a case study, the proposed MBT approach generated tests that detected more defects than the traditional hand-crafted tests. The MBT approach was especially effective at generating test cases that uncover defects caused by inconsistent requirements. Nevertheless, neither the traditional nor the model-based tests uncovered all defects. }

}

@inproceedings{Drave2018,

author = {I. Drave and S. Hillemacher and T. Greifenberg and B. Rumpe and A. Wortmann and M. Markthaler and S. Kriebel},

title = {{Model-Based Testing of Software-Based System Functions}},

booktitle = {Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)},

year = {2018},

pages = {146-153},

address = {},

doi = {https://doi.org/10.1109/SEAA.2018.00032},

publisher = {IEEE},

summary = {}

}

@inproceedings{Markthaler2018,

author = {M. Markthaler and S. Kriebel and K. S. Salman and T. Greifenberg and S. Hillemacher and B. Rumpe and C. Schulze and A. Wortmann and P. Orth and J. Richenhagen},

title = {{Improving Model-Based Testing in Automotive Software Engineering}},

booktitle = {Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)},

year = {2018},

pages = {172-180},

address = {},

doi = {https://doi.org/10.1145/3183519.3183533},

publisher = {ACM},

summary = {}

}