Model-based testing of simulation models

A case-study approach to test model design for automated testing of discrete-event simulation models

F.G.A. van Bergeijk

TU Delft

Model-based testing of simulation models

A case-study approach to test model design for automated testing of discrete-event simulation models

by

F.G.A. van Bergeijk

in partial fulfillment of the requirements
for the degree of Master of Science
at the Delft University of Technology,
in Engineering and Policy Analysis,

to be defended publicly on Monday February 5, 2024

Student number: 5198518
Project duration: March 3, 2023 – February 5, 2023
Thesis committee:

Prof. dr. ir. A. Verbraeck,	TU Delft, first supervisor
Dr. Y. Huang,	TU Delft, second supervisor
Ir. T. van den Berg,	TNO, external supervisor
Ir. O. Quispel,	TU Delft, advisor

TU Delft

Acknowledgments

Dear reader,

It has been a learning experience to complete this master's degree and to work on this thesis project. After my bachelor's degree in Mechanical Engineering, the Sustainable Energy Technology and Engineering & Policy Analysis programs have broadened my horizons. I have appreciated the unique combination of analysis of societal problems, and decision-making, while also delving into technical details and advanced methods. The following 70 pages can be seen as a great example of this last aspect.

I would like to thank to the committee for their constructive feedback and precision. I would like to give special thanks to Omar and Alexander for their involvement, enthusiasm, and flexibility in this project. It has been a unique experience to have discussions in such detail on the methods that I learned during my studies.

I would also like to thank the students and teachers of EPA for keeping everyone involved during the lockdowns, online classes, and between Delft and The Hague. That is not a given for any faculty! And lastly, I would like to thank my parents for their constant support during my journey as a student through all the places that a technical university has to offer.

Ferd van Bergeijk

Eindhoven, January 2024

Contents

Executive summary

In this thesis project, model-based testing (MBT) is applied for the verification of simple simulation models used in modeling & simulation for decision-making. MBT is an approach to testing where test suites are generated by a test model, usually with the aim of automating most parts of the testing process. While MBT necessitates additional effort to design test models, applying MBT can lead to more efficient tests compared to manual approaches, as many test cases can be generated at once. An additional efficiency benefit is improved adaptability because large test suites can be changed by only updating the test model(s) used to generate them. Furthermore, test models could be reused for systems under test (SUTs) that have similar functional requirements. Literature also indicates that MBT can improve the quality and effectiveness of tests.

It is found relevant to research MBT of simulation models, because some proposed benefits of MBT may be especially relevant for simulation models. Firstly, MBT is often used to generate many test cases, while verification often needs analysis of many simulation runs. Secondly, MBT can be done reactive, meaning that the generation of test cases is influenced during test execution by the (internal) states of the SUT. This is relevant for simulation models as SUTs, because of the unpredictable and stochastic nature of simulation models.

Literature review shows that not many publications exist where MBT is considered for testing simulation models specifically. Only some case studies are known where MBT techniques are used with a simulation model as the SUT, with varying levels of automation. Some literature on MBT for systems similar to simulation models does give general guidelines for specific choices for test design, especially on test execution. However, no general guidelines are known on a fundamental issue for MBT for simulation models specifically: the specification of the test model(s). The design of test models is therefore the focus of this project. The goal is to demonstrate the feasibility and to develop a methodology. An assessment of the efficiency and effectiveness of MBT for simulation models is not within the scope.

A case study approach is used to explore how test models can be designed. Three test packages are made where three different simple simulation models are used as SUT. An existing MBT software tool, AltWalker, is selected for this purpose. It is shown that AltWalker can be used to test reactive systems, but that it is not practical for testing long simulation runs.

In the process, it is found that MBT is useful for step-by-step verification of the dynamic behavior of simulation models. Test models are developed that are aware of the SUT's current states, and that can generate its expected behavior and select relevant tests for each time advancement of the simulation model that is used as SUT. This allows failures of the SUT to be detected as they occur, along with information on the SUT's state during a failure, and which state was expected. This can allow a modeler to better understand when, where, and possibly why faults occur.

The research is limited in that the test models only advance time in the SUTs and do not give other types of inputs to the SUTs. An alternative approach where test models give specific inputs to simulation models has been found in literature and is recommended for further research. Such an approach can lead to more efficient tests where the test model triggers specific events in the SUT that are suspected to cause failures. With this approach, test models can also be used to simulate the environment that a SUT normally interacts with. Such environment test models have been explored in earlier literature, but not in this project. This last approach is useful for more effective component testing and integration testing of simulation components that are part of a simulation environment.

In addition, the case studies show how dynamic verification of simulation models using MBT techniques can be combined with verification of the results from multiple simulation runs, in one test package. This is deemed essential to reach a complete verdict on the correct functioning of a simulation model given its requirements, especially because the influence of the input parameter space and the simulation model's stochasticity should be taken into account. It is found that automated test packages can be made that combine both purposes, but that separate test suites should be generated for dynamic verification and verification of results to prevent computationally expensive tests.

It is further demonstrated how black-box tests could be used to perform reactive tests with minimal communication between the SUT's interface and the test model. This is recommended for further research, as it may results in a modular approach where test models can be reused for SUTs that have similar requirements.

The project shows the feasibility of MBT of simulation models and shows different options for test model design. It is unfortunately not shown how well the developed test packages can detect faults in simulation models by using them on mutant SUTs. That is recommended for further research.

Glossary

Abstract (test) model A test model or part of a test model that is used to generate test paths that cannot be executed on the system under test directly.

Behavior (test) model A test model that is specifically used to generate the expected behavior of the system under test. Also called 'SUT models' in some sources.

Environment (test) model A test model that is specifically used to generate inputs to the system under tests. This is also called an input model in some literature.

Extended finite state machine (EFSM) An extension of finite state machines that adds internal variables, actions that can assign these internal variables, and trigger conditions with logic called guards.

Finite state machine (FSM) A transition-based model that consists of states (including an initial state), state transitions, and trigger inputs for these transitions.

Model-based testing (MBT) Techniques for testing of (software) systems where test models are used to make test suites, in order to automate some parts of the test.

MBT software tool An existing software tool for MBT that provides the means to develop model-based test packages. These tools can use test models written in a specific notation to generate and execute tests.

Oracle A function used to assign a verdict to some part of a test. An oracle consists of an expected behavior, and a procedure for how this is compared with outcomes from the system under test. (Richardson et al., 1992)

Oracle checking Evaluation of an oracle during testing, in order to get a verdict on some part of the test. Term introduced by Utting and Legeard (2007).

Reactive systems System that give a response to external events which depends on their internal states (Richardson et al., 1992, p.105).

Reactive tests Online, model-based tests where the output of the system under test can influence the test path generation. Term introduced by Zander et al. (2011).

System under test (SUT) The system that is being tested. The system under test can be a component of a larger system.

Test case A sequence of inputs for the system under tests, and associated expected behavior, optionally with oracles (Utting & Legeard, 2007, p.407).

Test model A model that can generate inputs and/or expected behavior, possibly with oracles, in a model-based test. In this project, it refers to the combination of an abstract model and the mappings that make the abstract model executable on the SUT.

Test path An abstract test case that is generated using an abstract test model.

Test suite A set of test cases made for a specific test goal.

Test package A collection of programs that are developed to generate and execute test suites for a specific system under test. This term is only used in this project to distinguish it from the more general-purpose MBT software tools.

1 Introduction

This thesis project is about the application of model-based testing (MBT) for the verification of simulation models. Testing of simulation models is a crucial step to assess if a model works as intended and that it gives the expected results. Testing is often a time-consuming endeavor in which most processes are typically not automated or formalized. MBT is a method to do this automation and formalization of some steps of the testing process. This could increase the efficiency and effectiveness of the verification of simulation models.

Testing, or dynamic verification, of simulation models is known to be an especially challenging task because of some distinguishing properties that simulation models possess: they are reactive systems with a time axis, stochasticity, and many states. This thesis aims to address the research gap in the literature by specifically considering simulation models as the systems under test (SUTs) for MBT. While most MBT literature focuses on testing software systems in general or (real-time) embedded systems, only a few sources specifically address MBT of simulation models.

Only some case studies have applied MBT for testing simulation models before, but they do not provide details on their test design processes, nor do they draw conclusions on how test design should be done. Therefore, this thesis project will use case studies to demonstrate some considerations of MBT for dynamic verification of simulation models. In this process, the usefulness of a specific MBT software tool will also be evaluated.

In this introduction, the relevance of testing for simulation models is first described in Section 1.1. Then, the problem statement of this thesis project is given first in Section 1.2 gives the problem statement and briefly states the research gap. Section 1.3 gives the research objectives and the associated research methods and questions. A short overview of the report is given here as well.

1.1. Relevance of testing for simulation models

Modeling & simulation for decision-making (M&S) is an important research field for all domains of engineering. Simulation models are used to better understand systems for which experiments cannot be conducted, or are too expensive to conduct (Sokolowski & Banks, 2010, p.3). This project focuses on components of stochastic discrete-event simulation models, which are used widely in domains such as supply chain management and logistics (Ullrich & Lückerath, 2017), manufacturing, defense operations, finance, and many more (Fu & Gross, 2013).

Properties of simulation models

Verification of simulation models is difficult for multiple reasons. The properties of simulation models make the definition of functional requirements for their behavior and results difficult, and they make it difficult to assess whether a simulation model meets these requirements. Simulation models may consist of many states and interacting processes that can change over time, which is already hard to analyze. Stochasticity is added to simulation models to represent the randomness and uncertainties of the systems that they represent (Mihram, 1972). This stochasticity means for verification that further analysis using statistical techniques is needed to draw conclusions (Sokolowski & Banks, 2010, p.21). Furthermore, simulation models are reactive, meaning that their next generated events or response

to given inputs depends on the current state. Likewise, the end results of a simulation run may be highly dependent on its initial conditions. In addition, the problem of testing is even more pronounced for large and complex simulation environments where synchronization problems and interoperability of components are additional issues that can be tested.

Importance of efficient and effective testing

Testing means dynamic verification of a simulation model: the model is executed during a test so that all potential faults can become apparent. Extensive testing is needed for simulation models because of their aforementioned properties. Many test cases must be run at different input parameters, in order to make a verdict that is statistically significant (Sokolowski & Banks, 2010).

Testing and debugging add extra costs to the development process of simulation models. Testing should be done continuously throughout the development cycle of a simulation model. This prevents problems from being found too late, which would lead to extra costs (Huq, 2000). Techniques that could lead to more efficient tests, such as MBT, must therefore be considered for simulation models.

Furthermore, a modular approach to testing itself is advised for large and complex simulation environments, as verification of an entire simulation environment in one go is not feasible. By partitioning the simulation model into modules that can be (re)composed, verification and validation can be made more achievable (Sokolowski & Banks, 2010, p.403). MBT is an adaptable technique that may help for such a modular approach, as is explained later.

1.2. Problem statement

The problem addressed in this thesis project is now explained. It is first defined what MBT is. The supposed advantages and disadvantages of MBT are summarized from literature, along with some MBT options that may be relevant for testing simulation models specifically. Furthermore, two relevant problems for testing are discussed: a distinction between dynamic behavior and results, and between black-box and white-box testing. Lastly, the research gap is formulated.

The term ‘MBT’ can refer to several testing approaches. A common feature is that the generation and/or selection of test cases is done by using one or multiple models, which are called ‘test models’ (Zander et al., 2011). Some authors further require that the expected behavior of the SUT is generated by this model (Pretschner et al., 2005; Utting & Legeard, 2007). The degree of automation of the testing process differs between MBT approaches. Utting and Legeard (2007) define MBT as the testing approach that has the highest level of automation, where not only the execution of tests but also the generation of test cases is automated, to some extent.

1.2.1. Advantages and disadvantages of MBT

The potential advantages of MBT are often explained in literature by contrasting them with the difficulties and shortcomings of manual testing. A complete overview of MBT including its advantages and disadvantages is given by Utting and Legeard (2007); these are now summarized.

Strengths of MBT in general

An important disadvantage of manual testing is that it can often be laborious, inefficient, and costly because it is not automated (Tretmans, 2008). This is especially relevant for tests of complex or reactive SUTs which necessitate a large number of test cases. MBT can be used to make the generation of these test cases more efficient by automation (Zander et al., 2011).

Another disadvantage of a manual test approach is that it may not guarantee that all functionality of the SUT is covered by the test cases. This is due to the unstructured nature of manual test designs. A test design is often made as a document that describes what should be tested and how that should be done. This is often done by a single engineer, and it may not be based on a rational or complete specification. For these reasons, the design process of manual tests can be called “unstructured, not reproducible, and not documented” (Utting et al., 2012, p.297). MBT alternatively uses formalized test models that can give a more clear and explicit representation of what is being tested, and how this should be done. Tretmans (2008) further states that the quality of tests can be improved by using models. A more formalized test design has the additional benefit that it can create a common understanding among engineers of the SUT’s requirements (Pretschner et al., 2005), and of the SUT implementation and the testing process itself.

An additional advantage of MBT is that a test model or even a test suite could be reused for testing multiple SUTs or similar components within a SUT. Such reusable, modular tests can be achieved with MBT when the test model is made abstract, meaning that it is only based on the functional requirements of the SUT. The test cases generated from such abstract test models are themselves only abstract. These can later be made executable on the SUT by mapping abstract steps to low-level instructions and the expected behavior of the SUT. An abstract model could thus easily be reused for testing an updated SUT. Only the mapping of instructions would have to be changed to achieve this (Utting & Legeard, 2007, p.29). Similarly, MBT could be used to make a test suite implementation-agnostic: an abstract test model could be used to test different SUTs that have similar functional requirements. And lastly, one abstract model could be used to test similar components of a SUT.

A similar advantage of MBT is that test models are adaptable. The maintenance of a test suite has been found easier when abstract models are used (Pretschner & Philipps, 2005). It is expected that MBT test suites can more easily adapt to changes in the SUT or its requirements. Whereas manual test cases would have to be written again, in MBT ideally only a part of the abstract model(s) and mapping have to be changed to make a large number of updated test cases.

Strengths of MBT for testing simulation models

It is assumed in this project that some potential advantages of MBT are especially relevant for testing simulation models. A first advantage holds specifically for online, reactive tests. Online tests are tests in which test case generation and test execution on the SUT are done simultaneously. Reactive tests are tests where the test generation is influenced by the SUT's outputs. Relevant test sequences can then be selected automatically during testing. Zander et al. (2012) indicate that reactive testing can be more effective than manual testing for reactive SUTs because faults can be noticed at the moment that they happen. Online, reactive tests are mentioned as an option for MBT in most literature, but they are not often presented as a key advantage for MBT. It is hypothesized in this project that reactive testing could be especially advantageous for testing simulation models. Because of the stochastic and complex nature of these models, (manually) writing a test sequence before a simulation run will not enable this test sequence to adapt to unforeseen behavior of the SUT. Marques et al. (2014) show in their empirical review of the effectiveness of MBT for simulation models that MBT can be better suited to find defects in scenarios that were not foreseen in manual 'ad-hoc' tests.

A second point for MBT of simulation models is an assumption that the aforementioned drawbacks of manual testing are even more pronounced for simulation models. This is because simulation models of complex problems can be seen as black-box models: the models are too complex to completely understand the causalities that lead to certain outputs (Kleijnen, 1995). Many test cases should be made to verify the simulation model's behavior, for two reasons, Firstly, an understanding and verification of the causalities can be improved by running the models with different sets of input parameters. And secondly, simulation models have inherent stochasticity, which means that all model runs will give different outputs, even when the same input parameters are used. Therefore, even multiple test cases may be needed where the SUT uses the same input parameters. This makes manual testing even more time-consuming. Thus, a method to automatically test a sequence of simulation model runs is expected to increase the efficiency of the testing process.

1.2.2. Empirical evidence of the effectiveness of MBT

Empirical research exists on the effectiveness of MBT compared to manual testing. It is deduced from a case study comparison by Marques et al. (2014) that achieving time benefits and improved test quality are not guaranteed. It depends on the scale and complexity of the SUT, and on the specific test goals, whether MBT can make a test more efficient or effective. On the other hand, Binder et al. (2015) found in a survey among 45 organizations where MBT was introduced that only 13% found it ineffective.

A trade-off for test efficiency can be formulated based on the potential advantages of MBT compared against the known drawbacks of manual testing. Applying MBT can initially require more work and costs, but it may save costs in the long run. Utting and Legeard (2007) exemplify this trade-off with a hypothetical comparison where for testing a small software system a choice has to be made between MBT and manual testing. They argue that if the software is developed with iterative versions, then using MBT becomes more efficient for every new software version since many (manual) adaptations of the test suite do not have to be repeated. Schieferdecker and Hoffmann (2012) list the types of additional costs that can be expected when MBT is applied to large software systems: more qualified

personnel may be needed, existing testing methods must be changed, and new models and test suites must be developed and validated. Regarding improved effectiveness or quality of testing, Pretschner et al. (2005) show in a case study the MBT detects “significantly more” errors related to requirements compared to manual tests. There was no difference found for errors related to programming.

The efficiency and effectiveness will not be quantified in this thesis. Rather, the feasibility and methodology of MBT for testing simulation models are explored.

1.2.3. Dynamic behavior and simulation results

An additional problem for testing simulation models is on what scale and about which aspects the functional requirements are. Dynamic verification, the main goal of MBT, could mean different things for a simulation model. It is found in this report that MBT is often used for step-by-step verification of the dynamic behavior of a SUT. This means that for every input, it is assessed whether the correct response is given by the SUT. This would mean for a simulation run that one unexpected event, not prescribed by the requirements, will lead to a ’failed’ verdict of the test case. However, the functional requirements for simulation models often include more aspects than only the correct type of events. The (range of) values of variables generated during a simulation run, given the values of initial parameters, may be prescribed as well. Based on this principle, it can be imagined that a wrongly implemented simulation model would generate the correct sequence of events given its input, but would still give the wrong results. This could be because the distributions for random numbers are implemented incorrectly, or wrong assumptions are implemented in the model. Verification of the outputs against a known or expected solution is then helpful to further detect faults.

Even this verification of values can be done on different scales. An approach for verification by Mihram (1972) is used to explain this. Two categories for the analysis of simulation models are distinguished: it can focus on static effects by looking at outcomes at the end of a run, or it can focus on dynamic effects by considering the model’s behavior during a run. Mihram (1972) concludes that both categories are needed for the analysis of stochastic models.

This point is similar to the earlier mentioned theory of Kleijnen (1995) that complex simulation models can be seen as black-box models. Kleijnen (1995) explains that analysis of these models is difficult because their expected output is not clearly defined. Techniques like sensitivity analysis and uncertainty analysis can be used for this purpose. These are also techniques where only the final outputs and the initial (input) parameters of a simulation run are considered, instead of the dynamic behavior.

Therefore, complete coverage of a simulation model’s requirements will necessitate more than step-by-step testing only. It must be explored whether MBT techniques can be used for verification against known (analytical) solutions, or how this aspect can be integrated in automated test packages.

1.2.4. Black-box testing and white-box testing

Most literature on MBT states that it is purely aimed at black-box testing. It is found that ‘black-box testing’ can refer to two related concepts: testing of a black-box SUT, or test development based solely on (functional) requirements. Tretmans (2008) and Utting and Legeard (2007) primarily use the first definition: black-box tests are tests where the SUT is seen as a black box. The test suite can only interact with the interface of the SUT, i.e. its accessible inputs and outputs. The SUT’s internal variables cannot be accessed by the test package. The second definition is based on a common ground rule for MBT that the development of test models should only be done based on the requirements documentation, as is argued by Utting and Legeard (2007). They state that therefore MBT is black-box testing by definition. This concept can also be referred to as ‘requirement-based testing’. White-box testing then means in contrast that the test design is (partly) based on the actual SUT code implementation. The term grey-box testing is sometimes used for test design that is based on both requirements and implementation (Shafique & Labiche, 2010, 4). Some literature mentions that MBT has been used for white-box testing in recent years (Zander et al., 2011).

It is hypothesized that using white-box testing to some extent may be useful for developing test models for testing simulation models. Using some knowledge of the SUT implementation during test model development can help in designing tests that cover all code and functionality of the SUT. Similarly, accessing some internal variables of the SUT may be necessary during testing. Still, the usefulness of white-box testing may depend on the scale and goals of testing. This idea can be exemplified by Kleijnen (1995) who explains that for the complete verification of a simulation model, one may even need to verify the ready-made components of the simulation language that are used. An example given

is that the pseudo-random number generators may be faulty; without testing it can only be assumed that they are not.

An obvious middle ground is to do requirement-based test development in order to make a test model that has all the advantages of an abstract model, and then use mappings to low-level internal SUT variables to make the tests executable and comprehensive. The development step would then be 'black box', while the test generation and execution is more 'white box' since not only the original SUT interface is interacted with. No term has been found for this approach. Of course, it is also up to the modeler or tester which SUT variables are internal and which are not. Because of this vagueness in definitions, this thesis project must call attention to the distinction between requirement-based vs. implementation-based testing, and between testing of black-box SUTs vs. white-box SUTs. It can be explored whether requirement-based testing leads to different test model designs.

1.2.5. Research gap and scope

It is found from literature research that most MBT literature does not specifically focus on the testing of simulation models. The leading taxonomies for MBT, by Marinescu et al. (2015), Utting et al. (2012), and Zander et al. (2011), only describe the testing of general software systems or embedded systems. A concept from these taxonomies that is closest to a simulation model is a reactive system, i.e. a system whose response to an event depends on its state. Some guidelines are given for testing such reactive systems.

Only some case studies have been found where MBT is applied to testing simulation models. Hollmann et al. (2012) also states in a case study for MBT of DEVS simulation models that the application of MBT is limited for M&S, while it is often used for other software programs. The case studies from literature are lacking in some points. Most do not provide clear guidelines on how MBT can be applied, or on how test models can be designed. Furthermore, none describe the need to consider both dynamic and static outcomes, or results, for verification of simulation models. A distinction between requirement-based and implementation-based testing is not enforced as well. A case study approach from TNO (2021) does work towards a standard methodology for MBT of simulation models, by selecting MBT software tools and by presenting simple test designs in two case studies. This work has been used in this thesis as a starting point for selecting certain MBT options that may be relevant, and for defining requirements and preferences for the selection of an MBT software tool.

More research into this topic is clearly needed. An overview of relevant options for MBT and further exploration of test model design in case studies can give a better understanding of the feasibility and usefulness of MBT for testing simulation models. This can be a first step to use this technique for developing more efficient and effective automated test tools for simulation models.

As mentioned earlier, the focus is on testing discrete-event, stochastic simulation models that are used for analysis and experimentation. This is because these are relevant to M&S for decision-making, well-known from literature, and because they possess many of the properties that make verification difficult. Topics like real-time, distributed, or live simulation are not within the scope. As a first attempt to develop a methodology, the focus in the case studies is on simple components of simulation models only. Furthermore, (model-based) testing is commonly seen as a method for verification, not for validation (Utting & Legeard, 2007). Validation is therefore not a subject in this project. Only the requirements and implementation are compared; the modeled system and usefulness of results are not considered.

1.3. Research design

The objective of this project is to develop MBT suites for simulation models used in M&S. The development process should lead to relevant options, considerations, and difficulties for applying MBT to simulation models. The main focus is on developing test models and integrating these into tests that can accurately assess whether a simulation model has been implemented correctly. The technology needed for running model-based tests is taken from existing MBT software tools, so this aspect is not focused on. As stated earlier, the term 'MBT' can refer to test suites with different levels of automation. The aim of this project is to automate as much of the test suite as possible, to get comprehensive conclusions. It is not the aim to automate any part of the test development process itself. This leads to the main research questions of this project:

> How can model-based testing be applied for automated dynamic verification of simulation models?

The research design is made by considering what has to be developed for a MBT suite. Literature can first be used for this to get an overview of how the necessary programs and models can be made. Taxonomies for MBT are available that list options for all dimensions relevant to MBT: test goals, test model specification, test case generation, test execution, and test evaluation (Marinescu et al., 2015; Utting et al., 2012; Zander et al., 2011). It can be chosen based on the properties of simulation models, which options may be relevant for further exploration in this project. This leads to the first subquestion:

1. What known options for model-based testing are relevant for testing simulation models?

This is also relevant for the selection of the MBT software tool that will be used. Requirements and preferences for such tools can be based on the options taken from taxonomies.

The chosen options for MBT are then explored by using the chosen MBT software tool to make test suites. This is done using case studies: example simulation models of increasing complexity are used as SUTs. Different simulation model paradigms are used to see if certain paradigms necessitate different approaches to MBT. As mentioned before, the main focus is on developing test models. A traditional goal of MBT, step-by-step dynamic verification, is first considered. This gives the subquestion:

2. How can test models be designed for step-by-step verification of the dynamic behavior of a simulation run?

The topic of black-box approaches to testing must be explored further, because this relates to two important concepts discussed in existing MBT literature: specification-based testing and testing with the SUT as a black box, as argued in Section 1.2.3. While black-box tests are already developed to answer subquestions 2 and 3, more focus should be given to explore whether different test model designs are needed when to test model can only interact with a limited SUT's interface. This could help to make tests that are composable or specification-agnostic. The following subquestion is defined for this:

3. How can test models be designed for verification of simulation models, with minimal knowledge of the SUT implementation and/or its internal variables?

Lastly, verification of a simulation model's results against known values is important as well, as is argued in Section 1.2.3. It is therefore explored how this aspect of verification can be integrated into an automated test suite. Ideally, the test suite should be able to run model-based tests of multiple simulation runs with different input parameters automatically. This is researched again by a case study with a simulation model as SUT. The following subquestion is defined for this:

4. How can automated test packages be developed, that use or integrate MBT techniques for verification of a simulation model's results?

Chapter 2 gives a literature study and case study analysis in order to answer subquestion 1. Section 3.1 then gives the requirements and preferences for the selection of an MBT software tool to be used in this project. Among several available open-source options, the tool AltWalker has been selected. The basic functionality and potential problems of this tool are explained in Section 3.2 based on documentation and experience with it.

The simulation models used in the case studies are given in Chapter 4. The intended purpose of each developed test package is introduced here as well[1]. Chapter 5 shows how test models have been developed in the case studies to answer subquestion 2, the step-by-step verification of the dynamic behavior of simulation runs. The options chosen from MBT taxonomies in Chapter 2 are explored here. Black-box testing with a restricted SUT interface is considered for one case study in particular, to answer subquestion 3.

Chapter 6 focuses on subquestion 4: the verification of the results of multiple simulation runs. A simulation model for which analytical solutions are known is analyzed. Numerical options for this aspect of verification are given. Functionality is added to a test suite here to automatically run and analyze multiple test cases. Finally, a discussion and conclusion are given in Sections 7 and 8.

[1] The code for the developed simulation models and test packages is available online at https://github.com/montequercus/MBT-sim

2

Literature study

This chapter discusses the literature study that is done to answer subquestion 1: "What known options for model-based testing (MBT) are relevant for testing simulation models?" This is done by first considering the different definitions of MBT in Section 2.1. Then, it is summarized which properties of simulation models make testing in general, in Section 2.2. Dimensions of MBT are taken from three taxonomies and discussed in Section 2.3. It is hypothesized here which options may be relevant for testing simulation models, and therefore which options need to be explored in the case studies in Chapters 4 - 6. The chosen options are also used for the selection of an MBT software tool in Chapter 3. In Section 2.4, case studies from literature are used to further determine which options for MBT may be relevant for testing simulation models. Reading questions for this case study review are based on definitions and options from taxonomies and other literature discussed in Sections 2.1 - 2.3.

2.1. Model-based testing

The most common definitions for MBT are discussed first to distinguish what features MBT can refer to, what a typical testing process looks like, and how MBT techniques are currently used. Reading questions are formulated along this process for the case study review in Section 2.4.

2.1.1. Definitions for model-based testing

MBT means, in the broadest sense, that some parts of a software testing process are automated by the use of a model. The system under test (SUT), which itself can be a (simulation) model, is tested using a 'test model'. The test model can be used to generate a large number of test cases, which are a series of inputs to the SUT along with the associated expected behavior.

The degree of automation in the testing process can vary. For example, some case studies that claim that they apply MBT use a test model only to generate the initial parameters for a test case. Utting and Legeard (2007) therefore draw a line of what constitutes MBT based on the level of automation: at a minimum, the SUT's expected behavior should be generated automatically using a test model. The test package should be able to "accurately" assign verdicts on whether the SUT's behavior conforms to the expected behavior. This is done with oracles, that can be defined in the test model. The term 'oracles' is used for the functions that can generate verdicts on the test, by comparing the SUT's behavior to the (generated) expected behavior. Assertion functions are examples of oracles (Li & Offutt, 2017). Checking of these oracles during test execution should be automated as well, according to Utting and Legeard (2007), so that a 'pass' or 'fail' verdict can be assigned automatically to a test case [1]. Oracle checking is often still done by hand.

Similar requirements for what constitutes MBT are given by other authors. Tretmans (2008) for example states that MBT implies that a model of the "desired behavior" of the SUT is used. Such test models are often based on the same requirements as the SUT, but they are more simplified or focus only on some part of the SUT (Utting & Legeard, 2007)

[1] Oracle checking thus means that verdicts on test cases are made automatically using the oracles in the test model. The term 'automatic generation of oracles' is used for this process in some literature, but this term could also mean automation of the test *development* process, when functions are generated based on requirements. This is not required for MBT.

Even with these stricter definitions of MBT, there are choices left regarding which parts of the testing process to automate. Some tasks will however always need a manual tester, such as: defining the test objectives, writing code to make the tests executable on the SUT, and drawing conclusions from the test's verdicts. Manual testers may also be needed to make verdicts of entire test cases, if no oracles are written to do so automatically. The notion of different levels of automation leads to a question for the case study review: "Which parts of the testing process have been automated?"

2.1.2. Functioning and development of model-based tests

Any testing process can generally be divided into three phases: (i) Test generation, (ii) Test execution, and (iii) Analysis of test results. The first phase, test generation, results in a test suite consisting of a number of test cases. Each test case defines a sequence of interactions with the SUT, and the associated expected behavior. Oracles are included in the test cases, to compare the SUT's output to the expected output and make verdicts.

In the second phase, a test suite is executed on the SUT. This can be done automatically with a test execution tool (Utting & Legeard, 2007). An option for execution is to do it step-by-step, which means that for every interaction with the SUT a verdict is made using the oracles. In the third phase, analysis of results, a manual tester is still needed to interpret the verdicts and take action.

There are two main approaches for the order in which test generation and execution are done: offline and online testing. In offline testing, test cases are generated first, and they are executed on the SUT afterward. In online testing, the generation of test cases and the execution are one integrated process (Utting & Legeard, 2007). Online testing can be done step-by-step: an MBT tool may generate one step of a test case, execute it, process the SUT's response, and only then generate the next step. This step-by-step approach enables reactive testing, where the SUT's output is used as an input for the generation of the next test steps. Reactive testing is generally done when the SUT is nondeterministic (Utting et al., 2012, chap.3). Since simulation models for M&S are often nondeterministic, this project will consider online, reactive tests.

MBT is unique in that a large number of test cases can be generated by a program (Tretmans, 2008). The development process for MBT starts with making one or multiple abstract model(s) of the SUT. An abstract model can be used to represent the SUT's behavior, or it can represent the environment that the SUT interacts with. Formally, there is a distinction between abstract test cases and executable test cases. Abstract test cases can already describe the inputs to the SUT and the expected output, but this is limited to high-level concepts of the SUT only. Abstract test cases are made into executable test cases, also called scripts, by making references to low-level concepts of the SUT and its code for each step in the test case. (Tretmans, 2008; Utting & Legeard, 2007). This distinction between abstract and executable tests is less formalized for online testing. There, what should be done in the next step is generated by a model, and this is immediately mapped to low-level instructions for the SUT. (Utting & Legeard, 2007, chap. 2)

Many programs are needed for an MBT package to perform the aforementioned steps. Utting and Legeard (2007) give an overview of the required programs. This overview is extended by distinguishing between functionalities that are covered by pre-existing MBT software tools, and functionalities that the developer of an MBT package has to develop themselves. A schematic based on this overview is shown in Figure 2.1. The required programs are:

- One or multiple abstract test models. Again, these can be an abstract model of the SUT's behavior or environment. This is one of the main programs that needs to be developed in order to do MBT. An abstract model is preferably based on the same functional requirements that were used for the SUT, but not on the SUT's implementation of these requirements, i.e. the SUT code. Choices must be made by the modeler on what aspects of the SUT to focus on. This is informed by the testing objectives (Utting & Legeard, 2007, chap. 2). When an MBT package is used, it must be assumed that its abstract model is valid, meaning that it represents the modeled system correctly (Tretmans, 2008)

- A program must be used that can generate abstract test cases, also called test paths, based on the test model(s). This functionality of test path generation is commonly provided by MBT software tools, thus it does not have to be developed from scratch. These tools often let a manual tester choose settings for how test cases are generated based on the models, the most important being coverage criteria. Coverage criteria can be used to instruct that certain parts of the (abstract) test

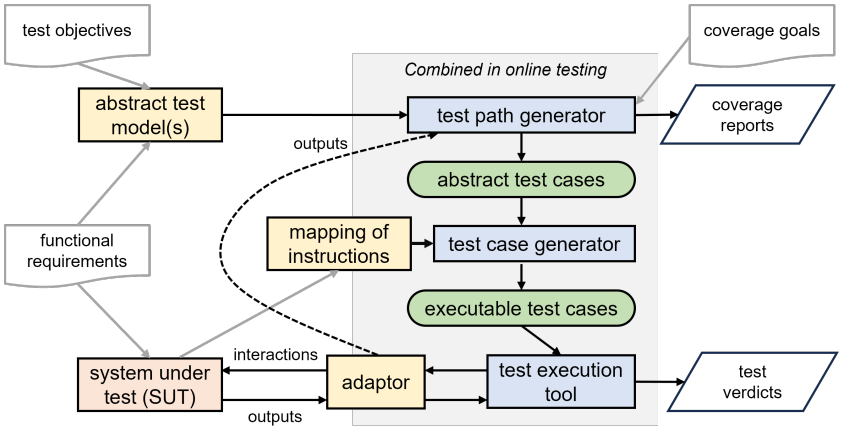


Figure 2.1: Functioning of a model-based test package. The dotted line indicates an option that is only available for online (reactive) testing. The gray lines indicate what information is used during the development or use of the associated programs. The programs in orange are the ones that will be developed for the test packages in this project. All other programs can be provided by pre-existing MBT tools. Adapted from Utting and Legeard (2007, p. 27) and Marinescu et al. (2015, p. 94).

model should be covered in a test case (Utting & Legeard, 2007, chap. 2). MBT tools can often provide coverage reports, that show what parts of the abstract model have been covered in a test case.

- A program must be made that can map the abstract test cases to concepts of the SUT's implementation, in order to make executable test cases. Such a program is named a 'test case generator' in Figure 2.1. This functionality is provided by most MBT software tools as well; abstract test cases can be mapped to sets of instructions for the SUT, and to oracles, including expected behavior, that can be checked during test execution.
- The actual mapping of abstract test cases to SUT instructions and oracles must be made by a developer. This is a tedious process, that can be done along with the development of the abstract test model(s). Unlike for the abstract model, the developer needs knowledge of the SUT's low-level implementation for this process. The program that contains the mappings will be called 'mapping of instructions' throughout this project.
- A test execution tool is needed to execute the test suits. MBT software tools often provide the following functionalities for this: messages for interactions with the SUT can be sent, the SUT's output can be compared to the expected output, and verdicts on entire test cases can be generated, for example by using the oracles present in the test case. One important functionality is not always provided by MBT tools, namely APIs to interact with specific SUTs.
- Adapter code has to be developed for this last purpose. The code should let the MBT software tool interact with the SUT, and it should interpret the SUT's outputs for use in the test execution tool.

2.2. Properties of simulation models

Dynamic verification, or testing, of simulation models is a difficult problem. The properties of discrete-event, stochastic simulation models that make them difficult to verify are now summarized. Some of these properties distinguish simulation models from other types of software programs. These difficulties may explain why MBT of simulation models is not an established field of research, while MBT is broadly used for other software programs. The properties are summarized in Table 2.1, and they are further discussed below.

Outputs are time series. The outputs of simulation models can be time series, which are difficult to analyze, naturally because they can show different features over time. That also makes it hard

Table 2.1: Summary of relevant properties of simulation models that make testing difficult

Property of simulation models	Consequences for testing
Outcomes are time series	Hard to define requirements. Further analysis such as statistical tests are needed. A warm-up period may be needed to reach steady state.
Model is stateful and reactive	Relevant variables must be chosen to test. Causality of outcome patterns and faults is concealed.
Output is highly dependent on inputs	Tests of simulation runs with different input parameters are needed.
Model has inherent stochasticity	Multiple replications with the same input parameters should be tested. Exogenous stochasticity can be fixed.
Model may have unknown nondeterminism	Internal processes of execution should be considered as well.

to define requirements for how a variable may change over time. Analysis of time series data often requires statistical techniques, for which the mean or variance of the data must be taken (Roungas et al., 2018). Furthermore, a warm-up period may be needed until a timed variable reaches a steady state that is useful for analysis (Sokolowski & Banks, 2010).

Model is stateful and reactive. Large and complex simulation models simply have many states that contribute to the end result, or that can be subject to testing. Defining and testing whether all states are correct, at all given times, is difficult. Furthermore, executing large simulation models can be computationally expensive, but execution is needed for testing. And lastly, simulation models are reactive systems, which means that their response to (external) events depends on the current state.

Output is highly dependent on inputs. A simulation model is meant to be used for a certain range of input parameters. The outputs (over time) of a simulation model can be highly dependent on the initial inputs given. The inner workings and effects of interacting processes cannot be overseen before execution. Otherwise, there would be no reason to build an executable simulation model. This means that simulation runs that use different sets of input parameters should be tested.

Model has inherent stochasticity. The project focuses on stochastic simulation models, as they can be used to analyze systems with many uncertainties as are common for decision-making. Stochastic simulation models are more difficult to analyze than deterministic models, as statistical techniques must be chosen and used (Sokolowski & Banks, 2010, p.21). Multiple simulation runs with the same initial conditions, also called replications, will still give a range of results because of this. Therefore, multiple replications must be considered in testing. Which number of replications is needed, must itself be analyzed based on statistical tests (Sokolowski & Banks, 2010).

Two types of stochasticity can be distinguished. Endogenous stochasticity arises from within the model itself when (pseudo-) random values are taken from a distribution. This can be the time for a future event or an updated value for a property (Mihram, 1972, p.20). Exogenous stochasticity arises from different initial parameters or from random events from the environment that influence a simulation run. Mihram (1972) proposes that stochastic effects can be partially suppressed for dynamic verification by making exogenous variables deterministic. It is hypothesized that this can be achieved with MBT when a test model is used to make inputs for a simulation model.

Model may have unknown nondeterminism Lastly, there are potential issues with simulation models that could lead to unexplained results or faults, that are difficult to test. For example, synchronization issues in distributed simulations can lead to inconsistent results, not because of faults in the simulation model, but because of faults in the machines that execute them. Topics that lead to such problems, such as distributed simulation and real-time simulation, are not within the scope of this project.

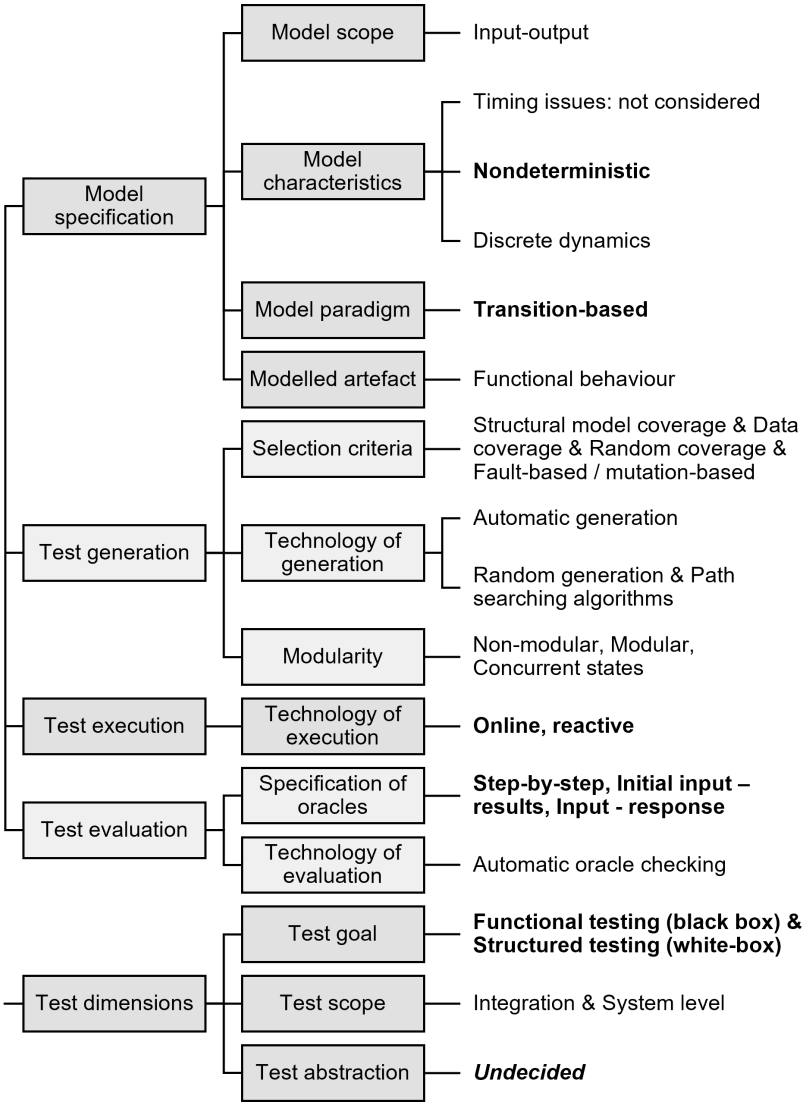


Figure 2.2: Taxonomy of MBT and dimensions of testing. The options that have been chosen to use or further explore in this project are given on the right. Options of focus for this project are given in boldface. Adapted from Marinescu et al. (2015), Utting et al. (2012), and Zander et al. (2011)

2.3. Relevant options from MBT taxonomies

Taxonomies for MBT from literature are now used to list the known dimensions of MBT. The available options for each dimension are summarized. Dimensions of testing mentioned in the following three taxonomies are considered: a main taxonomy by Utting et al. (2012), a more restricted version by Marinescu et al. (2015), and an altered version by Zander et al. (2011) that is specifically about testing embedded systems. Embedded systems are deemed relevant for this project, as many MBT case studies are about testing embedded systems, and because embedded systems share some similarities to simulation models. The reader is referred to these sources for a more thorough explanation of all dimensions and options. The taxonomies are overall divided into four classes: model specification, test generation, test execution, and test evaluation. A fifth class is added to this: dimensions for testing in general.

The taxonomies are used to further scope this project by hypothesizing which options may be relevant for testing simulation models. This is done by matching options to relevant properties of simulation models, as defined in 2.2, where applicable. If this is not possible, it is considered what options are supported by most MBT tools. The results of this analysis are shown in Figure 2.2. The chosen options will be further explored in the test packages that are developed in this project.

2.3.1. Model specification

The model specification concerns the test model(s), what their purpose is, and how they are modeled.

Scope

'Scope' is only mentioned by Utting et al. (2012). It simply means whether the test model only provides inputs for the SUT, or whether it gives the expected input-output behavior. Only this last option is considered in this project. Some existing case studies on MBT do not use the test model to generate expected behavior, as is later exemplified in Section 2.4. These test models have an input-only scope. That does not count as MBT according to the definition of Utting and Legeard (2007) that is used throughout this report. The related question in the case study review is: "Is the expected behavior generated by a test model?" This also gives insight into what phases of the testing process have actually been automated.

Utting and Legeard (2007) further indicate that test models can be 'input models', also known as 'environment models'. These are made to give the same inputs that the SUT's environment would normally give. This can be contrasted with test models that simulate the SUT itself in some way, in order to generate its expected behavior. A term for such test models is 'behavior model'. A test with an input-output scope means that "some aspects of the environment and some aspects of the SUT" are specified in the test model (Utting & Legeard, 2007, p.302).

Model paradigm

An important dimension is the modeling paradigm, or notation, of the test model. The main options for notations are state-based (pre/post), transition-based, stochastic, and data flow notations (Marinescu et al., 2015; Utting et al., 2012). Only transitions-based notations will be considered in this project. This choice is made because this type of notation is most discussed and recommended in literature. Furthermore, it is the most supported by MBT software tools (Marinescu et al., 2015, p.129). TNO (2021) also recommends transition-based models for MBT of simulation models for the same reasons. These claims are further substantiated in Chapter 3 of this report. Another advantage of transition-based models is that they have notations that are familiar to developers of simulation models, such as finite state machines, labeled transition systems, and input-output automata. Since there are many options for transition-based models, a question is added for the case study review: "What notation is used for the transition-based test model?"

It is found in Chapter 3 that many transition-based MBT software tools support directed graphs that are in essence extended finite state machines (EFSMs). These are graphs consisting of vertices that represent states, connected by edges that represent state transitions. The EFSMs can have internal variables, which can be assigned by state transitions or actions related to states. Edges can have guards, which are conditions for certain transitions to be available or not. Sabbaghi and Keyvanpour (2017) argue in a comparison of state-based models that EFSMs are suitable to test reactive systems. TNO (2021) also recommend EFSMs for MBT of simulation models. Other options like timed automata or timed labeled transitions systems exist, but these are more suitable for real-time systems with timing issues (Sabbaghi & Keyvanpour, 2017). Such systems are not within this project's scope, see "Model characteristics" further on.

A hypothesized difficulty of testing simulation models is that they can generate events themselves without any input given. These are endogenous events. New events, and thereby eventual state transitions, are generally created in simulation models by advancing the simulated time. 'Time advance' can thus be seen as another input to the simulation model. This means that in this project the test models could give 'time advance' instructions to the SUTs. It is also recommended by TNO (2021) that MBT tools for testing simulation models should be able to advance the (logical) time in the SUT. A test model then also needs to keep track of its own time variable, in order to compare this to the SUT's time.

This approach can be expanded with a test model that also gives proper inputs, namely exogenous events, to the SUT apart from advancing its time. However, that may not be needed for simulation models where inputs from the environment are not necessary for proper functioning.

This issue of model notation leads to another reading question: "How is time advancement implemented in the test model?" The model paradigm of a simulation model that is being tested may be relevant for answering this question as well. Berkenkötter and Kirner (2005) mention for testing real-time and hybrid systems that if a SUT has discretized time, the time can simply be kept in the abstract

model by incrementing a counter. However, if 'dense time' is used, modeling time becomes more complex. As simulation models are software programs, they only use a discrete time domain. Still, different approaches to the time domain in the modeling paradigm used in the SUT may influence the test model design. For example, a formalism like agent-based modeling uses discrete time steps of fixed length, while a formalism like discrete event simulation has time steps of varying length. It is not found in the literature if this would matter for test model design, so this should be further explored in this project.

Model characteristics

Utting et al. (2012) distinguish some special characteristics that may occur in the SUT: the type of dynamics, timing issues, and nondeterminism. Their idea is that the test model should accommodate or include these characteristics of the SUT.

The type of *dynamics* means that the SUT can be a discrete, continuous, or hybrid system (Utting et al., 2012). This relates to the time domain that is used: a discrete system uses a discrete time domain, while continuous and hybrid systems use a dense time domain. Real-time systems will not be considered in this project; only simulation models that use discrete time will be tested. Therefore, the test models will only use discrete time.

If the SUT were a real-time system, then *timing issues* would be relevant. Utting et al. (2012) explain that these should then be accounted for in the test model as well. Timing issues occur when a model has to give a response within a certain time frame to their environment or to other components, as is often the case for real-time systems. Utting and Legeard (2007, p.302) conclude that systems with timing issues are "hard to test", and do not indicate how a test model should be designed to accommodate for it.

While real-time systems are not considered in this project, simulation models for M&S often have a problem similar to timing issues, namely timing constraints. This means that a response to an input is expected within a certain response time. Timing constraints can be relevant to include in tests. The time dimension of simulation models then should be considered in the test design. This gives a similar conclusion as found for the 'Model paradigm': the test model should have the same notion of time as the SUT, and it should be able to advance the logical time in the SUT.

The last characteristic given by Utting et al. (2012) is that the SUT and test model can be either *nondeterministic* or deterministic. They relate nondeterminism to two issues: time jitter and concurrency. These two issues are outside the scope of this project. A related issue that is not mentioned in the taxonomies is nondeterminism due to the stochasticity that is built into simulation models. This important characteristic of simulation models may require nondeterminism in the test models as well. A set of input parameters can and should lead to different outcomes from the SUT every run, so the expected behavior modeled by the test model can be a range of values. It must therefore be explored how a test model can make verdicts for a range of possible outcomes.

Utting et al. (2012) further mention that a nondeterministic test model can be used to test deterministic systems. From the search for MBT tools in this project, see Chapter 3, it is found that such nondeterministic test models are often used for testing user interfaces like websites. A test model is for example made that represents a user who gives inputs to the website. Nondeterministic path-finding algorithms are used to produce a large number of test cases from such test models. These test cases are used as inputs for navigating the SUT, which is deterministic in this case.

It is unknown how a nondeterministic test model could be used to test a nondeterministic SUT. A question related to this issue is defined for the case study review: "Are the SUT and/or test model deterministic or nondeterministic?"

Test artifact

Marinescu et al. (2015) extend the taxonomy with the test artifact, which is the type of information or requirements that are modeled for testing. The options given are functional behavior, extra-functional behavior, and architectural descriptions. Zander et al. (2011) mention one other option: a model of the testing strategy itself. The main goal of this project is to test the behavior to functional requirements, so only the SUT's functional behavior will be modeled in the test models.

2.3.2. Test generation

Test generation includes all dimensions for generating tests from the test model.

Test selection criteria
Test selection criteria are needed to select relevant test cases to be generated using the test model. Across the three taxonomies considered many options are given for test selection criteria. No reasons are found now to not consider any of these options for the purpose of testing simulation models. Utting et al. (2012, p.303) mention too that "the 'best' criterion is not possible in general". It is found that two options are most supported by MBT tools: structural model coverage criteria, and ad-hoc (manual) test case specification criteria (Marinescu et al., 2015, p.130). This last option is not considered, as it is not relevant for a truly automated testing process.

Structural model coverage criteria are relevant for transition-based models. A criterion can indicate whether all nodes (states), all edges (state transitions), or certain paths through the EFSM should be covered (Utting et al., 2012). It should be noted that this means coverage of the test model; this does not guarantee coverage of the SUT's code, which would be relevant for structural or white-box testing.

Another idea is that test models could model states or transitions that are not supposed to happen in the SUT, for fault detection. It is unknown how coverage criteria would work for this purpose. Therefore, this idea is further explored in this project.

Other criteria are *data coverage criteria* and *random generation*, which includes input parameter sampling. Data coverage may be relevant for making test cases that describe different simulation runs with varying input parameters. This includes boundary analysis and domain analysis Utting et al. (2012). TNO (2021) also recommends that MBT tools for simulation models should support data coverage criteria for these purposes. Lastly, *fault-based* and *mutation-analysis-based* criteria could be used to validate the test package once completed.

Technology for generation
Zander et al. (2011) extend the taxonomy with the 'technology' used for test generation. Relevant to this project is that *automatic generation* of test cases will be used. Some options on how to do automatic generation are given: by randomized generation, graph search algorithms, or model checking. These options are left open for this project.

Modularity
Modularity of the test models is not mentioned as a dimension in any of the taxonomies. It is added as a dimension in this project, as it may enhance the benefits of MBT. Many MBT software tools for test generation support modular test models, see Chapter 3. As simulation models are often built out of similar components, the use of modular or even composable test models may be useful to test different compositions of these components. Likewise, one test model could be reused to test similar components within the SUT. TNO (2021) also recommends that an MBT tool for simulation models should support modular test models.

While not mentioned is a dimension, examples of modular test models are found in the literature. The taxonomy of Utting et al. (2012) does not mention modularity or composability, but an example is given of an MBT software tool that uses state-based test models that are composed out of more simple models. A concept related to modular test models is testing of components within a SUT. Only the taxonomy of Zander et al. (2011) mentions such method, where the SUT exists out of 'subsystems' that can be tested separately.

The dimension 'modularity' could also be placed under the dimension of model specification. It is relevant for test generation as well, in the sense that an MBT tool should have some method to combine multiple test models and use them for test generation. Options for transition-based models are that (sub)models are combined into one test model or that concurrent states are used. Since modularity is not required for testing simulation models but may be valuable for MBT, it is decided to consider both non-modular and modular setups in this project.

2.3.3. Test execution
Test execution includes all dimensions about making abstract test cases executable and executing test cases on the SUT.

Technology for execution
Online vs. offline testing can be seen as properties of both test generation and execution, and these options are mentioned in all three taxonomies. As explained in Section 2.1.2, only online testing will

be considered as it allows the SUT's outputs to be used for test generation. *Reactive testing* is a more specific term for this approach Zander et al. (2011). Online, reactive testing is generally recommended for nondeterministic SUTs, and TNO (2021) recommends online testing for MBT of simulation models specifically. However, it is found that most MBT literature, except for some case studies, does not explain how test models can be designed for reactive tests. A related question is therefore used in the case study review: "Is online testing applied, and is it reactive?"

Execution options
Zander et al. (2011) extend the taxonomy with execution options specifically for embedded systems, such as hardware-in-the-loop and processor-in-the-loop. No relevant simulation-like concepts are mentioned here. A promising idea mentioned is to generate test logs along with the verdicts, that give more information on the SUT's or test model's internal states. That would be most useful for white-box testing.

2.3.4. Test evaluation

The taxonomy by Zander et al. (2011) uniquely identifies test evaluation as a distinct category, with two categories: specification of evaluation, and technology of evaluation. In order to understand options for evaluation, a clear definition must first be given of what an 'oracle' is. Oracles are defined as functions or methods that are used to assign a verdict to test cases, or to some part of a test case. An oracle can consist of two parts: an expected behavior, and a procedure to assign a verdict to the SUT's behavior by comparing it to this expected behavior. Assertion functions are examples of oracles (Richardson et al., 1992, p.106). Because 'expected behavior' and 'oracles' are closely related terms, literature often blurs the distinction between them. Another source of confusion is that case studies often do not make clear whether an oracle assigns a verdict to one step in a test case, or to an entire test case. This is likely seldom mentioned because in many MBT tools, a 'fail' verdict in one step immediately leads to a 'fail' verdict for the entire test case.

Specification of evaluation
Zander et al. (2011) distinguishes multiple options for the specification of evaluation, i.e. which procedures can be used to assign verdicts to tests. An option, derived from this taxonomy, is to use oracles in a step-by-step approach, as explained earlier. Inputs from the test cases can be associated with some expected behavior, which is compared to the SUT behavior in an oracle; for example, an assertion function. This approach is useful for detecting failures, which are undesired outputs or even crashes that occur during test execution (Utting & Legeard, 2007, p.405). It is hypothesized that a step-by-step approach with oracles can help to precisely indicate where in the SUT a fault exists, and why.

Zander et al. (2011) show that there are alternatives for this step-by-step approach. These alternatives are mostly relevant for testing systems that give (continuous) signals over time, such as embedded systems. If a reference signal is available for testing, that can be seen as the expected behavior. It is then an option to compare the entire SUT signal to the reference signal. Another option is to focus on the features of both signals. This means for instance that it is expected that a signal increases for a particular time, and then decreases.

Since simulation models for M&S often produce time series data, which are similar to signals, these options could be used in this project as well. Especially the comparison of the features of time series data may be useful since the output of a simulation model can be a range of variables over time due to stochasticity. If tests were to integrate sensitivity analysis, it could be useful to examine only the direction (ascending, descending, or static) of a variable over time.

An option that can be added is to use aggregate data generated from time series outcomes, such as the mean and (standard) distribution. Aggregates can condense the results of the SUT and the associated test cases into single numbers. This makes it more convenient to compare multiple runs of a simulation model, and to consider a range of outputs. Statistical tests could be used for this. This method is therefore considered to address subquestion 4 in this project. The use of aggregate data and statistical tests in an MBT test package is not mentioned in any of the taxonomies. This is likely because MBT is more geared towards a step-by-step approach.

Some other events should lead to a 'passed' or 'failed' test verdict, which do not involve the expected (functional) behavior of the SUT. If the SUT or test model for example has a crash during test execution, then naturally the test has failed. The test selection criteria mentioned in 2.3.2 can be involved in test evaluation as well. For example, a criterion commonly used for test models is that all vertices should

be reached in one test case. If the test path generator fails to do so, then the test case has 'failed' as well. In reactive tests, failure to traverse the entire test model can indicate a fault in the SUT as well.

Technology of evaluation
Zander et al. (2011) give some options for the technology used for evaluation. Firstly, the expected behavior can be automatically generated using a test model, or it can be made by hand. This dimension is similar to the 'model scope' mentioned in Section 2.3.1. As established earlier, this project only focuses on automatic generation of the expected behavior.

Secondly, the assessment itself can be automated or not. This refers to the checking of oracles: is it done automatically during test execution, or is a manual tester needed for this? To exemplify: a test could have automatically generated expected behavior, but still necessitate manual assessment. For instance, if the test model generates time series data, a manual tester compares this to data from the SUT based on face validity.

Of course, the assignment of verdicts should be done automatically in this project, as the intent is to automate as much as possible. This topic is related again to the reading question for the case study review: "What parts of the testing process have been automated?"

2.3.5. Test dimensions

Some other considerations are important for software testing in general. These are not part of the MBT taxonomies, but they are relevant nonetheless. These dimensions are the intended goal, scope, and level of abstraction of the test package (Zander et al., 2011).

Test goal or test characteristics
The terms 'test goals' and 'test characteristics' are used with overlapping meanings by different authors. Zander et al. (2011) state that the test goal is simply the purpose of testing a SUT. They give three options for dynamic testing: structural, functional, and nonfunctional tests. Nonfunctional tests are outside this project's scope: they focus not on the SUT's behavior but on metrics like performance and usability (Zander et al., 2011). *Functional tests* assess whether the functional behavior of the SUT conforms to its functional requirements. *Structural tests* concern the internal structure of the SUT's implementation (Zander et al., 2011). Utting and Legeard (2007) further distinguish robustness testing from functional testing. In *robustness* tests, errors are found when the SUT is run with invalid conditions (Utting & Legeard, 2007). This can be a quick way to test a simulation model: MBT could be used to automatically generate and run many (exogenous) events that are possible but unlikely to happen during a normal simulation run.

This dimension of test goals can be related to two approaches for MBT, introduced in Section 1.2.4: functional tests are done as black-box tests according to Zander et al. (2011), while structural tests are done as white-box tests. Here again, the two related definitions of black-box tests are relevant. Black-box tests can mean that the test model is developed based on the functional requirements rather than on the SUT's implementation, or it can mean that the SUT is seen as a black-box model during testing. The test suite ideally only has interaction with the SUT's interface. This last definition gives the most common approach for MBT. Utting and Legeard (2007) even state that MBT per definition is black-box testing. The functional requirements do not describe details of the SUT's implementation, so the implementation is unknown to the test. They also state, using the same logic, that testing is a distinct phase of a software development process that comes before debugging. Tretmans (2008) treats MBT as a formal technique that is only for functional testing with a black-box approach as well.

However, some authors state that MBT has been used for automated white-box testing in recent years (Zander et al., 2011). White-box testing implies that the test model is developed with knowledge of the SUT's code (Utting & Legeard, 2007), or that the test package can access internal variables or interact with internal processes of the SUT. The distinction between black-box and white-box testing can therefore be vague: a test package can also be made that applies both approaches (Zander et al., 2011, p.4). Of course, a test may access more variables of the SUT than that it would generally use for communication with its environment. Then, the SUT interface is changed for testing. Similarly, one could make a test model based only on requirements, thus a black-box approach, and include white-box and even debugging features in the mapping of abstract test cases to executable code. With this hypothesis, special attention is given in this project to whether test development is done based on requirements or not.

Because of the unclear distinction between black-box and white-box testing, this issue is considered in the case study review with the question: “Can the process be considered black-box and/or white-box testing?”

Test scope
The test scope or scale is the “granularity of the SUT” (Zander et al., 2011, p.6), or which parts of the SUT the test focuses on. The options are in order of decreasing granularity: unit, component, integration, or system testing. In *unit testing* and *component testing*, small parts of the SUT would be tested in isolation. These two scopes are not relevant for this project, because the interaction between components during execution is essential to understand how a simulation model works. Therefore, only *integration* and *system testing* are considered. Integration testing focuses on specific components functioning within their environment, while system testing focuses on an entire simulation model (Zander et al., 2011).

The test scope, test goal, and model specification are closely related (Utting et al., 2012, p.298). It is hypothesized that a full system test of a large model could require a more white-box approach. In order to test a large simulation model, it is less useful to make a distinction between the SUT’s interface and its internal variables or this distinction may even be left out. For component and integration testing, however, a SUT component is tested of which the interface can be clearly defined. The inputs and outputs on the interface are simply what the SUT component uses to communicate with its environment, i.e. other components. Such SUT components can thus be regarded as a black box during testing. TNO (2021) gives two case studies of similar tests, where a test model simulates the environment of an SUT component, and checks if it responds to messages as expected.

Test abstraction
The level of abstraction of the test objectives and the test model is a difficult problem. It has been called the “key step” for developing transition-based test models (Utting & Legeard, 2007, p.187). It could also be seen as a dimension of model specification. A test model becomes more understandable and can be reused for other SUT implementations if it is more abstract. However, the mapping to executable test cases must still be feasible (Zander et al., 2011). This problem may be even more difficult when the SUTs are simulation models, because they often have many possible states and their response to inputs is complex. The right elements from a simulation model must thus be chosen if an abstraction of it would be made. This problem must therefore be explored further in this project’s case studies in Chapter 5. Apart from the taxonomies, a case study by Pretschner et al. (2005) mentions two approaches to achieve abstraction: by encapsulation or by leaving out details of the SUT or requirements.

2.4. Review of case studies

A review is made of several case studies where MBT is applied. It is found through literature search that only a few papers discuss SUTs that are simulation models. Therefore, other case studies are discussed as well that have SUTs that are similar to simulation models, or that use test models with a time domain or a nondeterministic aspect. Only papers that use transition-based models are considered as only that notation type will be used in this project.

This case study review supplements the discussion of taxonomies from Section 2.3. The goal is again to find options for MBT that may be relevant for testing simulation models. It is also reviewed if the taxonomies should be extended with additional dimensions. Two general reading questions for this case study review are now given:

- What is the main reason that MBT is used?
- Is the SUT a simulation model?

The other reading questions for analysis, which were formulated throughout Sections 2.1 - 2.3, are now repeated:

- What parts of the testing process are automated?
- *Test approach:* Can the process be considered black-box and/or white-box testing?
- *Model scope:* Is the expected behavior generated by a test model?
- *Model characteristics:* Are the SUT and/or test model deterministic or nondeterministic?

- *Model paradigm:* What notation is used for the (transition-based) test model?
- *Technology of execution:* Is online testing applied? Is the test reactive, meaning that it uses the SUT's output for test generation?

2.4.1. Summary of relevant case studies

The approaches used in noteworthy case studies are now summarized using the reading questions. This will show that MBT is applied for different purposes, with varying levels of automation of the tests.

Embedded systems as SUT

MBT can be a cost-effective method to test a piece of hardware and/or its computer system, without actually using that hardware in its real environment. Most research on (model-based) testing of embedded systems is from the automotive industry. Embedded systems are deemed relevant in this project, because much research is done on them, and because of their similarities to simulation models. Some properties of simulation models given in Section 2.2 can be compared. Firstly, both embedded systems and simulation models are reactive systems. The response reactive systems give to external events, or inputs, depends on their internal state (Richardson et al., 1992, p.105). A second similarity is that embedded systems have a time domain. And thirdly, embedded systems can be nondeterministic (with endogenous stochasticity), and they can be integrated in a nondeterministic environment (giving exogenous stochasticity). A difference from the simulation models used in this project is that embedded systems run in real-time. However, testing can sometimes be done in logical time.

Keränen and Räty (2012) use MBT for a hardware-in-the-loop (HiL) test with as SUT an embedded system of a car's automatic transmission controller. A simple test model generates the expected behavior. Online testing is used: a single input for the SUT, called a 'stimulus' in this context, is made at a time and immediately executed on the SUT. The test is reactive as well: the test generation is based on the SUT's outputs. And lastly, verdicts are made automatically during test execution. The authors believe that this online, dynamic, and reactive application of MBT is unique for HiL testing. For this reason, it is mentioned in this report; it is found that most other case studies on HiL testing do not focus on these topics.

Keränen and Räty (2012) interpret MBT as a purely black-box testing approach, and they see the SUT as a black-box model. Therefore, they recommend using MBT to test the "overall behavior of the SUT" (p.375). This is best done for system or integration testing.

Zander et al. (2011) apply MBT to testing of (real-time) embedded systems. Their considerations have been discussed in Section 2.3. They use reactive testing because the SUT is nondeterministic so that the test "can react to changes in model variables within one simulation step." (p.13).

Software-in-the-loop tests

Some (embedded) systems are tested in a software-in-the-loop test. Poncelet and Jacquemard (2016) do this for testing of their Interactive Music System. This is a real-time human-in-the-loop system that can play electronic musical instruments along with human players. The aim here is to automate all steps of the testing process, and it is found that custom MBT tools must be developed for this purpose. The SUT itself has no stochasticity, but its inputs are highly stochastic human players. The reason to apply MBT here is that testing would have to be done with real musicians, and would not guarantee fault coverage. Testing is done as software-in-the-loop using two test models: an environment model simulates inputs that musicians would give, and a behavior model simulates how the SUT should handle these inputs. The behavior model is notated as a network of EFSMs that have some stochasticity built in: it features "delays, asynchronous communications, and alterations" (Poncelet & Jacquemard, 2016, p.144). While the SUT is real-time in its normal application, it is run with virtual clocks during testing. Therefore, this SUT is similar to the non-real-time simulation models that are focused on in this report. It is mentioned that the tests are black box.

Unique in this case study is that the behavior model itself is automatically generated from the functional requirements. Full automation is considered feasible here because the functional requirements are very clear and formalized: they are musical scores. Poncelet and Jacquemard (2016) further show how both online and offline testing can be done. Online, reactive testing is done so that the behavior test model can adapt to the nondeterministic SUT, and so that step-by-step path generation can be used. Interestingly, Poncelet and Jacquemard (2016, p.165) explain how online testing can "prevent

state explosion". State explosion means for a transition-based model that to have any complexity a vast number of states and even more state transitions must be modeled.

A similar software-in-the-loop approach is used by Iftikhar et al. (2015). They apply MBT for verification of a simple platform video game. This SUT is normally a real-time system, but it is executed with virtual clocks during testing. Similar to Poncelet and Jacquemard (2016), two test models are used: an environment model mimics the inputs that a human player would give, while a behavior model simulates what the SUT should do in an abstract manner. The generated inputs are used on both the behavior model and the SUT. The main incentive to apply MBT here is also similar to the music system case study: manual testing would be very laborious, as manual testers would need to actually play the video game for hours.

They intend to automate all steps of the testing process. A custom MBT tool is developed for this purpose. The abstract behavior model consists of separate UML state diagrams for all components of the game. For example, there is a state diagram of all possible behaviors of 'Super Mario'. The (generated) user inputs are used as guards on this state diagram. Testing is done online, but reactive testing is not needed: the modeled user inputs are fixed, and the abstract test model should do the same as what the SUT is doing. Step-by-step, it is tested if the SUT's behavior is as expected, otherwise the test fails. The test generation uses specific technologies such as 'sneak path' and 'round trip' strategies.

Iftikhar et al. (2015) stress that their case study is about functional, black-box testing: seeing where the game crashes, whether certain events happen when they should, and if scores are tracked correctly. The game's internal states are checked continuously to assess this. A wrapper is made to summarize these into states that are similar to the more abstract states of the test model. This is done for the dynamic behavior, and the SUT's internal variables are considered: the intent is in line with the idea of identifying where (in which process) and when (during which states) a crash or wrong count occurs. That is different from, for example, having a verdict for a finished game on whether the score is correct.

Lastly, Pretschner et al. (2005) test a complex software-in-the-loop deterministic simulation model. The SUT is composed of many components, and therefore a modular test model is used: it too consists of components. A simple method is used to keep time synchronized between these submodels. All components are tested at virtually the same time.

Simulation models as SUT

Lastly, some case studies that apply MBT to simulation models are discussed. Schmidt et al. (2016, 2015, 4) base their case study on the research on MBT for embedded systems of Zander et al. (2011), but they claim that the SUTs used in such research are "not necessarily simulation models" (p.829). Schmidt et al. (2015, 4) therefore provide a framework based on Discrete Event System Specification and the Experimental Frame, to conduct such experiments for simulation models in a structured manner. They exemplify this with a case study of MBT applied to a simulation model. This has a clear time axis: it is a Matlab/Simulink model of a robot arm with a controller that must reach certain positions at certain times. The test model is used to make test cases, which are a few positions that the robot must reach at certain times. The main efficiency benefit of using MBT is the automatic generation of many such position scenarios. Online testing is not needed: it appears that the test cases, the positions, are merely initial input parameters for a simulation run of the SUT. It is not mentioned whether the test model is deterministic.

Verification is done by simply letting a real robot arm reach these positions, which are compared to the SUT's simulated positions. Thus, the scope of the test model is input-only: no expected behavior (outputs) is generated by the test model. Rather, the SUT models a real-life system, and the same inputs are given to both the SUT and the system. While the generation of expected behavior is not automated, oracle checking is. It is compared at many (sampled) points in time whether the SUT trajectory is close enough to the actual trajectory. The approach is thus clearly not step-by-step; many more points in time are evaluated, compared to the number of points in the test cases. The level of automation does not align with the definition of MBT by Utting and Legeard (2007) that is used in this project. However, a useful aspect that can be taken for testing simulation models may be the comparison of time series data.

Lindvall et al. (2017) also apply MBT to a simulation model of a real-life system: the SUT is a simulated autonomous drone that flies around in a simulated 3D environment. The inputs, or initial conditions, for each simulation run are more complex here: they are not merely some values for parameters. Rather, the entire simulated 3D environment can be seen as an input. This is thus a clear

example of where a test model is used to generate test cases, but the scope is input-only again, and oracle checking is not automated. Instead, the verdicts should be made by a manual tester who checks animations. The test execution does automatically summarize the results into short videos to be used for this manual verdict.

Van Osch (2005) apply MBT to discrete-event simulation models in the context of embedded systems. Simulation models are often used for validation of hardware implementations. These simulation models are however often not validated themselves. Two case studies are done. The first tests a simple simulation model of the alternating bit protocol modeled in χ, a discrete-event process interaction language. The MBT tool TorX is used, which uses test models notated as labeled transition systems. This is related to input-output conformance theory (ioco): the behavior model must be input-output conform to the SUT. The SUT's process that normally would generate inputs, namely input messages, for the alternating-bit protocol is replaced by an input generator that communicates with the test script. TorX then generates some messages and the expected (eventual) answer, thus the scope is input-output. The test package is validated by running it with mutated versions of the SUT.

Van Osch (2005) thus uses step-by-step testing. It is unclear whether it is reactive. The SUT is deterministic, but the test model has stochasticity implemented: it simulates a random input process, in the sense that the input times are randomized. However, for each input, exactly one expected output is known a priori. The second case study by Van Osch (2005) uses a more complex test model that uses the same principles. The intent is named here to make as few changes to the SUT's code as possible. This can be seen as a black-box approach: only the SUT's interface is considered.

Gerhold et al. (2019) also use ioco theory for MBT, but now with SUTs that have stochasticity. Two MBT model notations are used: input-output Markov automata (IOMO) and stochastic automata (IOSA). Both can be represented as states with transitions in extended finite state machines (EFSMs). They add probabilistic choices to these graphs, i.e. the probability of going to either edge, and stochastic delays. These delays give the time that the system spends in a certain state. The IOSA can model delays from any distribution and are therefore seen as the better option. A case study is given of a test of the Bluetooth communication protocol. The SUT is nondeterministic, while the behavior model is kept simple with some stochasticity: it is an IOSA, that only sets up a delay for a 'connect?' message, taken from a distribution, that counts until an 'acknowledge!' message is expected. Similarly to Lindvall et al. (2017) and Van Osch (2005), the test model generates inputs that are used on both the abstract test model and SUT. The resulting behaviors over time are compared afterward; it is checked whether the entire SUT trajectory is similar enough to the generated trajectory.

The tests have two goals: there is functional testing of single simulation runs, here called 'traces', and statistical evaluation of the results from multiple traces. It is checked if the results are within a certain confidence interval. A separate test (verdict) model is used for this last goal. This test package was validated with mutant SUTs, similarly to Van Osch (2005).

TNO (2021) apply MBT to simple simulation models in two case studies. The SUTs are components of a complex simulation environment. The test models are environment models that simulate other components that normally interact with the SUT. The test models give messages to the SUT, in order to test if the SUT gives the expected response within an expected time. The SUT is thus seen as a black box. Online, reactive test execution is used, so that the test model can check whether certain objects in the SUT exist, which it can send messages to. Interestingly, the test model also instructs the SUT to advance its logical time, until an event warrants an assertion is generated by the SUT. The scope is input-output and testing is done step-by-step, as the expected behavior from certain events is given in the test model. The case studies explore the use of two existing MBT tools for path generation using transition-based models: GraphWalker and OSMO. The test results are given in coverage reports, that show whether certain requirements for the path generation have been met during the test. The test models are nondeterministic: they randomly select objects in the SUT to test, or send randomly selected messages to the SUT.

2.4.2. Conclusion from review of case studies

It can be seen that every case study uses a different approach for MBT. New insights taken from the case studies are now discussed.

Level of automation

As expected, different parts of the testing process are automated. Some approaches use only a behavior test model while others use an environment test model or multiple test models, and some tests are reactive while others are even offline. The papers by Van Osch (2005) and Gerhold et al. (2019) are the main examples where a simulation model is tested with a test model that generates expected outputs. The paper by Poncelet and Jacquemard (2016) is an example where almost all steps of the testing process have been automated. No clear guidelines on applying MBT to simulation models can be taken from these case studies. The report by TNO (2021) uses test models of the SUT's environment, and gives some recommendations for applying MBT to simulation models. These are mentioned throughout this report where relevant.

Oracle specification and types of inputs

A few considerations are found that were not mentioned in some of the taxonomies. By considering the dimensions 'specification of evaluation' as defined by Zander et al. (2011), see Section 2.3.4, some differences between the case studies become clear. When online testing is used, in some studies oracles are not specified or checked for each step of the test case. For instance, only the traces of intermediate results are compared by Gerhold et al. (2019) to make verdicts. For offline testing on the other hand, sometimes many more data points are evaluated in oracles than there are data points in the test cases. This is done by Van Osch (2005). This seems like an important distinction to see what an MBT package may be used for. If oracles make verdicts for the SUT's response to every input that is sent to the SUT, then the test can give a clearer idea of where and when a fault occurs. This step-by-step approach is also most in line with the common practice of MBT found in the literature.

Another point taken from the case studies is that a distinction is often made between an environment test model and a behavior test model. This can be linked back to the dimension 'model scope' discussed in Section 2.3.1. Environment models are mostly used to generate inputs, thus exogenous variables, while behavior models in some way simulate the SUT's expected behavior, thus its endogenous variables. In some studies, inputs from environment models are used on both a behavior model and the SUT. In other studies, the test model is only used to generate the initial values for a (test) execution of the SUT. The point from Section 2.3.1 should be added that the test model may not even need to provide input events when the SUT is a simulation model. This is because a simulation model will generate endogenous events upon time advancement.

From the aforementioned points on oracle specification and types of inputs from test models, three approaches to testing can be deduced:

1. Step-by-step verification of endogenous variables. This lends itself to oracles that are specified for every step of the generated test case, in order to answer the question: is the event generated by the SUT, a simulation model, expected given its previous state? The oracles can represent all logic that the SUT should adhere to in a detailed manner.

2. Verification of responses to exogenous variables. Oracles can be specified for this purpose, that check whether the response that the SUT (eventually) generates is correct or within the specified time constraints. An oracle for each step in the test case is thus not necessary for this approach.

3. Verification of a simulation model's results, given its initial input values. This is what is studied with subquestion 4 in this report. Oracles should be made to check whether a given set of input parameters leads to the expected outcomes. This can be done for an entire simulation run, for instance by taking the mean or variance of time series data. It could also be done for traces of intermediate results, or by time series analysis with a reference time series or spectrum.

It should be noted that the third approach, the analysis of end results given the initial conditions, is similar to what is done for validation of simulation models. This project only considers verification, specifically the correct implementation of the functional requirements in an executable model. Validation is "always relative to a model's intended use" (Roungas et al., 2017, p.4); the practical use of the SUTs is not considered in this project. Moreover, validation is a distinct phase that is done after

verification, and all MBT sources indicate that testing is strictly for verification (Utting & Legeard, 2007). Numerical comparison of (means of) outputs to theoretical values is recognized as part of verification (Kleijnen, 1995).

3

Selection and functioning of MBT software tools

This chapter discusses the selection of an MBT software tool. An existing MBT software tool will be used to build test packages for the case studies. As explained in Section 2.3, existing tools can provide the core functionality of the MBT process: test path generation, test case generation, and test execution. The MBT tool further dictates aspects like what notation is used for test models, what types of coverage can be achieved, and whether test models can be combined. The choice of an adequate MBT tool is therefore important. This chapter therefore first gives the selection process for finding a tool. The tool AltWalker is chosen to be used in this project. Its functionality is discussed in this chapter as well. A conclusion is given on what features are missing from this tool. Further practical problems encountered during the development of the case studies are discussed in Appendix A.

3.1. Selection process for an MBT software tool

The selection process for an MBT tool is done at the start of this project. Naturally, the MBT tool should support the relevant options for MBT of simulation models that have been defined based on the taxonomies in Section 2.3. More requirements and preferences are partly taken from previous research by TNO (2021) which focused on MBT for simulation models as well. In addition to the known dimensions from the taxonomies, some practical requirements and preferences are decisive for the selection. Based on the selection process, a number of MBT tools have been tried in practice. This is described in Section 3.1.2.

3.1.1. Requirements and preferences

The technical requirements for a MBT tool that are based on analysis of the taxonomies are first discussed. The tool should support test models specified as nondeterministic transition-based models. Extended finite state machines (EFSMs) are preferred, but other transition models could be used as well: it is important that internal variables and guards can be specified in the test model. The test generation should support automatic generation of tests. The user should be able to specify the technology used for generation, such as random generation or path finding. Regarding test selection criteria, data coverage is a preferred option, so that simulation runs with different input parameters can be easily tested. Structural model coverage must be supported as well because it must be explored how this option can be relevant for testing simulation models, see Section 2.3.1. For test execution, the tool should support online, reactive testing. The tool thus has to support some method of influencing the path generation based on the SUT's outputs. And lastly, the tool should be geared towards automatic oracle checking, so that it can assign verdicts to test cases.

The practical requirements for selection are partly taken from TNO (2021). A practical requirement for the test models is that they should be editable using a graphical user interface (GUI). That feature enhances the usefulness of an MBT tool for making (abstract) models that are visual, easy to edit, and easy to communicate. A preference is that the test model is composable into submodels, that can be used in varying compositions. Another preference is that concurrent states can be used for increased

flexibility in designing test models. Additionally, the overall preference for parallel execution is regarded as well. A preference is that the MBT tool can run test cases in parallel, or that it would be possible to adapt it for this purpose. Another preference is that adapter code exists or can be made readily for communication with a SUT simulation model.

Lastly, the selection process is narrowed down by only considering open-source tools. It was found during the search for tools that many open-source MBT tools are Java-based. During the selection, a requirement is established that only those tools would be considered which can operate within a Python environment. This is because the author mostly has experience with Python, and the simulation models used as SUTs will be in Python as well. A related preference is that as much code that the tester provides as possible should be in Python. Some other requirements can only be checked by actually installing and using MBT tools: installation must be feasible and usage of the tool must not be too convoluted. Related preferences are that the documentation is clear and complete, the tool is in active development, and code examples are available.

3.1.2. Search for MBT tools

The features, alignment with the requirements and preferences, and ease of use are now discussed for MBT tools that have been found in the selection process.

Types of notations used in open-source MBT tools

A broad search for any open-source MBT tools is first done. A finding from this search was that most open-source tools are based on Java. It is found that most tools will use some semi-structured data format such as JSON to specify the test models. Interestingly, a variety of notations for transition-based test models are used by the tools. Some examples of tools with different notations are:

- GraphWalker uses a transition-based notation: EFSMs, that can be edited with a GUI. It is Java-based but the test models are specified with semi-structured data files. A Python port called AltWalker is available as well.
- OSMO does not give a name for its test model notation. It continuously checks guards, and will check the state before and after an event is triggered in the SUT. It has no GUI for the test models. It is Java-based and the test models are specified in Java as well by using annotations. A Python port is available as well, namely py-osmo.
- Modelator uses models notated with TLA+ (Temporal Logic of Actions), a specific language that can describe concurrent states. Java and Python versions are available.
- TorXakis uses labeled transition systems, a transition-based notation that can have an infinite number of states. This is more geared towards complex systems and supports concurrent states. A domain-specific language is used to specify the test models, which can be viewed with a GUI.

Candidate Python-based MBT tools

With the requirement to only consider open-source Python-based tools in active development that use transition-based notations, the candidate tools are narrowed down to the following ones: AltWalker, PyModel, and ModelTestRelax. It is first checked which requirements and preferences they meet based on their documentation and repositories. After this, the practical considerations are checked by installing the tools and attempting to run their code examples.

AltWalker is a Python port of GraphWalker. The models are transition-based EFSMs. Many test generation options are available. The documentation is quite complete, and a few code examples are available. Even more examples are available for GraphWalker. While it is Python-based, the test models are given in JSON data files, and the path generation is actually done by GraphWalker, which must be installed separately. Test models can be (re)composed out of submodels (AltWalker, 2023a). AltWalker meets most technical requirements and preferences that were given in Section 3.1.1, except that concurrent states and parallel execution are not supported. It was unclear during the selection process whether data coverage requirements for path generation were feasible with AltWalker.

PyModel is not a port of another tool. Its model notation can be seen as EFSMs because guards and internal variables are available. Interestingly, a feature is included specifically for making environment models of the testing strategy. These are called ‘scenario machines’ and are used to more specifically select test cases. A viewer for the test models is available, but they cannot be edited with a GUI. The test models can be (re)composed as well, but concurrent states or parallel execution are not possible (Jacky, 2011). PyModel’s documentation is less clearly presented than AltWalker’s, but many code examples

are available. Some simple structural coverage criteria for path generation are available, but more complex coverage criteria and generation technologies must be made by the modelers themselves.

ModelTestRelax lastly is only Python-based, as it can be controlled by a Python script, and the mapping of abstract to executable test cases can be done in Python. As with AltWalker, the test models are specified in JSON files. Even GraphWalker's GUI editors can be used for the models. A difference with AltWalker is that an executable (written in C++) is used for path generation and for part of the test execution. Also, no Python package exists for communication with this tool. ModelTestRelax uses EFSMs for model notation as well (ModelTestRelax, 2023). It is documented thoroughly and has some code examples as well. Many options are given for (re)composition of test models, and many coverage criteria and path generation options are available as well. Concurrent states and parallel execution are not possible. ModelTestRelax supports batch runs, making it possible to easily define a sequence of test cases to be run. A GUI is even available for this. That can be useful for implementing data coverage requirements.

It seems that ModelTestRelax meets roughly the same technical requirements that AltWalker does. An exception is that test execution with a connection to the SUT is not part of the tool yet. The test case must be run on GraphWalker for this.

Selection based on practicality

It is found that AltWalker can be easily installed as a Python package. The installation of the GraphWalker Java executable has some caveats but these have been solved ultimately. The online available code examples of AltWalker have been successfully executed. AltWalker's file structure is easy to understand. Simply put, one data file is the test model, another script is the mapping, and another script or commands set up the execution. Tests can be run with commands in the CLI, or they can be described using objects in Python. The `altwalker` package will interpret these objects into commands that are executed by GraphWalker. PyModel is easily installed as a Python package as well. As it does not depend on other programs like AltWalker does, the basic functionality of a test package can be specified entirely using Python scripts. However, the tester must specify commands for execution as strings. These cannot be generated based on objects as is done in AltWalker. A PyModel test package generally consists of many files. ModelTestRelax relies on some Python packages that could be installed. However, the installation of the ModelTestRelax executable was successful. Unfortunately, the code examples that work with Python scripts could not be executed by following the instructions.

It is decided to use AltWalker as the MBT tool in this project. This is based on the complete documentation, the ease of use of a GUI for editing test models, the option to run tests based on Python objects, and the fact that installation was successful and the code examples could be executed. A reader interested in using Python-based MBT tools is recommended to reconsider this choice based on personal preferences. For example, PyModel and ModelTestRelax provide better functionality for batch execution and ad-hoc specification of tests.

3.2. Functionality of AltWalker

The functionality of AltWalker is now further discussed. It is shown how important options from the MBT taxonomy can or cannot be implemented using AltWalker. This explanation will help to further understand the case studies in Chapters 4 - 6. Details on the file structure, code used, and different approaches that have been tried for new functionality are given in Appendix A. Note that AltWalker can run tests written in Python or C#; only Python is considered in this project.

Section 3.2.1 shows how abstract models are notated and how test generation can be influenced in AltWalker. Section 3.2.2 shows how a missing feature is solved, so that simulations of fixed run length can be tested. The findings are demonstrated with an example test model in Section 3.2.3. Section 3.2.4 shows how tests are set up and executed. Section 3.2.5 shows how automated testing of multiple simulation runs can be achieved. Features that are missing are discussed throughout the report. These are given in Section 3.3.

3.2.1. Functioning of test models

AltWalker supports many concepts from the taxonomies regarding model notation, test selection criteria, and test generation technology. An overview is now given based on the documentation (AltWalker, 2023a).

Abstract test models
Abstract test models are called *graphs* or *directed graphs* in AltWalker. They are specified as data structures in JSON or GraphML format; only JSON is used in this project. Vertices represent states and edges represent state transitions. The directed graphs can be seen as extended finite state machines (EFSMs) because they have internal variables and trigger conditions. The internal variables are called *graph variables*. These variables must be declared in a *model action*, and new values can be assigned in *vertex actions* or *edge actions* when an element is passed. The trigger conditions are called *guards*. Guards are used on edges: they are if-statements that use the graph variables to decide whether an edge may be passed. Multiple actions and guards can be combined (AltWalker, 2023a).

Both the actions and guards are written with snippets of JavaScript inside the abstract model's JSON file. (Sub)models can be combined by declaring the same *shared states* on vertices. Two similar GUI editors for graphs are available: GraphWalker Studio and AltWalker's Model Editor.

Mappings to executable test cases
The mappings to make the test cases executable are given in a *test script*, which is a Python file. Every element in a graph must be associated to a function in this test script. During a test, GraphWalker will generate a path through the elements in the graph, given all conditions. For each element that is passed, the associated function in the test script will be executed. The test scripts can also contain fixtures, which are functions with predefined names that are always executed before or after a test case run.

The functions in the Python test script can use assertion functions as oracles. Specialized assertion functions from the `unittest` library can be used. During test execution, these oracles are checked automatically step-by-step. An oracle that gives a `False` outcome will mark that test step and therewith the entire test case as 'failed'. This means that path generation and test execution will stop. As stopping test generation may be unwanted for testing simulation runs of fixed length, in this project sometimes error messages are used instead of assertion function in the the script. This ensures that a manual tester will be alerted of potential problems, while the test case keeps running. The `warnings` library is used for this purpose.

Since the test script is simply a Python script, it too can keep internal variables, that are different from the directed graph's internal variables. The test script and its oracle functions, or assertions, can access the directed graph's internal variables as well. This gives various possibilities for how the expected behavior can be expressed.

Other than the step-by-step oracle function in the test script, more implicit oracles can be prescribed by coverage criteria that can be set per test case. If the path generation fails to meet the coverage criteria, a 'failed' verdict will be assigned to the test case after execution.

Reactive tests
The graph, which is used for path generation, cannot access the SUT's interface directly. Reactive tests, thus tests where the SUT's output is used for path generation, are made possible with AltWalker's `data` functionality. An object called `data` is then used in the test script to access and update the graph's internal variables.

In this report the term 'test model' is used to refer to one or multiple graph(s), thus abstract model(s), along with their test script(s) that are used for a test case run. It is likely that an abstract model designed for reactive tests cannot be run isolated from the test execution. The internal variables and edges may rely on the SUT's output, which means that running the abstract model isolated would likely lead to deadlock. This makes it more difficult to validate abstract models that are made for reactive tests.

Modularity of test models
The user has the option to use multiple models in the execution of one test suite. For example, two models with an associated test script can each be run on one SUT sequentially. Another option is to combine multiple models, now called 'submodels', into one graph by using *shared states* on vertices, which are declared by giving two vertices from different submodels the same *shared name*. If the path generator comes across a vertex with a shared name, there is a probability that it will transition to a vertex in another model with the same shared name (AltWalker, 2023a). A limitation is that this probability cannot be influenced: guards and weights are not available for this transition between models.

It should be noted that this AltWalker feature is not the same as having concurrent states; rather the two submodels are combined into one graph. The path generation thus never considers multiple elements, or states, at once. Parallel execution of submodels is therefore not supported as well.

The setup for test models is modular: if a user makes two models with shared states, they could still choose to run a test with only either of these models. This may be useful for selectively testing certain components of the SUT using only one test package. One JSON file may contain multiple submodels, and likewise, submodels from different JSON files could be composed into one test model.

Test generation

AltWalker gives many options for path generation. A model can be run with different path generation options; this adds flexibility to the test design. Path generation options are specified by a combination of a *generator*, which defines the technology of generation, and a stop condition, which gives the test selection criteria.

Appendix A.2 gives all relevant options for path generation. It is found that random generation and graph search algorithms are available in AltWalker. Path generation can be influenced by guards on edges, by weights on elements, and by requirements, which are elements that must be passed. The supported test selection criteria are structural model and requirements-based coverage criteria. It is found that no distinct functionality is available for data coverage criteria and random generation.

A start element can be defined in the graph, but it is not possible to explicitly set an element where test generation should stop. An end element can be attempted by setting a requirement on an element and then running a test with a 'requirement coverage' as its stop condition.

3.2.2. Testing simulation runs of fixed length

A finding important to this project is that most path generation options will generate paths of unpredictable lengths. This means for testing simulation models that it is unknown a priori at what simulation time in the SUT the test execution would stop. Ideally, it would be possible to stop test execution once the SUT reaches its simulation end time t_{end}. It would therefore be useful if path generation and test execution can be stopped based on the value of a graph variable or SUT variable. Unfortunately, none of the test generation options of AltWalker (see Appendix A.2) are found capable of this.

Some approaches have been tried to stop test execution based on a variable. These are discussed in Appendix A.1.2. The solution used throughout this project is to include an 'end vertex' in each abstract model, which can be reached with incoming edges from all other vertices in the graph. These edges have guards for the time like $t \geq t_{\text{end}}$. A drawback of this approach is that it makes the graphs harder to understand visually. These 'end vertices' will therefore not be displayed in the figures in this report.

This simple problem shows an advantage of MBT software tools that use abstract models with different notations as is done by AltWalker. The MBT tool Osmo will for example check all guards during any step of test execution. Thus, implementing this feature would be easier with a MBT tool like Osmo.

3.2.3. Example of a directed graph

An example is now given of how a directed graph is made for AltWalker. The example is outside the context of a test package, and it is made to show how the aforementioned features of AltWalker can be used in practice. The (test) model has simple functional requirements. It should either increase an integer clock value t by 1, or it should run some other checks. These checks should be in a separate submodel. The path generation should end if t reaches 20.

The implementation of this model is shown in Figure 3.1. It consists of two (sub)models: A and B. Two graph variables are used, these are initialized using a model action. The variable `global.done` is made global so that it can be used by both submodels. One vertex, shown in green, is chosen as a start element. From `v_check_clock` the path generation can choose between increasing the clock t or going to a check. The guards `t<20` are included so that the path is guaranteed to take `e_end` once `t==20`. This model can be run with a path generation option like `random(edge_coverage(100) && reached_vertex(v_end))` to ensure that it stops at `v_end`.

Increasing the clock is done with a vertex action. The check is implemented in a submodel. This submodel can be reached through a shared name between `v_random_check` and `v_start_check`. This already shows a limitation of how shared states work in AltWalker. Because they are not given with normal edges, no guards or weights can be added to them. It can therefore not be prevented that

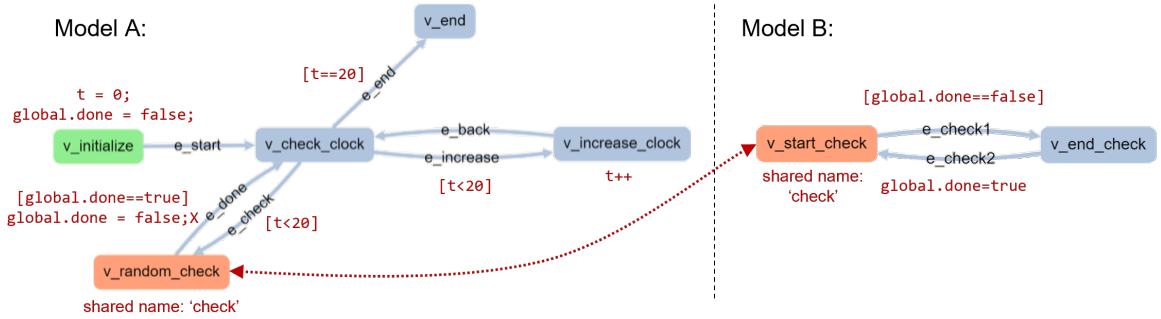


Figure 3.1: Implementation of a simple graph using two models in AltWalker. The actions and [guards] on elements are added in dark red text. An implicit edge is drawn between the two vertices which have a shared name.

the path randomly goes back to model A, before going to edge `e_check1`. Adding a weight to this edge did not change this. Lastly, guards and edge actions that use the variable `global.done` are included to make sure that the path returns to model A once `v_end_check` has been passed.

Other limitations that were identified while creating this example concern the GUI editor. Guards and actions cannot be shown on elements in the edges, so the information shown at a glance does not give the entire abstract model. Screenshots of these graphs must be edited further, to give all information of an EFSM.

3.2.4. Structure of test packages and test execution

An understanding of the file structure that AltWalker uses is important to further discuss how test execution works.

File structure of an AltWalker test package

A test package made for AltWalker will at least consist of four files: (i) One or multiple *abstract models*, or directed graphs, in JSON or GraphML files. (ii) A *test script* made as a Python script. This contains the mappings to executable instructions on the SUT, as explained previously. The test script can communicate with the abstract model via the `data` functionality, and a method is needed to communicate with the SUT. Oracles, given as assertion functions, can be associated with elements of the abstract model. (iii) The *SUT* can be any application. In the case of this project, the SUTs are all simulation models written in Python. (iv) An empty Python script `__init.py__` is needed to make the `test` folder into a module.

Test cases can be either set up using AltWalker's command line interface or from a Python script. This last option is used in this project, as Python can be used to set up a sequence of multiple test cases. For this purpose, a fifth file is introduced: a *test execution script*. The test execution script uses the API functionality of the `altwalker` library to set up one or multiple test case runs. It can contain an experimental set-up in order to run multiple experiments with different initial parameters for the SUT and/or test model. The term 'test execution script' is not found in AltWalker's documentation; most code examples there use the command line interface.

Test execution with AltWalker and GraphWalker

AltWalker's test execution is complicated as it uses multiple programs. Upon initialization of a test case, AltWalker in Python will set up a GraphWalker client, which is a Java program. GraphWalker is responsible for path generation using the abstract model. Communication between AltWalker (Python) and GraphWalker is done via TCP/IP messages. Communication between Python and the SUT may necessitate adapter code. In this project, the problem of establishing communication with the SUT is mitigated by loading the SUT directly into the test script. This can be done because the SUTs used are Python scripts as well. Further details on how the test execution script works and how the API is used are given in Appendix A.3.

Figure 3.2 gives a schematic overview of how test execution is done using the different programs and files in this project. Note that a line is drawn between the abstract model and the test script: this represents the `data` functionality that allows reactive tests to be done. The test script may thus read the

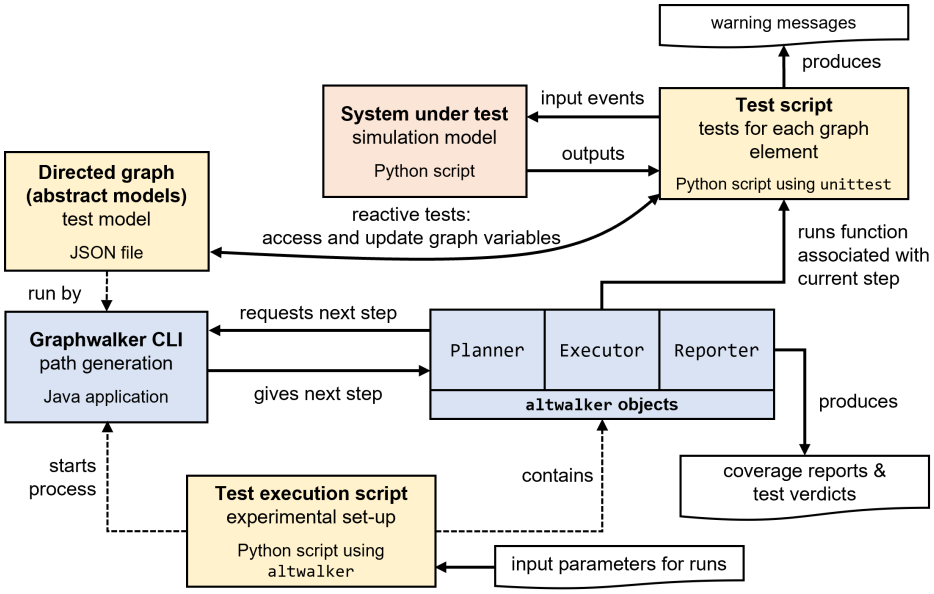


Figure 3.2: Scheme of how all files and programs for an AltWalker test package are used for one online, reactive test case in this project. This can be compared to the general process of a test package given in Figure 2.1

SUT's output, and based on this assign new values to graph variables in the abstract model. Hereby the path generation is influenced by the SUT's output. This also gives a method to verify the abstract model and/or the SUT: the graph variables can be continuously accessed and compared to the SUT's outputs. This can be taken a step further by having shadow variables in the test script for each graph variable.

3.2.5. Automated testing of multiple simulation runs

An important step for the verification of simulation models is to analyze multiple simulation runs. This can be either about simulation runs with different input parameters, or replications, which are defined here as repeated simulation runs that use the same values for the input parameters. This step is mostly related to the goal of verification of a simulation model's results, see Chapter 6, rather than step-by-step verification of the dynamic behavior.

A reason to vary the input parameters for testing is that a fault in a simulation model may only occur for a certain set of input parameters. The simulation model may even show specific behavior patterns only for certain inputs, so full coverage of (sensible) inputs is needed for full coverage of the model's potential behavior. Replications with the same input parameters must be done to test if the model's stochasticity does not lead to faults or unexpected behavior, or if the influence of stochasticity is as expected.

It is found that AltWalker's documentation does not mention specific methods to run a sequence of test cases or to pass initial parameter values to a graph and/or SUT. The AltWalker API is therefore used to add this functionality to the test execution script. This can be done in several ways; different approaches are discussed in Appendix A.3.2.

The approach used throughout this project is that an experimental set-up is defined in the test execution script. Each experiment consists of a combination of input parameter values, and multiple replications per experiment can be run. These input parameter values are passed via the abstract model (using `data`) to the test script. The test script in turn can pass these to the SUT. Unfortunately, direct communication between the test execution script and the test script is not possible in AltWalker. Along with input parameters, a simulation end time and seed for the SUT can be given. Each AltWalker test case run only tests one simulation run of the SUT. A new GraphWalker instance is therefore set up by AltWalker for each experiment or replication.

A bug has been found with the communication between AltWalker and GraphWalker. This problem causes the computer system to run out of TCP/IP ports, which means that no more tests can be executed. This bug and potential solutions are further discussed in Appendix A.3.3. The bug became

apparent in this project, because step-by-step testing of (multiple) simulation runs requires large paths to be generated, meaning that much communication is needed between AltWalker and GraphWalker. No true solution has been found. The bug is partly mitigated by periodically halting all test execution for two minutes, so that the operating system will automatically reset all ports. This naturally gives long execution times, and the bug may still occur if the computer system uses ports for something else. Therefore, this bug is a serious practical issue for using AltWalker for any tests with long paths.

3.3. Conclusions for usage of MBT software tool

Six features missing in AltWalker have been found, that would be helpful for doing MBT for simulation models:

1. Stopping conditions based on a value of variables of the SUT and/or abstract model. Alternatively, continuous checking of specific guards. This would allow a test package to run a simulation model until a specified simulation end time, with a more intuitive syntax.

2. A formalized method for making a test execution script. The method used in this project is based only on the AltWaker API documentation and on trial and error. No term is given in the AltWalker documentation for something like a test execution script.

3. A formal method for a test execution script to do batch execution of multiple test cases and associated SUT runs. This should support data coverage criteria, and include a method to pass input parameters to the abstract model and preferably to the SUT as well. Such method has been developed in Chapter 6 using the `data` functionality and GraphWalker API.

4. Guards, weights, and actions on (implicit) edges between submodels. This would help to get the clarity that submodels provide with the flexibility that normal edges provide. This would improve the composability of test models.

5. A live GUI overview of the abstract model being traversed during online testing would help the understanding and troubleshooting of a test package. AltWalker only provides an animated abstract model that is disconnected from the SUT. This is not useful for reactive tests, where the SUT's output is essential to get full coverage of the abstract model.

6. Lastly, concurrent states of different submodels could be useful in some cases. However, this could quickly get too complicated for simulation models, because multiple states in different processes of the simulation model can be expected to change if time is advanced.

The first and second missing features could be resolved more easily if direct communication between the test execution script and test script were possible.

4

Test packages for case studies

This project uses case studies as the main method to further explore the options for MBT that have been selected in Section 2.3. These are mainly options that are deemed relevant for testing simulation models, but that have not been described in that context in existing literature or case studies. Exploration of these options in the case studies is aimed to answer subquestions 2, 3, and 4.

Each case study consists of a simulation model as the SUT, along with an AltWalker test package developed with a specific purpose. The results of developing test packages for these case studies will be a number of considerations and problems that one may encounter during MBT test development for simulation models. These considerations are given in Chapters 5 and 6. Additionally, eventual problems with the MBT software tool AltWalker come to light.

Furthermore, the simulation models that are used as SUTs utilize different modeling paradigms, namely agent-based modeling (ABM) and discrete event modeling. By considering these two paradigms, it may be found that certain modeling approaches in simulation models may require different approaches in the MBT test packages that are developed for them.

This chapter first gives a short overview of the test packages that have been developed for the case studies in Section 4.1. Then, an overview is given of the three simulation models that are used as SUTs in Section 4.2. The modeling paradigm, functioning of the model, and model results and options for analysis are given. The code for the simulation models and test packages is available online at https://github.com/montequercus/MBT-sim

4.1. Overview of developed test packages

Each case study is meant to focus on one or multiple of the subquestions, as given in Section 1.3, and on some specific features of AltWalker. The first test package is developed for a simple SUT. With the experience from developing this package, further test packages are made increasingly complex with regard to the SUT, test goals, and the AltWalker features that are used. An overview is now given of the three test packages that are developed in this project, see Table 4.1. These are in increasing order of complexity.

Table 4.1: Test packages developed in this project

	System under test	Explored topics	Focus on feature of test package	Sub-questions
A	*Two-way switch*	Verification of dynamic behavior	Online testing, graph data manipulation	2
B	*M/M/1 queue*	Verification of dynamic behavior, verification of results, batch runs	Test execution scripts, graph data manipulation	2, 4
C	*Airport check-in*	Verification of dynamic behavior, implementation-based testing, modular test models	Modularity, communication with SUT	2, 3

Test package A is developed to answer subquestion 2: test model design for step-by-step verification of a simulation model's dynamic behavior. This is done by developing an AltWalker package for online testing. The test is made reactive: the SUT's outputs are continuously used for path generation. This is achieved by incorporating AltWalker's `data` functionality to access and update graph variables in the abstract model from the test script.

The same concepts are explored again in test package B, but now for a discrete event model, *M/M/1 queue*. In addition, subquestion 4 is explored: verification of a simulation model's results. This can be done for this SUT, because analytical solutions are available for *M/M/1 queue*. It is given in Chapter 6 how for this purpose functionality is added to AltWalker for defining multiple test runs and replications.

Lastly, test package C is made for a more complex SUT, *Airport*, that consists of multiple components. Therefore, a more modular setup is developed where the test package uses multiple abstract models; one for each component in the SUT. This is done to answer subquestion 3: specification-based test design for MBT. It is in line with a black-box approach to functional testing. In this context, this means that the test model should only interact with the SUT's interface, and not with its internal states. An effort is made as well to develop this test based on the functional requirements only.

4.2. Overview of simulation models used as SUT

The three aforementioned simulation models that are used as SUTs are now further explained. For each model, the following is explained: the fundamentals of the modeling paradigm used, the problem that is modeled, how it is implemented as a simulation, and what the simulation's results are.

4.2.1. *Two-way switch* model

The first simulation model is called *Two-way switch*. It is a simple model of a two-way light switch. It uses the modeling paradigm of ABM and it is implemented in Python using the Mesa package. ABM can be used to make models with complex behavior, but this case study only uses a rudimentary ABM.

As mentioned before, the development of the associated test package will mainly be done to answer subquestion 2; test model design for step-by-step verification. It is established in Section 2.3.1 that time advancement is an important topic for test model specification, in the context of testing simulation models. It must be explored what it means if time is advanced within the simulation model, and how this could be done from the test model. Furthermore, it is hypothesized in Section 2.3.1 that the modeling paradigm of the simulation model that is used as the SUT may influence the test model design. For these reasons, a brief summary is now given for *Two-way switch* on the fundamentals of ABM and Mesa, and on how time advancement works in this context.

Fundamentals of agent-based modeling and Mesa

ABM is a paradigm in which the model is composed of autonomous entities, called agents, that exist in an environment. An agent-based model describes for each agent its properties and behavior. An agent's behavior can be that they perform some action or change their own properties, but they can also interact with other agents or their environment (C. M. Banks, 2010, p.18). For example, an agent may trigger an action from another agent. An important concept within ABM is emergence: the idea is that the system's behavior emerges from the behaviors of different agents and their interactions. This makes ABM suitable for modeling complex systems: a few interacting agents with simple individual behavior may well lead to complex emergent behavior (Wilensky & Rand, 2015, p.6). The Python package Mesa is used to write the agent-based simulation models in this project. Mesa is object-oriented: different types of agents are described with `Agent` classes. A `Model` class must be made as well, which essentially serves as the agent's environment. The model contains a `scheduler`, to which instances of the agent classes can be added (Kazil et al., 2020).

Like many other ABM packages, Mesa uses discrete time steps of a fixed length. The modeler must therefore think of what simulated time is represented by one time step. In the *Two-way switch* model, for example, each time advancement represents one second of simulated time. A time step must be sufficiently small as to accurately represent the time scales of processes in the modeled system. This is an important difference between ABM and discrete event simulation (DES), which is used in another case study, see Section 4.2.2. Where the time steps have a fixed length in ABM, the simulation time may take on any value in DES.

Table 4.2: Possible inputs and outputs for *Two-way switch*

inputs		output
switch a	switch b	light
0	0	0
0	1	1
1	0	1
1	1	0

Time advancement is done in Mesa with the `step` function. Both the `Model` class and `Agent` classes have a `step` function, which gives the instructions that get executed for each time advancement. The model's `step` function activates the scheduler, which in turn triggers the `step` functions from each agent to be executed in a certain activation order. This order can often have a notable effect on the simulation's results (Kazil et al., 2020, p.311; Sokolowski and Banks, 2010, p.63). For example, if 'agent 0' always gets to act first after a time advancement, this could turn out as a preferential treatment for this agent. For this reason, Mesa offers a random scheduler, which will randomize the agent activation order for time advancement (Kazil et al., 2020). This is an important source of stochasticity in the model. The seed for the random number generator that is used for this activation order and for other stochastic processes can be controlled in Mesa. This ensures that the simulation runs are replicable. A modeler can do multiple simulation runs with the same input parameters but with different seeds; these similar runs are called replications in this project. The outcomes should be different for each replication because of the model's stochasticity. For the analysis of the results of an agent-based simulation model, it is therefore essential to consider multiple replications and the variance in their results (Wilensky & Rand, 2015, p.130).

It should be stressed that because ABM advances time in fixed increments, multiple events may happen during one time step. This is problematic: if the activation order is not considered, it may appear that these events happen at the same time, while the actual, hidden order of events still influences the outcomes (Sokolowski & Banks, 2010, p.75). In ABM software, agent variables and model variables are often only updated and/or saved after the completion of a time advancement (a `step` function in Mesa). This means that each time step can be seen as a black-box model, with the previous states as the inputs, and the new states as outputs. Verification of the dynamic behavior may therefore be difficult if the modeler does not know what events, or state transitions, have taken place during a step and in which order they did.

Modeled system

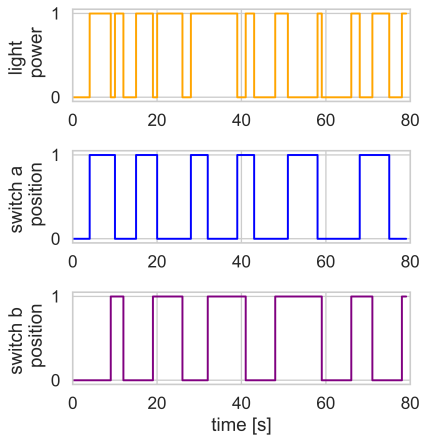
As explained before, only a rudimentary ABM model is used for this case study. The modeled system is a two-way switch, also known as a staircase switch. This refers to a commonly known electrical network, in which a light is controlled by two or more switches. The switches are wired such that, at any time, the power to the light can be controlled by any one of the switches. A network with two switches and one light is used in this case.

This network can be represented as a system with two inputs and one output. The inputs are the positions of switches a and b, which are either in a on (1) or off (0) position. The output is the power state of the light, which is also either on (1) or off (0). Table 4.2 gives the combinations of inputs and output that are possible. This is essentially the truth table of an Exclusive OR (XOR) gate.

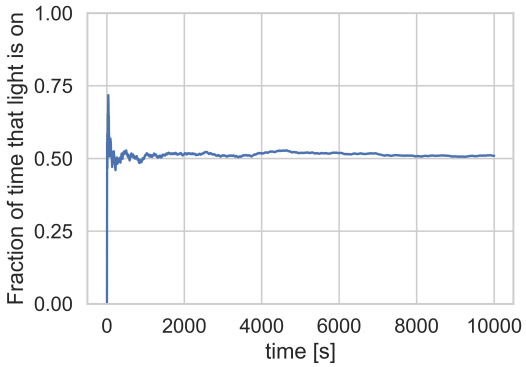
Because time advancement of the SUT should be explored, the simulation model should represent a system that has some time aspect. This is achieved by adding a requirement: both switches should toggle their position independently at random times taken from an interval between 2 and 10 seconds. An analytical solution can be hypothesized easily for the system's only output, the power state of the light. Both switches will be in either position for 50% of the time. This means that the system is in any of the four states from Table 4.2 for 25% of the time. Thus, the model's steady-state solution is that the light should be on 50% of the time.

Simulation model implementation and results

The *Two-way switch* model is implemented in Mesa with two agent classes: a `Light` class, and a `Switch` class. Two instances of the switch class are initialized in the model. The light has only one



(a) Switch and light agent states over time



(b) Counting variable of light over time

Figure 4.1: Results from one run of *Two-way switch* model

property, the Boolean `power`, which shows if the light is either on or off. Each switch has two properties:

1. `position`: Boolean that gives the position of the switch, either down (0) or up (1)
2. `counter`: Integer to count down until the switch's position is toggled. The counter value is sampled from a uniform distribution between 2 and 10 seconds. This value is decreased by 1 for every time step until it is 0. Once `counter` is 0, the switch's `position` is toggled, and it instructs the light to toggle its `power` too.

The results of the model are shown in Figure 4.1 to illustrate. Clearly, the analytical steady-state solution is accurately simulated: the light is on for 50% of the time.

The model results thus give the XOR logic that is typical for a two-way light switch, while this logic is not explicitly modeled. This is one of the properties that are typical for ABM that this model represents. Another property is that multiple events can happen during one time step. For example, it is possible that both switches toggle at the same time.

It must be noted that many other properties typical for ABM cannot be demonstrated with this simple model. Herd et al. (2014) list some characteristics that make verification & validation difficult for ABM specifically, such as the output variance due to stochasticity, the large set of possible input parameter values, and the various macro-behaviors that could occur. Such properties have not been included, as they were deemed less relevant for developing the first test package as a proof of concept.

4.2.2. *M/M/1 queue* model

The second simulation model considered is an *M/M/1 queue* model. This is a common example of a simple discrete event simulation (DES) model. It is relevant to consider DES after the previous ABM example, because the modeling paradigm of DES has a fundamentally different approach to time advancement (Sokolowski & Banks, 2010, p.75). The test packages to be developed for this model will again be used to answer subquestion 2: step-by-step verification of the dynamic behavior. Subquestion 4 is now considered as well: verification of a simulation model's results. As mentioned before, an M/M/1 queue problem is suitable for this aspect of verification because analytical solutions are known for most outputs.

Fundamentals of discrete event simulation and Salabim

DES is a simulation modeling paradigm that makes use of next-event time advance, which means that future events are placed on an event list. The time is advanced by any amount until the time of the next scheduled event. Thus, different from ABM, the time steps have no fixed length (Diaz & Behr, 2010). Each time advancement should in principle produce only one event. That is unless two events are scheduled at exactly the same time, in some DES implementation.

An advantage over ABM is that all variables can be updated precisely when an event actually occurs, instead of at pre-set intervals. All variables and statistics are thus always up-to-date. This makes DES

suitable for supply chain problems, which are best expressed in discrete variables and events, instead of continuous variables (Diaz & Behr, 2010). For example, the current state of a server with queue can be expressed by only using integers: how many entities are waiting, and how many are in the server?

This project uses Salabim, a Python package for DES based on the principle of process interaction (Van der Ham, 2023), where the model is decomposed into components, and for each component, a process is modeled that describes what it should do. The processes can interact with other processes, forming a network (J. Banks & Carson, 1986, p.4).

Simple and custom components are made in Salabim by defining a `Component` class, while ready-made component classes exist for common features. Components may have a `process` function that describes their behavior. Future events of all components are put on an event list. If a component has no future events, it is put in a 'waiting' state, until it is prompted by some other component to become active again. (Van der Ham, 2023). An *Environment* class must be initialized as well, in order for the simulation to run.

Time is advanced with the `run` function. This will keep advancing time until a certain stop condition is reached, or until no more events are scheduled. For each time advancement, the simulation time is set to that of the next scheduled event of one process. Some events may have the same timestamp. For instance, most processes will have some initialization events at $t = 0$. In Salabim, one time advancement will then still only execute one event, instead of all events scheduled for that exact time.

The ready-made components that Salabim offers are useful for supply chain problems. These classes have built-in data collection and summary statistics can be easily shown. An example is the `Queue` class, which can be used to let other components enter and leave a queue. It will automatically keep track of the number of entities in queue, and of statistics like the average waiting time.

Modeled system

A queue, here called a server with queue, is a concept from queueing theory. It is a system that entities can arrive and depart from. It can be decomposed into four processes: arrivals, queueing, service, and departures. In the arrival process, a new entity will enter the system at certain times. It is supposed to go through service in the server, which takes some time. The server may have limited capacity. If the server capacity is already in use by other entities, a newly arrived entity will join the queue instead, and wait until a place for service becomes available. The M/M/1 queue is a server with queue that has some specific settings:

- The arrival process has an interarrival time taken from an exponential distribution with mean λ^{-1}. Here, λ is the interarrival rate.
- The service process has a service time taken from another exponential distribution with mean μ^{-1}. Here, μ is the service rate.
- There is one server with capacity $c = 1$. Thus, only one entity can be in service at a time.

Once an entity has completed service, it will depart from the system. Therefore, μ can be interpreted as the departure rate as well. In addition to these settings, the queue will use 'first-in-first-out' (FIFO) as its queue discipline in this case. This means that the first entity to leave the queue, once a place becomes available in the server, is the entity that has been in the queue for the longest time. A schematic of an M/M/1 queue is given in Figure 4.2.

The only input parameters that can be set are thus λ and μ. This case study uses specific settings in most examples: a mean interarrival time of $\lambda^{-1} = 1$s, and a mean service time of $\mu^{-1} = 0.9$s.

Analytical solution from queueing theory

Verification of the simulation model's results, to answer subquestion 4, is feasible because of the analytical solutions that exist for M/M/1 queues. The relevant variables and summary statistics for servers with queues are now discussed, along with their analytical steady-state solution for the M/M/1 case. These analytical solutions can be used in the test packages to verify the simulation model's outcomes. Oracles should be added to the test package that assess, at the end of a test case, whether the outcomes give an accurate approximation of the known solutions.

Firstly, the utilization factor ρ gives the average fraction of time that the server is utilized, i.e. that an entity is in the server. It is given by:

$$\rho = \lambda/\mu \tag{4.1}$$

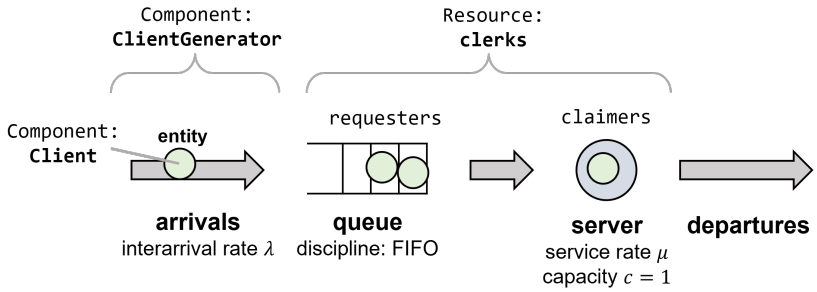


Figure 4.2: M/M/1 queueing system, with processes and their parameters. The object names of the Salabim implementation are shown on top.

A steady-state solution is only defined for a server with queue if $\rho < 1$, thus if $\lambda < \mu$. Otherwise, the system becomes unstable, because on average more entities will arrive instead of depart (Gross & Harris, 1974, chap. 2). This means for the *M/M/1 queue* simulation model, that the exact outcomes can only be verified if input parameters λ and μ are set so that $\rho < 1$. For $\rho > 1$, it could only be verified if the queue length indeed becomes high. The definitions of other relevant outcomes of a server with queue are taken from Gross and Harris (1974):

- L: Mean number of entities in system. It is the expected value of N, the number of entities in system in steady-state.
- L_q: Mean queue length. It is the expected value of N_q, the number of entities in queue in steady-state. This should include times when the queue is empty.
- W: Mean time in system for entities. It is the expected value of T, the time that an entity spend in system in steady-state.
- W_q: Mean waiting time. It is the expected value of T_q, the time that an entity spends in system in steady-state.

The analytical solutions for these outcomes for a M/M/1 queue are given in Equations 4.2 – 4.5 (Gross & Harris, 1974, chap. 2). For the standard input parameters used in this case, $\lambda = 1$ and $\mu = 0.9$, the outcomes are: $\rho_{\text{an}} = 0.9$, $L_{\text{an}} = 9$, $L_{\text{q,an}} = 8.1$, $W_{\text{an}} = 9\text{s}$, and $W_{\text{q,an}} = 8.1\text{s}$.

$$L_{\text{an}} = \frac{\lambda}{\mu - \lambda} \tag{4.2}$$

$$L_{\text{q,an}} = \frac{\lambda^2}{\mu(\mu - \lambda)} \tag{4.3}$$

$$W_{\text{an}} = \frac{1}{\mu - \lambda} \tag{4.4}$$

$$W_{\text{q,an}} = \frac{\lambda}{\mu(\mu - \lambda)} \tag{4.5}$$

Lastly, there are relations between these variables that must hold for a M/M/1 queue. The most important one is Little's law, see Equation 4.6. This is extended to the queue's variables in Equation 4.7.

$$L = \lambda W \tag{4.6}$$

$$L_q = \lambda W_q \tag{4.7}$$

Simulation model implementation and results

The simulation model is adapted from one of Salabim's sample models, *MMc* (Salabim, 2023). The names of the component (classes) used for each process are shown in the M/M/1 schematic in Figure 4.2. The entities are called `Client`, which are simple `Component` classes that try to join the server with the `request` function. The arrival process is implemented with another simple `Component`, namely

`ClientGenerator` which generates a new `Client` at rate λ. For the server and queue, Salabim's built-in `Resource` class is used. This class will put entities that must queue on its `requesters` list, and an entity in service is put on the `claimers` list.

Some relevant variables for a server with queue can be tracked with this class. The outputs ρ, L_q and W_q can be approximated as follows. For the occupancy ρ, the server's utilization state $B(t)$ is tracked for each time step with time t (Diaz & Behr, 2010):

$$B(t) = \begin{cases} 0 & \text{if no entity in server} \\ 1 & \text{if an entity is in server} \end{cases} \tag{4.8}$$

$$\rho(t) = \frac{\int_0^t B(t)\mathrm{d}t}{t} \tag{4.9}$$

The mean queue length L_q is calculated by tracking $N_q(t)$, the number of entities in queue at time t. The time-average until time τ is taken for this:

$$L_q(\tau) = \frac{\int_0^\tau N_q(t)\mathrm{d}t}{t} \tag{4.10}$$

Similarly, the mean waiting time W_q is calculated by saving all $T_{q,i}$, the time in queue for entity i. The average is taken over $N_{\square}(t)$, the total number of entities that have been in the system until time τ (Diaz & Behr, 2010):

$$W_q(\tau) = \frac{\sum_{i=0}^{N_{\mathrm{tot}}(\tau)} T_{q,i}}{N_{\mathrm{tot}}(\tau)} \tag{4.11}$$

To illustrate the outputs of this model, some results for a single run are shown in Figure 4.3. It has been extensively verified, before developing the test package, that the *M/M/1 queue* model's results are good approximations of the analytical solutions. The variables ρ, L_q and W_q ara approximated well when run for a sufficiently large simulation time of $t = 100000$ s. The approximations are better in the input parameter space where ρ_{an} is low. Furthermore, it is found that Little's law (see Equations 4.6–4.7) holds for the model's results as well, even at low simulated times t. An analysis of convergence and sensitivity is given in Section 6.2.2

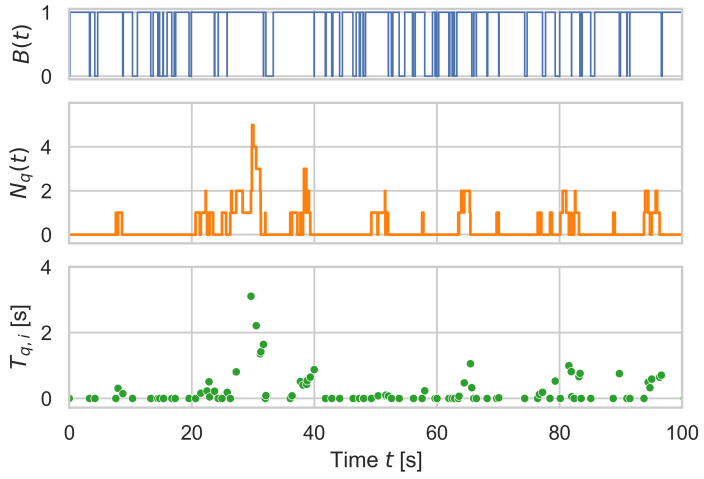


Figure 4.3: Results over time for a single run of the *M/M/1 queue* model, with input parameters $\lambda = 1$, $\mu = 0.5$. The waiting times $T_{q,i}$ are taken for each entity i at the time when they leave the queue. If an entity did not join the queue, then $T_{q,i} = 0$.

4.2.3. *Airport* model

The third simulation model considered is called *Airport*. It is made for a more complex system: an airport check-in consisting of multiple components, with some parallel and conditional components as well. The idea is based on a study exercise where groups of students used the same functional requirements to each make a simulation model using different modeling languages. The ultimate goal of the test package to be developed for *Airport* is therefore to be as agnostic to the SUT implementation as possible. Ideally, it would be able to test implementations of *Airport* in different modeling languages using the same test model. This is in line with subquestion 3: requirement-based testing with minimal communication with the SUT.

To achieve this, the test model should be as abstract as possible so that it can be used with multiple test scripts developed for SUT implementations in specific modeling languages. A common syntax for communication messages between the test script and SUT is made for this. This is further described in Section 5.4.

Five simulation models have been viewed, that were developed for the aforementioned study exercise. An adaption of an ABM for *Airport* made in Mesa has been developed in this project. An ABM is chosen because servers with queues in discrete event models are already considered in the case study of *M/M/1 queue*. Unfortunately, the test package of the *Airport* case study has not been adapted to any of the five student's simulation models due to lack of time. However, the test package is developed to be adaptable: the test models are abstract and modular.

Subquestion 4 can be considered again as well in this case study. The system's processes are similar to the *M/M/1 queue* example, as they are servers. A difference is that there are multiple servers, and that some have unlimited capacity, thus no queue. Furthermore, the arrival processes are unknown and not analytically solvable. Therefore, no analytical solution is available to verify *Airport* and its components in this manner. Still, conformance to Little's law can be analyzed, but this has not been implemented in the test package for *Airport*.

Modeled system

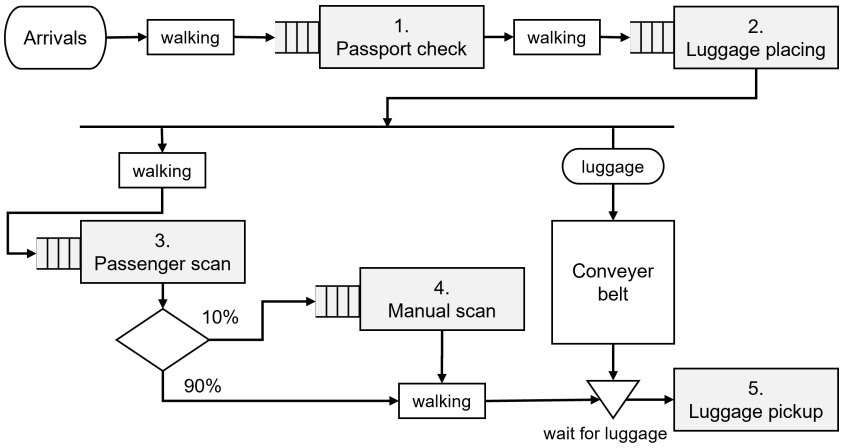


Figure 4.4: Flowchart for passengers and their luggage that traverse the *Airport* system. Components that are modeled as servers are numbered and colored gray.

The functional requirements are taken from the aforementioned study exercise. The system is a passenger check-in of a small airport. A new passenger arrives on average every 60 s. Passengers first go through a passport check, which takes between 30 s to 90 s with a median of 45 s. Then one passenger at a time can place one piece of hand luggage on an accumulating conveyor belt. Luggage drop-off takes between 20 s to 40 s. A piece of luggage traverses the belt of 10 m with a speed of 0.5 m/s. Meanwhile, a passenger goes through a scanner, which takes 30 s to 85 s. A subsequent manual check is needed for 10% of passengers. Only one passenger can be checked manually at a time. After these scans, passengers can pick up their luggage, if it has traversed the conveyor belt. Picking up the luggage takes between 20 s to 40 s. There is a walking distance of 5 meters between all processes and the passengers walk with an average speed of 2.5 km/h.

A flowchart of passengers and their hand luggage that traverse the *Airport* is given in Figure 4.4. This figure already implicates how this system can be implemented in a simulation model: all stages of the check-in can be seen as separate components. Some components can be modeled as servers with infinity capacity, thus without a queue. The manual check and luggage pickup components can be modeled as single-server queues.

Simulation model implementations

The ABM model developed in Mesa is divided into components, or processes, as illustrated in Figure 4.4. A class of Passenger agents is made that are either performing an action at a component or walking in between components. The arrival process is made with a simple agent called `PassengerSource` that creates new passengers at set intervals. The agents for the servers are based on two classes: `ServerWithQueue` and `ServerWithoutQueue`. All server agents will add passengers in service to a list. A passenger that enters service is assigned a service time clock, taken from a distribution that is specified per server. Each time step, this clock is reduced by 1. Once the clock reaches 0, the passenger leaves the server and walks to the next server. Walking is implemented with another clock in the passengers. The `ServerWithQueue` class has an additional list, to which passengers that have to wait are added if the server is utilized. If the passenger in service is done, the server agent will prompt the passenger at the head of the queue to go into service.

The agent classes of specific servers add functionality to their parent classes. `LuggagePlacing` adds instructions for the passengers to wait for their luggage. The conveyor belt is simply implemented by starting a counter of 20 s in the passenger once it finishes the luggage drop-off. The luggage conveyor belt is thus not implemented as a separate agent. Since there is only one piece of luggage per customer, and the conveyor belt is accumulating, this solution suffices. `PassengerScan` is based on `ServerWithQueue` and is extended with routing of the passengers to either the manual scan or the luggage pick-up. The attribute that a passenger needs a manual check is assigned to 10% of passengers at their creation already. `LuggagePickup` is based on `ServerWithoutQueue` and adds the restriction that a passenger can only enter service if their luggage clock has reached zero. Otherwise, the passenger is put on a waiting list.

Stochasticity is added to the model by sampling the interarrival time of the passenger source and the service times of the servers from distributions. Distribution for service times can be given to the servers as an argument. An overview of distributions used is given in Table 4.3. The walking speed is taken per passenger from a distribution as well: uniform on [0.611,0.777] m/s. Further stochasticity is added by a random activation order of the passenger agents. The five servers do a `step` in a fixed order after the passenger agents so that the passenger's updated clocks are taken into account by the servers.

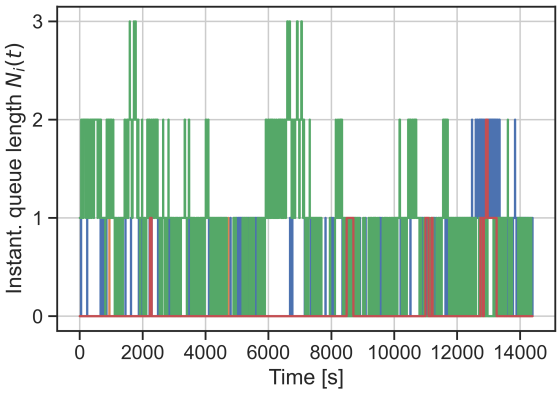
Table 4.3: Components and associated classes and service time distributions for the *Airport* Mesa model

Server i	Component name	Agent class	Service time distribution
	Arrivals	`PassengerSource`	Uniform on [50,70]
1	Passport check	`ServerWithQueue`	Triangular on [30,90] with mode 35
2	Luggage placing	`LuggagePlacing`	Uniform on [20,40]
3	Passenger scan	`PassengerScan`	Uniform on [30,85]
4	Manual check	`ServerWithQueue`	Uniform on [120,300]
5	Luggage pickup	`LuggagePickup`	Uniform on [20,40]

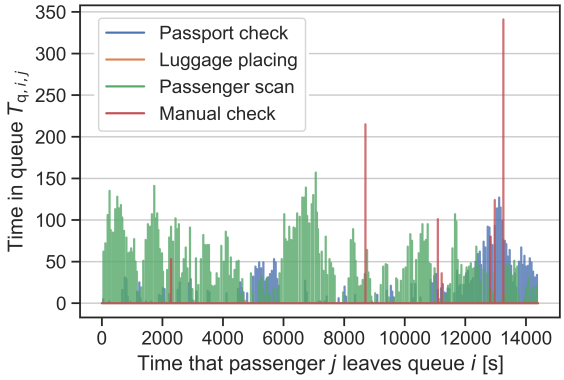
Variables and results

Many variables are collected during each model run. Per server i the instantaneous queue length $N_{\mathrm{q},i}(t)$ is collected. Once a passenger j leaves a queue at server i, its time in queue $T_{\mathrm{q},i,j}$ is collected. These values are shown for one run in Figure 4.5. To compare with other simulation runs, these values are used to calculate the mean $L_{\mathrm{q},i}$ and $W_{\mathrm{q},i}$ per server using Equations 4.10 and 4.11. The mean throughput time is calculated similarly from the time in system $T_{\mathrm{system},i}$ of all N_{tot} passengers:

$$W_{\mathrm{system}}(\tau) = \frac{\sum_{i=0}^{N_{\mathrm{tot}}(\tau)} T_{\mathrm{system},i}}{N_{\mathrm{tot}}(\tau)} \tag{4.12}$$



(a) Instant. queue lengths $N_{\mathrm{q},i}$ per server i over time



(b) Time in queue $T_{\mathrm{q},i,j}$ for passengers j that have left queue i.

Figure 4.5: Results from one run with 5 simulated hours of the *Airport* model, for the four servers with queue.

The model cannot be verified using an analytical solution, but the results from Figure 4.5 can be understood well. Some periods have relatively more arrivals, which gives longer queues at the passport check. There are rarely queues at the luggage placing because luggage placing takes way shorter than the passport check. The passenger scan then has the longest queue times because it has longer service times. The manual check only has queues sometimes since only 10% of passengers go there. However, the time in queue of these passengers can be very long, since the service times are high for the manual check.

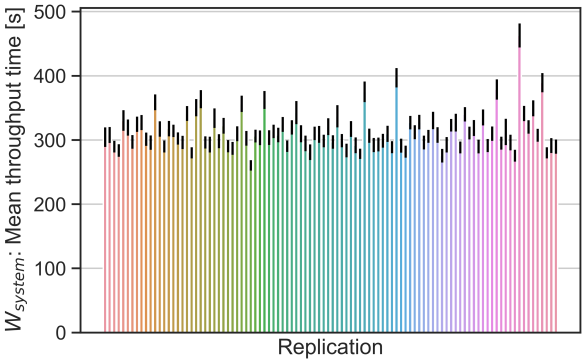


Figure 4.6: Comparison of mean throughput times W_{system} from 100 replications of the *Airport* Mesa model. Black lines give the 10% confidence interval per replication.

Summary statistics L_{q}, W_{q} and mean system throughput time W_{system} must be approximated by running with sufficiently long simulation times. Then still, differences between replications can be found. The range of outcomes that is deemed correct by a test package must therefore be based on a thorough analysis of the results of a verified *Airport* simulation model implementation. This is exemplified with the throughput times W_{system} from 100 replications in Figure 4.6. Clearly, some slight outliers are possible.

Modularity

This implementation in Mesa is made such that all components are modular: a modeler could easily change the distributions of interarrival times per server, add or remove servers, or assign which servers use a queue and what their capacity is. The ultimate goal is that the test package is modular in a similar way, by testing the dynamic behavior components individually. However, the actual simulation results from different runs depend on the interactions between these components and cannot be verified easily. A thorough analysis of the model at certain settings is needed for this, as explained earlier. It seems infeasible to make a test model that simulates the total (expected) system behavior and the interactions of all components: one would essentially be making the SUT simulation model again.

5

Design of test models for dynamic verification

In this chapter, it is explored how the model specification and test generation can be done for each case study. The objective of developing test models is to further explore the dimensions of MBT, which were selected or left open from the literature study in Chapter 2. These choices were based on literature and reasoning; they are now put into practice. The acquired knowledge is used to answer subquestion 2; test model design for step-by-step verification of the dynamic behavior of simulation models. Furthermore, subquestion 3 is researched by developing a test model based on functional requirements only, that uses minimal communication with the SUT. In the process, again some features and limitations of AltWalker, the MBT software tool selected in Chapter 3 will come to light.

The model specification options relevant to MBT of simulation models that have been established in the literature study are now repeated. The test models should be input-output, which means that they should model inputs to the SUT, as well as the SUT's expected behavior. Furthermore, some model characteristics of the SUTs should be present in the test models as well: they should be nondeterministic, and have a discrete time axis. The test models should be transition-based, more specifically they should be extended finite state machines (EFSMs) as that is supported by AltWalker. And lastly, only the functional behavior of the SUT should be modeled.

This chapter is structured around the options for model specification that were left open in the literature study in Chapter 2. The choices made for these options are discussed in Sections 5.2 - 5.4: the selection of representative states of the SUT, event triggering and time advancement, and a more strict black-box approach to testing. First, Section 5.1 gives an overview of how the three developed test models work. This chapter will also discuss in passing the MBT dimensions that are not model specification, namely test generation, test execution, test evaluation, and general test dimensions.

5.1. Overview of test models for case studies

Three test models have been developed, in order to test the SUTs *Two-way switch*, *M/M/1 queue*, and *Airport*. The term 'test model' is used to refer to the combination of an abstract test model and a test script. The abstract model is a directed graph for AltWalker, and the test script is a Python script. A common characteristic of the test models is that they are made to generate one test case in order to test one simulation run of the SUT. This approach is substantiated in Appendix A.3. Due to this approach, the test models do not contain the experimental set-up for batch runs nor for the interpretation of results from multiple simulation runs of the SUT. Instead, a setup for batch runs is implemented in the test execution script. This approach is further explained in Chapter 6.

5.1.1. Test package A: test model for *Two-way switch*

The test model of test package A mostly advances time until an event should occur in the *Two-way switch* SUT. Then, it does some assertions to see if the outputs from these events are as expected. The AltWalker abstract model used for testing is given in Figure 5.1. Three graph variables are used: the current tick t, the maximum number of ticks t_{end}, and a Boolean `bool_counter_zero`. The current

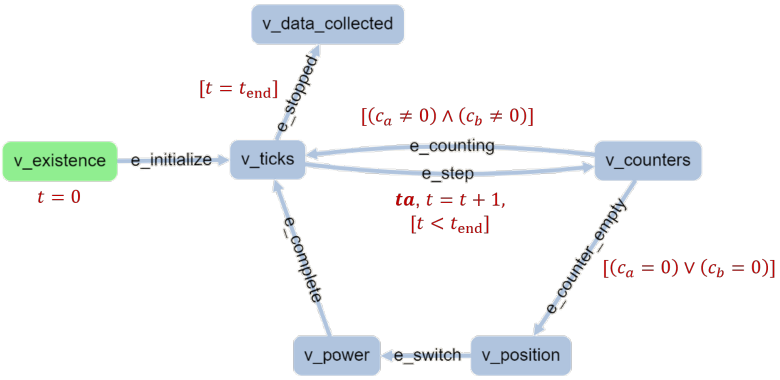


Figure 5.1: EFSM overlayed on AltWalker graph for test model used for *Two-way switch*. Guards and actions are given with dark red text. The time advancement signal is given in bold text.

tick is set as $t = 0$ for every model run. The maximum number of ticks is updated by the test script during the `setUpModel` function. Lastly, `bool_counter_zero` represents whether one of the counters of either switch a or switch b has reached zero in the SUT. These counters can be given as $c_{a,b} \in \mathbb{N}$. This logic is updated by the test script after each time advance, based on the SUT's outputs. Guards on the edge `e_counter_empty` ensure that that edge is only taken if `bool_counter_zero` is true. The test execution is thus reactive: the counters in the SUT's agents influence the path generation.

Once a counter is zero, the vertex `v_position` checks if the positions of switches a and b are as expected. `v_power` checks if the light's power state is as expected given the Exclusive-OR (XOR) logic. From `v_ticks`, the abstract model will only go to `e_step` until $t = t_{end}$. The start vertex `v_existence` is used at the start to check if all of the SUT's objects have been initialized. The end vertex `v_data_collected` checks if the correct data has been collected by Mesa at the end of the simulation run.

5.1.2. Test package B: test model for *M/M/1 queue*

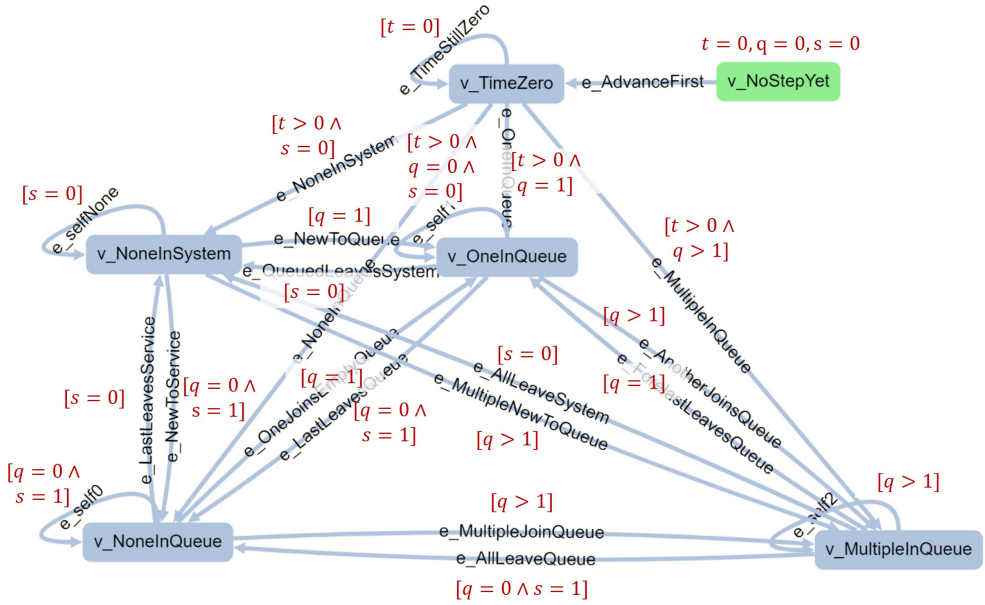


Figure 5.2: EFSM overlayed on AltWalker graph for test model used for *M/M/1 queue*. Guards and action are given with dark red text.

The test model for test package B follows the state of the queue and server in the *M/M/1 queue* SUT. This is achieved with many guards on the edges. This gives the EFSM as given in Figure 5.2, which may be harder to interpret at first sight. Three graph variables are used: the current queue length q, the current server utilization s (number in server), and the time t. Note that the test model's t is a floating-point number because the SUT uses discrete event simulation. The values for these three graph variables are updated by the test script based on the SUT's values after each time advancement of the SUT. As these graph variables are used in the guards, the test execution is reactive here as well.

All state transitions in the test model denote potential events in the SUT, including those that are faulty. In other words, the edges represent what may happen in the SUT due to a time advancement. These events are either arrivals or departures from the server. That is except for the self-transitions in the test model, that denote (faulty) events of the SUT where no arrival or departure would occur. The states of the test model are distinguished by different values of q and s for which distinct assertions would be needed, as is further explained in Section 5.2. A simplified representation of this model is given in Figure 5.5.

The start vertex `v_NoStepYet` and the next vertex `v_TimeZero` are included because the first few time advancements in Salabim give events that all have time $t = 0.000$ s, and where the system's state does not change. These events are used to initialize all processes. Note that there is no end vertex in the test model: a different approach is used to end the test and simulation run at a certain simulated time t via the test script. The inclusion of an end vertex would have made the graph even more complicated. An end vertex would have edges coming from the bottom four vertices with guards $t \geq t_{\text{end}}$. All other state transitions should then have $t < t_{\text{end}}$ added to their guards, to ensure that the model always transitions to the end vertex. Alternatively, a stop condition like `quick_coverage(reached_vertex(v_end))` could be used.

5.1.3. Test package C: test model for components of *Airport*

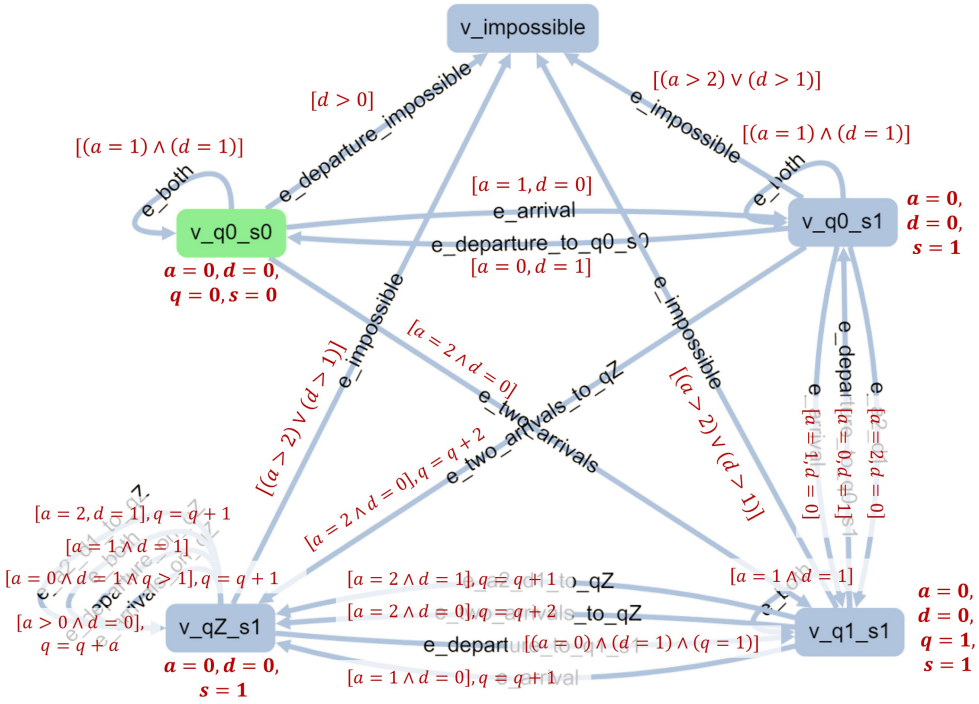


Figure 5.3: EFSM overlayed on AltWalker graph for test model used for *Airport*. Guards and edge action are given with dark red text. Vertex actions are given with bold text.

The test model for test package C is made to test one component of the *Airport* SUT, namely one of its servers with queue. The developed test model for test package C is similar to that of B because the SUTs are similar. In both test models, the states represent a combination of the current server utilization and queue length in the SUT. However, the test model's state transitions are now based on input events

for the SUT component, namely arrivals and departures. This is different from test package B, where the transitions are based on the current values of the SUT's internal variables. The test model for C is thus more a model of the SUT's behavior: the states now represent what state the SUT should be in, rather than what state the SUT is in. This is achieved by modeling the queue length q and server utilization s, instead of updating these values from the SUT. The time variable t also does not have to be copied from the SUT, because the SUT is an agent-based model that uses time steps of fixed length. The time t is now an integer for the same reason.

Additional graph variables are needed to represent the SUT component's input events: a and d give respectively the number of arrivals and departures from a server during one time step. These are updated from the SUT's interface via the test script. A simple communication method between the test script and SUT is implemented so that any SUT based on the functional requirements of *Airport* could pass values for a and d.

A vast number of guards, edge actions, and vertex actions is needed to model the server component based on arrival and departure events only, as can be seen in Figure 5.3. A vertex `v_impossible` can be reached from all other vertices: this is needed for when a faulty combination of input events should occur. The test execution is made reactive as a and d are used in the guards on edges.

All vertices are again associated with oracles, but this is done differently from test package B. Different assertions were needed for each vertex in test package B, to compare the expected values to the SUT's internal variables. This is no longer the case in test package C: each vertex can be associated with the same assertion function, namely that the SUT's behavior should equal the expected behavior that is modeled.

It should further be noted that test packages B and C test different aspects of their SUT, a server with queue. Only test package C takes into account the time constraints. It is checked whether a departure happens within the expected time interval in the SUJT after an arrival or an entry into service, based on the server's service time distribution. For this purpose the time since the last departure is saved in the test script (it is thus not a graph variable).

5.2. Selection of representative states from the SUT

The choices made during the design process of test models are now discussed in more detail. An important choice that one encounters is how an abstract version of the SUT and/or functional requirements should be made and what aspects of the SUT are to be modeled. This section gives the thought process behind this design process. Some dimensions mentioned in the taxonomies in Section 2.3 are relevant here again.

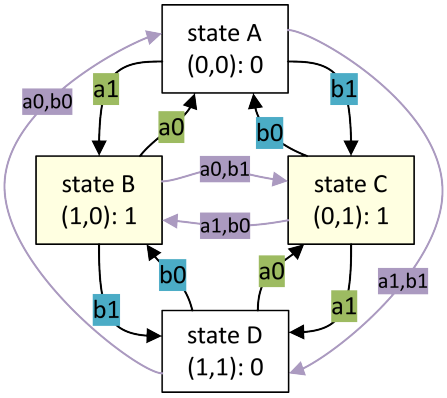
The model paradigm will be transition-based, which means that distinct (system) states and state transitions should be selected, that can abstractly represent the SUT's states. The SUT's functional behavior that is expected given some inputs should be generated in some way by the test model, and this should be done step by step. An input can also be an instruction for a time advance of the SUT. The test model should thus model what events the SUT may or may not generate upon a time advance, given its current state.

It was decided in Section 2.3.5 that the test scope should be either integration testing or system testing. This means that the SUT or a component of the SUT should not be tested in isolation. The inputs and outputs to and from other components of the SUT or its environment should thus be incorporated as well in the test models where relevant. It was further explained why the level of abstraction is key to choosing an MBT approach. It was left undecided how the level of abstraction should be chosen. It must therefore be decided per test package which states and state transitions can represent the SUT in a way that is understandable, finite, and meaningful for testing.

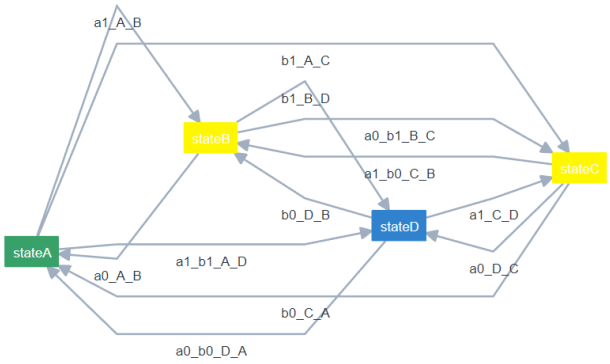
5.2.1. Test package A: state transitions based on SUT's current state and test sequence

The first attempts at making test models in this project focused on 'pure' behavior models: simple finite state machines (FSMs) where all elements mimic (aggregated) states and state transitions of the SUT. That means in essence that the abstract model made in AltWalker is similar to a conceptual model that one would during the development of the SUT. An example of this simple approach is given for the *Two-way switch* SUT in Figure 5.4. The conceptual model is an FSM where each unique set of inputs and outputs forms a system state. The inputs are the states of the switches, and the output is the state

of the light. State transitions occur when one or both inputs change.



(a) FSM of conceptual model



(b) AltWalker graph based on conceptual model. This graph is not useful for testing.

Figure 5.4: Development of a simple test model based on a conceptual model of the *Two-way switch*.

A problem with this approach is that it does not make sense for automated testing. AltWalker associates each vertex, thus a system state in this case, with a number of oracles that are checked on the SUT by the test script. However, this particular SUT is a simulation model that is simple enough that the exact same assertion functions would be sufficient as oracles for testing each of these system states. Each state transition here means only that one of the switches has been toggled. The main assertion needed after a transition is that the XOR logic still applies given the inputs and output. Thus, the distinction between system states as done in this simple FSM, see Figure 5.4, is unnecessary for this SUT. Furthermore, such a test model could not explicitly show how the other functional requirements of this SUT are tested. An example is the timing constraints: how would this FSM show the requirements that the toggle times of switches should be taken from a uniform distribution between 2 and 10 seconds?

Another reason to not use a 'pure' behavior model is a problem known as state explosion. One could imagine a model similar to *Two-way switch*, but with even more switches or a light that cycles through different brightness settings. That would mean more state variables and processes, which would result in more states and exponentially more state transitions. The FSM and abstract model as in Figure 5.4 would then be even more unwieldy. And for the case of *Two-way switch*, all vertices would still only need the same assertion functions, because the model is not complex.

This idea that any distinction between the SUT's system states is not relevant for testing the *Two-way switch* SUT leads to a new test model as given in Figure 5.1. This test model advances time until any event happens in the SUT, and then runs a series of assertions once an event has occurred. This EFSM is thus less of a behavior model of the SUT since it does not (explicitly) convey some important variables of the SUT, such as the positions of the switches. The only graph variables that are kept analogous to the SUT is the time t and `bool_counter_zero`; this is what makes the test reactive. This Boolean `bool_counter_zero` is thus not modeled by the test model, since it is only updated from the SUT's current state. It is used on the guards for edges `e_counting` and `e_counter_empty` to ensure that the test model only runs the relevant assertions once an event *should* have occurred in the SUT. That is when one of the clocks c_a or c_b of the switches reaches 0. Then, in the next time step, a fixed oracle is checked: has the switch's position been toggled, and does the XOR logic still apply?

A shortcoming of this new test model is that it no longer gives the tester a helpful abstract and visual representation of what the SUT's current state is or should be. That aspect is still a reason to consider test models that are more based on the system states, especially for more complex SUTs.

5.2.2. Test package B: state transitions based on SUT's current state

The same initial idea of making a test model based on the conceptual model has been used for the *M/M/1 queue* example. This SUT is a more complex simulation model, therefore more choices have to be made. With the experience from developing a test model for *Two-way switch*, it was decided to look for (system) states of the SUT that would actually necessitate different oracle functions, i.e. different

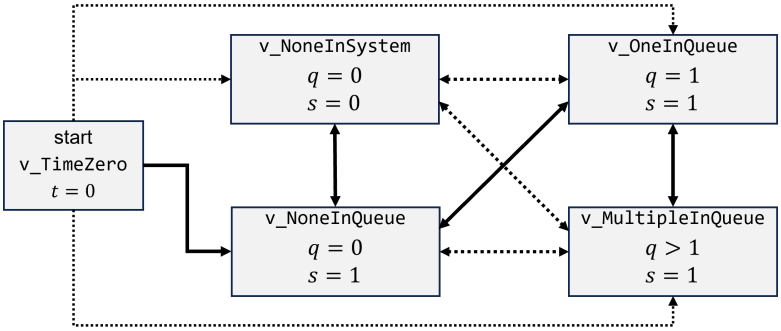


Figure 5.5: Simple representation of the meaning of states in test model for *M/M/1 queue*. Solid lines give state transitions that are permitted. Dotted lines give transitions that are faulty.

assertions. The simulation model's current system state can be represented by a set of variables. The first is the instantaneous queue length $N_q(t)$, called q in the test model. The second is the instantaneous server utilization $B(t)$, called s in the test model.

A simple representation of this conceptual model is given in Figure 5.5. This has been made into an AltWalker test model, which was given in Figure 5.2. While the state transitions are based on arrival and departure events, these events themselves are not used to traverse the test model. Instead, the aforementioned queue length q and utilization s are checked in the SUT by the test script after each time advancement of the SUT. The vertex corresponding to a SUT state is reached in the test model because of the many guards used on the edges. This test model is thus not strictly a behavior model, because it does not simulate what state the SUT should be in. Instead, it follows which state transitions the SUT makes, and does the relevant assertions on the SUT's internal variables for each state. Additionally, it will notice if the SUT makes a faulty transition.

The assertions made in the test script for each vertex are now listed. One assertion is common to vertex: the number of entities in the SUT may not change by more than 1 for each event. This is implemented by keeping a memory (list) of the number of entities $N_q(t) + B(t)$ in the test script. The assertions unique to different vertices are as follows:

- `v_NoneInSystem`: It is checked that the SUT's number in queue $N_{\mathrm{q}}(t)$ and number in service $B(t)$ are indeed 0. This is a method to validate the test model.
- `v_NoneInQueue`: The same thing is checked, but now $B(t) = 1$ should hold.
- `v_OneInQueue`: The same thing is checked, but now $N_q(t) = 1$ and $B(t) = 1$. Furthermore, it is checked whether the time of creation (timestamp) of the entity in service is lower than that of the entity in queue.
- `v_MultipleInQueue`: The same thing is checked, but now $N_q(t) > 1$ and $B(t) = 1$. Furthermore, the timestamps of all entities in queue are now compared to ensure that the first-in-first-out (FIFO) discipline is applied correctly.

In conclusion, the MBT technique of having a test model is only used here to select the assertions that are relevant given the SUT's current state. The assertions contain the expected behavior for all internal variables in that state. This is thus clearly white-box testing. And again, not all functional behavior is covered in this test package. For example, the test model by itself does not test whether the interarrival times and service times follow the correct distributions. It is chosen for test package B that this, among other requirements, is checked by analysis of multiple test cases: see the steady-state results are verified against the analytical solutions that are available for *M/M/1 queue*, see 6.2. Testing of distributions by the test model itself is also possible; this is done in test package C.

5.2.3. Test package C: state transitions based on SUT interface

The test models in test package C only do test on individual components of *Airport*. The test models for test package C are for component of the SUT, which are servers with or without queues. The test model can therefore be similar to the one developed for test package B. Similarities are that both are based on the server utilization s and queue length q. The vertices have a similar meaning to those in test package B as well. The state transitions are now however based on the SUT's input and output events, rather than the SUT's internal variables. The number of arrivals and departures per time step

are the only SUT variables that are used for path generation by the test model. This way, the test model actually models what the SUT should be doing given its inputs over time. The SUT's internal variables such as $N_{\mathrm{q}}(t)$ and $B(t)$ are still used for the assertions, so it is not purely black-box testing.

Design of test model

Figure 5.6 gives a simple representation of the test model. More edges, guards, and actions are needed compared to test package B, because the graph variables q and s must be tracked by the test model itself, instead of being updated from the SUT.

The choices for including particular edges are based on the assumption that two arrivals may happen during one time advance. This ensures that test package is adaptable to different input parameters for the *Airport* model. Simultaneous arrivals are not forbidden by the functional requirements, and they are theoretically possible because of the random walking times between servers. For the SUT's standard settings however, the service time distributions and walking distances are such that this should not happen: the maximum walking time is lower than the lowest minimum service time. The edges with two arrivals therefore give warnings. Two departures from a server will however always give a transition to `v_impossible` (not included in Figure 5.6); this should not be possible in any implementation of this system, and will throw an exception during a test.

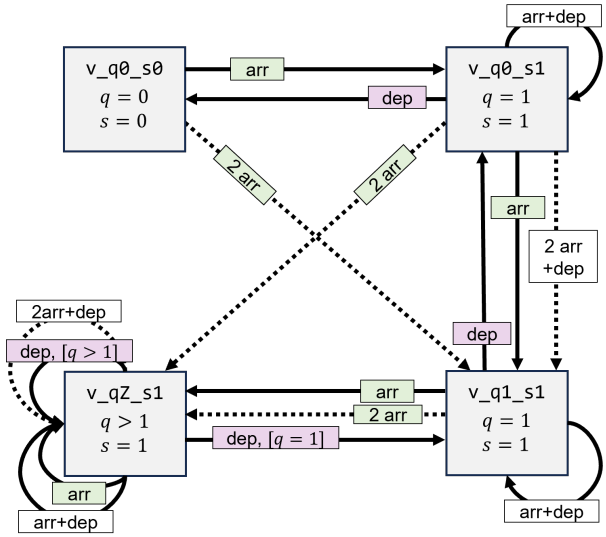


Figure 5.6: Simple representation of meaning of states in test model for *Airport*. Solid lines give state transitions that are permitted. Dotted lines give transitions that are questionable. The events arrivals (arr) and departures (dep) are used as guards on edges. Edges for impossible transitions are left out in this figure.

The assertions associated with all vertices now simply check if the test model's state is equal to the SUT's state, for all representative variables. This means that the SUT's number in queue $N_{\mathrm{q},i}(t)$ should equal the test model's q. Similarly, the SUT's server utilization $B(t)$ and the test model's s are compared. This this means that all vertices can use the same sequence of assertions, which is a slightly different approach as was used in test package B.

Against a validated SUT, these tests will show if the test model's guards and actions have been implemented correctly. The test script also saves a number in queue based on the number of arrivals and departures for even more validation of the test model. Lastly, it is checked that the number of entities in the SUT component has not increased or decreased by more than 2 overall after a time advance.

Testing of time requirements

It is not possible to use multiple test models with concurrent states from one AltWalker run, see Section 3.2.1. For this reason, one server of the *Airport* SUT must be chosen to run a test case on at a time. The test scope is still integration or system testing, so all SUT components are still executed during a test case, in order to incorporate the interactions between components.

Due to these interactions the arrival processes per server are not known in *Airport*. This means that the distributions cannot be tested by validation of steady-state results against known (analytical) solutions as was done for *M/M/1 queue*. Therefore, the correct use of distributions is tested inside the test model itself. Assertions associated with edges are used for this. Each edge with a departure event as a guard ($d = 1$) will assert that this departure happened from within the expected time interval since the last entry into service. This is implemented by starting a clock in the test script each time that a new passengers enters into service.

5.3. Event triggering and time advancement

Another choice that one encounters is how time should be advanced and how events should be triggered in both the test model and SUT during testing. From the taxonomy and case studies in Chapter 2, it was found that no guidelines exist regarding test model specification for MBT of simulation models or similar systems. It must therefore be explored here what event triggering means for simulation models. We can distinguish four factors that will determine a simulation model's behavior during a simulation run. These can be related to endogenous variables, which can be explained within the model itself, and exogenous variables which come from the model's environment.

1. The actual simulation model itself will determine what sequence of events can be generated during a run, given the inputs. Endogenous events are generally created in a simulation model when a time advance function is executed. This can be tested step-by-step, as explained in Section 2.4.1.
2. Exogenous parameters or events from the environment can further influence the model run. Initial input parameters must always be set before a run, and generally influence the entire sequence of events that follows.
3. Optionally, exogenous events, or 'triggers', can be used as further input during a simulation run. When considering components of simulation models, the outputs of other connected components can be seen as exogenous events. An oracle for this could simply specify what counts as a correct response, or within what period a response is expected (see again Section 2.4.1).
4. All stochasticity present in a model will make the output nondeterministic. Therefore, a range of outputs is possible for a given set of input parameters and/or a sequence of external events.

As established in Section 2.2, this reactive nature of simulation models makes dynamic verification difficult. Naturally, it is found to give a problems for the specification of test models as well. Test models are often used to generate exogenous parameters: initial parameters and/or exogenous events, that are used as inputs on the SUT during a test run. The testing of real-time embedded systems is a good example of this: in most case studies discussed in Section 2.4 one test model, called the environment model, is used to generate input parameters and/or events. These inputs are then used by another test model which models the behavior, and they are used as inputs for SUT itself. The expected behavior and the SUT's actual behavior can then be compared. The stochasticity that can be built into the test models is a great advantage for this purpose: it allows a tester to automatically generate many (sequences of) input and/or event triggers.

However, most simulation models used for M&S are different from such real-time embedded systems because they generate events themselves and have inherent stochasticity. Giving event triggers during a simulation run is then not needed; given some inputs, the simulation run (SUT) itself will generate events when time is advanced. This means that the test model does not necessarily have to give exogenous events, and that it should be able to generate the behavior that is expected for the next time advance in the SUT. This last goal is difficult because of the stateful and reactive nature of simulation models.

5.3.1. Classification of options for event triggering

A choice must thus be made on whether the test model triggers specific events in the SUT, or only monitors what a SUT does 'by itself' during a run. Based on this dilemma, a classification is made of how time can be advanced by a test package during functional testing of a simulation run:

Monitoring only The test package only executes the SUT's time advance function. The behavior of the SUT is monitored and compared to the expected behavior, which is generated by a behavior

model. Another test model or (automated) sampling can be used to generate initial parameters for both the behavior model and the SUT.

Mediated run The test package executes the SUT's time advance function, and will occasionally send event triggers to the SUT when some conditions are met. The setup with different test models can be similar to the first option. The difference is that the test model now sends inputs to the SUT, that are not only time advance instructions.

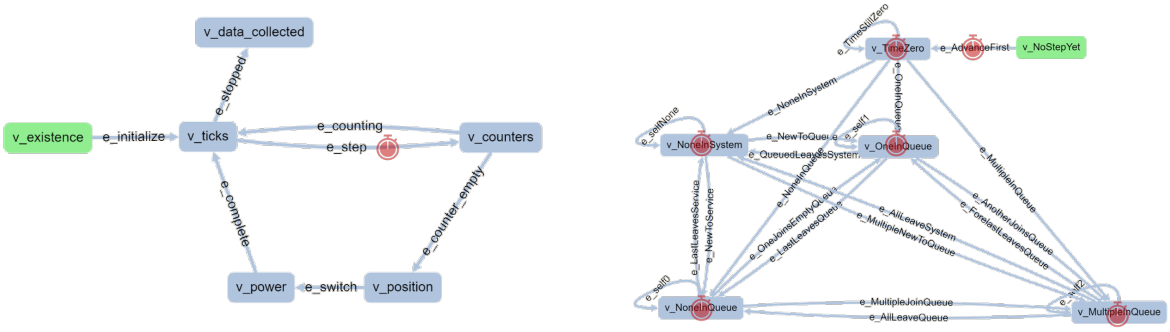
Explorative run The test package sends a predefined or generated sequence of event triggers to the SUT, and executes the time advance function when there are no event triggers. A more complex and nondeterministic test model is needed for this. A goal can be to simulate the SUT's environment during a run or to quickly simulate and test what may occur during a run.

Note that in a monitoring run, the test model only gives two types of inputs to the SUT: initial parameters at the start of a run, and time advance signals during a run. Thus, no exogenous inputs are given during the run. A mediated or explorative run would give exogenous inputs during a simulation run. That is in line with the idea by Mihram (1972), see Section 2.2, to suppress the exogenous stochasticity for dynamic verification. Only test packages with monitoring runs are developed in this project. The other two options may be useful for testing of simulation models as well and are thus recommended for further research.

5.3.2. Implementation of time advancement in a test model

It has been found from the literature study, see 2.3.1, that the test model should have the same notion of time as the SUT, and it should have a method to advance the logical time in the SUT. It has been left open for the case studies how this can be implemented in test models for MBT.

The first problem is which elements of the abstract test model should be associated with a time advance function in the test script. It is found in the case studies that this depends on how the test model is specified. The test model specification of test package A needs a time advance only on one edge, while the specification of test package B needs a time advance on all vertices. Figure 5.7 illustrates which elements of the abstract models are associated with time advance functions. Test package A uses the two edges between `v_end` and `v_counters` to iteratively advance time, then check if the SUT's system state has changed in any way. Thus only one edge in the test model has a time advance function associated with it. Test package B on the other hand uses a vast number of edges with guards to let the test model follow the SUT's system state. The path generation must be based on the SUT's current state, and the path generation will only choose a certain edge based on its guard. Therefore, the time advance must be done here before the guards are checked. This is achieved by having the time advance on vertices: the functions in the test script that are associated with each vertex have the SUT instruction `MM1.env.step()` after all assertions are made. After this instruction, the graph variables are updated based on the SUT's new variables so that these can be considered for the guards.



(a) Test package A: *Two-way switch*

(b) Test package B: *M/M/1 queue*

Figure 5.7: Elements in test models where a time advance signal is sent to the SUT by the associated function in the test script. The time advance signal is indicated with a clock.

The approach for test package B is counter-intuitive to how EFSMs and AltWalker's implementation of them are supposed to work: it seems like state transitions now happen on the vertices instead of

the edges. That is only partly true: the vertices will cause a state transition in the SUT, but the state transitions of the test model itself still happen on the edges. This distinction must be kept in mind when this approach is used for designing a test package. The main advantage of this approach is that the abstract test model (graph) needs fewer elements and will be easier to understand once it has been designed.

The design choice of which elements to associate with time advancement functions did not depend on whether the SUT was an ABM or discrete event model in this project. Test package C is used to test an ABM model, and it uses time advancements on vertices similarly to how test package B does this for a DES model. However, a time advance here may not necessarily lead to a significant event, because the test model only considers one component of the SUT, and because ABM is used. Therefore the test script will advance time after running all assertions of a vertex. This is done until a message arrives from the SUT about an arrival or departure. The test model may thus stay in one state with no new path being generated, while the SUT does multiple time advances. A maximum number of time advances for waiting until a new message is implemented to prevent an infinite loop during test execution. This does mean that conditions for path generation are now present in both the abstract model and the test script, which makes the test package more difficult to maintain.

In short, three subtly different methods for advancing time with a test script have been used:

1. Explicit time advance with an edge. This makes the abstract model of test package A easier to understand.
2. Implicit time advance on vertices, after running all assertions. This is more suitable for an abstract model with some complexity like in test package B, to reduce the number of elements needed.
3. Multiple implicit time advances on vertices, until a condition is met. This is useful for complex abstract models where time advances do not necessarily lead to events that are significant for testing, like in test package C.

5.3.3. Communication between test model and SUT

A common problem in the development of MBT packages is the communication between the SUT and the test package. This is not part of any of the MBT taxonomies, but the concepts relevant to this communication have been explained in Section 2.1.2. Abstract test cases must be made executable by mapping them to instructions for the SUT. This is done in this project by associating oracles, i.e. assertion functions, with vertices and edges in the test models. These assertions must access the SUT's outputs or internal variables, so the test package must support communication with the SUT. Adapter code may be needed for this.

Note that this problem of communication is not fully considered in this project. All test packages of the case studies simply import the simulation model as a module and execute functions on it directly. Thus no adapter code for communication is truly made. Where it is stated in this report that a time advance signal is sent from the test script to the SUT, this actually means that the SUT's time advance function is run within the test script. No statements can therefore be made on how to solve problems of communication with external SUTs and the associated timing issues.

Test package C gives a first attempt at having minimal and formalized communication between the test package and the SUT using messages. This is further discussed in Section 5.4.2. Similar approaches could be useful for communication when the test package and SUT are actually different programs that are executed separately.

5.4. Requirement-based test design and modular test models

It is hypothesized in Section 2.3.5 that MBT could be an approach to white-box testing, whereas model-based tests are usually seen as black-box tests. It is concluded that many examples of MBT are not strictly black-box tests, in the sense that internal variables of the SUT are accessed during testing for either oracles or even to influence path generation. However, a general rule exists in literature that MBT packages should be developed based on the functional requirements, and not on the SUT implementation. This is a requirement-based test design, where there is minimal communication with the SUT, as is formulated for subquestion 3. Special attention is given to this topic during the development of test package C. It is now evaluated whether this approach is implemented in any of the developed test packages.

5.4.1. Strictness of requirement-based test design

The rule that the model specification should be done based only on the functional requirements is not strictly followed in this project. This can be illustrated by the distinction between how time is advanced between test packages A and B. That is already based on how respectively the *Two-way switch* and *M/M/1 queue* SUTs have been implemented, see Section 5.3. Not adhering to this rule is not necessarily a bad thing. Considerations like when to advance time are simply needed to create a test model that can function with the given SUT.

For another example: test package B even adds the vertices `v_NoStepYet` and `v_TimeZero` to account for a characteristic of the SUT's simulation package Salabim: processes must first be initialized. These could have been left out of the test model by including them in the test script's `setUpRun` fixture. That would have made the test model more agnostic to the SUT's implementation.

For test generation and execution, all test packages use assertions that are done on internal variables of their respective SUT's. Test package C is a good example: the path generation and test model behavior is solely based on the SUT's interface, but still internal variables from the SUT are accessed by the test script for the oracles. That means that the test cases are abstract at first, and are then made executable. That is within the definition of black-box testing for MBT. It could have been made even 'more black-box' by only asserting that departures happen when expected: only the SUT's output would then be tested, but the test package becomes less useful in return for understanding what faults the SUT may have.

In short, for the test model specification of test packages A and B the SUT implementation was definitely considered. This was only not done for test package C. Test package C only assumes that the SUT consists of similar components, that can be tested separately.

5.4.2. Minimal communication between test package and SUT

Test package C demonstrates how minimal communication between the test model and SUT can give a test package that is based more on the SUT's requirements, rather than its implementation. Communication between the SUT and the test script for path generation is done here with messages. The *Airport* SUT will output strings that describe how many arrivals and departures occurred at each component during the last time step. These strings are formatted as either `'arrival'` or `'departure'`, along with the associated server's name. The test script will check for these messages, after each time advance signal that it sends to the SUT. The messages are filtered for the name of the component that the test model is currently testing, and they are counted to get the number of arrivals a and number of departures d for that time step.

This simple method of communication also enables a modular setup, where test package C can be used to test similar components within the *Airport* SUT. Furthermore, this method could enable test package C to be more agnostic to the SUT implementation. It could ideally be used with other SUTs that model the same system. The modeler then only has to include an interface that denotes arrivals and departures per time advance in this specific format. Testing of similar SUTs has not been done in this project.

5.4.3. Specification of oracles

The test models mainly test the dynamic behavior of the SUTs: each time advancement is a step in the test case, and an oracle is used on each step. Up-to-date values of the SUT's internal variables are needed for this step-by-step approach.

Alternatives to a step-by-step approach can also be named a 'more black-box' approach to testing, particularly to oracle checking. This can be related to the three approaches to testing defined in Section 2.4.2. A first alternative is that the SUT's state is not assessed after every time advancement, but that its eventual correct response to a given input is only tested. Another alternative is that the test package only assesses the correctness of a simulation run's output or aggregate of outputs, given its initial values. For testing simulation models, this should even be done for multiple runs, because the SUT then has inherent stochasticity. This is the topic of subquestion 4. As mentioned before, batch runs are implemented outside of the test model: each test model run is used to test only one SUT simulation run. This implementation of batch runs is discussed in the next chapter.

6

Automated tests of multiple simulation runs

This chapter aims to answer subquestion 4: the development of automated test packages that use or integrate MBT techniques to verify the results of a simulation model. It has been discussed that verification of a simulation model's outputs necessitates that multiple runs are considered because of the inherent stochasticity of simulation models, see Section 2.2. Some batch-run functionality of test cases is thus needed. Such functionality is missing from AltWalker, see Section 3.2.5. A choice has been made to make test models that only consider one SUT simulation run at a time. Additional functionality must thus be developed to enable batch runs.

It is first shown in Section 6.1 how a functionality for batch run has been added to test package B. Section 6.2 shows what numerical tests can be used for verification of (time series) results, and what analysis of a simulation model is needed to define the expected results used that will be used in the tests for *M/M/1 queue*. Lastly, Section 6.3 shows how numerical tests of multiple simulation runs have been implemented in test package B

6.1. Test package design for analysis of multiple simulation runs

An experimental setup must be defined to get a test suite that can reach a certain testing goal. The experimental setup includes the number of experiments, the input values for each experiment, the number of replication runs per experiment, and the simulation end time. In this project, a setup is chosen where each experiment tests one simulation run, using one AltWalker run. The reasons for choosing this approach and details on its implementation are discussed in Section 3.2.5. The experimental setup is defined in the test execution script, so a batch run functionality must be implemented here.

Test package B, the test of a *M/M/1 queue* discrete event model, has been developed to answer subquestion 4 specifically because analytical solutions are known for this problem. A version of test package B is now considered, that runs many replications for a given set of input parameters.

6.1.1. Experimental setup via test execution script

The experimental setup for test package B is simple: the SUT only uses two input parameters. These are the `iat` (interarrival time) and server time. The test package further has three parameters itself: the number of experiments with distinct input parameters, the number of replications to be run and tested per experiment, and the simulation end time t_{end} for each SUT run.

The simulation end time is now implemented by adding a vertex `v_FailOrEnd` to the abstract model. This vertex can be reached from all other vertices, with the guard `t > t_end`. These transitions are guaranteed to be taken because a requirement is added to the vertex, and the model is run with a requirement coverage condition. This new vertex and associated edges were not shown in Figure 5.2.

Two options are mentioned in Appendix A.3.2 for passing initial values from the test execution script to the AltWalker abstract model: directly via the `Planner` and abstract model, or with a parameters JSON file that is loaded by the test script. Both options have been found feasible while developing the test packages. Packages A and B use the first option, while package C uses the latter.

The current implementation will sample values for input parameters uniformly from a range that is specified in the experimental setup. Of course, this can be extended with more complex sampling techniques.

6.2. Verification of simulation model results to analytical solutions

The outputs of single or multiple experiments can be verified if the expected outputs for certain initial parameters are specified in the functional requirements. Options are now given for how to implement verification for test package B, because analytical solutions are known for the SUT *M/M/1 queue*.

To define in the requirements and in the test package what outputs of *M/M/1 queue* are allowed, an analysis of the results of this simulation model is first needed. This analysis has been done using three experiments, to establish the stochasticity, convergence, and sensitivity of this simulation model. This is discussed in Section 6.2.2. The *M/M/1 queue* simulation model is assumed to be valid. The results of these experiments are therefore used to inform what numerical tests can be implemented in test package B. The choice of numerical tests is now first discussed.

6.2.1. Options for numerical methods for verification

An applicable numerical method must be chosen to define in a test package what range of outputs is deemed expected or correct, given some inputs. This must be a range because a simulation model with inherent stochasticity will give different outputs for each replication, and the outputs will never exactly equal the (analytical) solution. It is useful to consider only aggregate results from simulation runs, such as the mean and standard deviation of a variable, instead of the full time-series data.

A numerical method can be used during test development to analyze the aggregate results of a validated SUT. The resulting ranges of outputs can then be used as the expected behavior in oracles in the test package. These oracles can likely be specified in the test model as the same numerical functions that were used for analysis, possibly using the same numerical methods. Two numerical methods are now discussed: statistical tests, and more simple output ranges.

Use of statistical tests

Statistical tests are an obvious choice for a numerical method that can consider a range of outputs. The student's t-test is commonly used to compare the mean and standard deviation of a sampled population to a theoretical mean. This is useful for problems like the *M/M/1 queue* for which analytical steady-state solutions exist for some outcomes; these solutions are the theoretical means.

For the *M/M/1 queue*, some important outcomes are the steady-state L, W and utilization ρ found with long simulation times. The sample is then a set of these outcomes of many replications. These can be compared to the analytical means L_{an}, W_{an} and ρ_{an} calculated from Equations 4.1, 4.2, and 4.4. A condition is that the system should be stable, meaning that input parameters λ and μ should be such that $\rho_{\text{an}} = \lambda/\mu \leq 1$.

There are some problems with using the student's t-test for a model like *M/M/1 queue*. The student's t-test is based on the assumption that the sample follows a normal distribution. It can be assumed that the samples from many replications have a normal distribution for L_q, W_q, and ρ. This is found to only be the case for ρ. This can be seen on face value in Figure 6.1.

Another problem is that the t-test becomes stricter if it considers more samples because the number of degrees of freedom increases. So even if the mean of means is very close to the analytical mean, the t-test may still indicate that it is not significantly proven that the sampled means give the correct mean. This can be clearly seen from 1000 runs of the *M/M/1 queue* model with input $\rho_{\text{an}} = 0.9$ in Figure 6.1. The mean of means $\bar{\rho}$ over 1000 replications is 0.89904. Thus, with many runs the steady-steady behavior of this simulation model is very close to the analytical solution on face value. However, a t-test of these 1000 ρ values against ρ_{an} will give a p-value of only 0.0158. That would mean that there is no significant proof that this close approximation was not only due to coincidence. The same problem occurs when time series results from one simulation run are used in a t-test: these are generally many data points, resulting in an unnecessarily strict test.

Statistical tests like the student's t-test are therefore not used for analysis of *M/M/1 queue*. Developers that want to incorporate such tests into automated test packages are advised to choose the α value such that type 2 errors are avoided, while some type 1 errors may occur. Type 1 errors can be seen as false negatives. The automated test package can show these to a manual tester, who can further check if the results are indeed wrong.

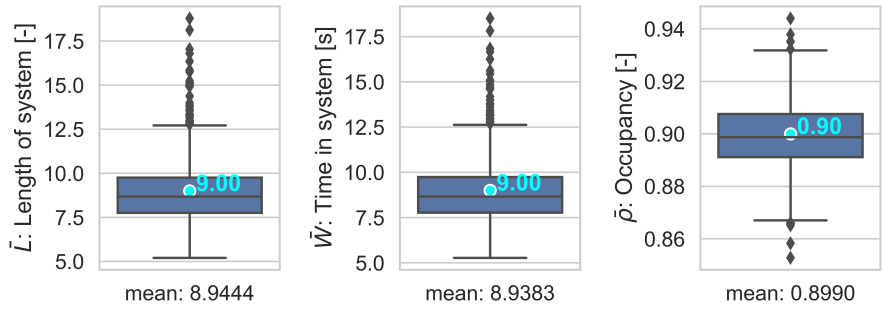


Figure 6.1: Distributions of mean outputs from 1000 replications of *M/M/1 queue* with $t_{\text{end}} = 10000$ s. Analytical solutions are given in light blue.

Simple numerical methods for acceptable output ranges

A more simple numerical method is therefore used to analyze and later test *M/M/1 queue*: a bandwidth of acceptable results is defined. For instance, a tester can define a relative deviation of $d = 0.5\%$ from the analytical mean $\bar{x}$ as acceptable. This gives lower bound $x_{\text{lower}} = \bar{x} \cdot (1 - d)$ and upper bound $\bar{x}_{\text{upper}} = x \cdot (1 + d)$.

Such analysis is done on the results of *M/M/1 queue* with $\rho_{\text{an}} = 0.9$ for 1000 replications, with a simulation end time of $t_{\text{end}} = 10\,000$ s. It is found that ρ is within the bounds for 28.3% of runs if $d = 0.5\%$. When $d = 1.0\%$, this increases to 54.2% of runs with $\rho \in [0.891, 0.909]$.

6.2.2. Analysis of *M/M/1 queue* results and decisions for test package

Three experiments are done with *M/M/1 queue* to analyze the results and sensitivity of this model. The results from these experiments are used to inform the development of the test package in Section 6.2.3.

Stochasticity of *M/M/1 queue*

The experimental setup to analyze the influence of stochasticity in *M/M/1 queue* is to run 1000 replications of the model with input parameters $\lambda = 0.9\,\text{s}^{-1}$ and $\mu = 1.0\,\text{s}^{-1}$. This is done with $t_{\text{end}} = 10\,000$ s. This should give a utilization $\rho_{\text{an}} = \lambda/\mu = 0.9$. This value is chosen, because it is close to the unstable system $\rho_{\text{an}} > 1$; a broader distribution of the outputs can be expected for higher ρ_{an}.

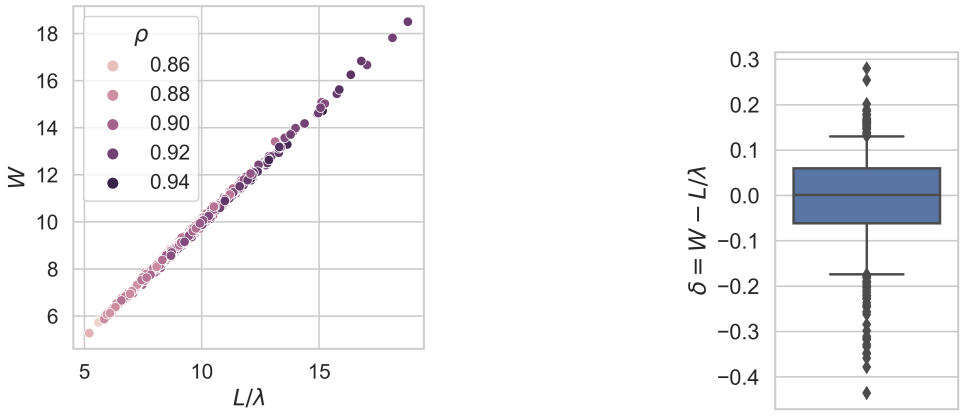
The distributions for the mean ρ, L, and W of 1000 replications are shown in Figure 6.1. As stated before, only the results for ρ follow a uniform distribution. W and L have more outliers that are higher than their mean values. That can be explained: periods of congestion in the queue randomly occur during some runs. These take long to clear and will therefore give higher queue length and waiting times during some runs. Meanwhile, there is a minimum to L and W because the server's utilization is high. The mean of means from 1000 replications almost equals the analytical solution for ρ. For the mean of means of L and W, there is respectively a 0.61% and 0.68% deviation from the analytical solution.

For test package B, this means that it is most useful to focus on ρ: the means taken from runs show are uniformly distributed, and it is proven that the analytical solution can be well estimated by a model. It can thus be tested at the end of each test case whether ρ is within acceptable bounds of ρ_{an}. Furthermore, it can be tested from multiple simulation runs whether the mean of means $\bar{\rho}$ is close to the analytical solution ρ_{an} given the inputs. The exact settings and bandwidths used in test package B are defined in the next section.

The outputs L and W can be tested as well. While there are some high outliers for the means of these outputs, it is found that Little's law, $L = \lambda W$, always holds for long runs of *M/M/1 queue*. This is shown in Figure 6.2a. An option is therefore to test the SUT for the deviation δ from Little's law. This is defined by rewriting Equation 4.6:

$$\delta = W - L/\lambda \tag{6.1}$$

Ideally, the difference δ is only due to the model's stochasticity, meaning that it should be normally distributed and should have a mean of 0. It is found that this is the case for δ from 1000 replications, see Figure 6.2b. Furthermore, δ only depends on ρ, not on either W or L. The difference δ can thus be used in test package B to further verify the SUT's results. An acceptable bandwidth for δ is defined based



(a) Relation between mean W, L, and ρ for 1000 replications of *M/M/1 queue* with $\rho_{\text{an}} = 0.9$.

(b) Distribution of difference δ.

Figure 6.2: Conformance to Little's law for 1000 replications of *M/M/1 queue* with $\rho_{\text{an}} = 0.9$. This shows that Little's law holds: W is proportional to L/λ.

on the results from these stochasticity experiments: an absolute difference of $|\delta| = 0.5$ is deemed acceptable. This way, if the mean L/λ is unexpectedly high or low compared to the mean W, the tester knows that something is wrong with the SUT.

Convergence of results for *M/M/1 queue*

The convergence experiment aims to find the order of magnitude for simulation end time t_{end} that gives an adequate steady-state solution for the most important outputs of *M/M/1 queue*. This is only done for the result ρ, since it was found in the previous experiment that the validated model gives the most consistent approximation of this variable. It will be assumed in test package B that if ρ is close enough to the analytical solution, and Little's law holds for L_{q} and W_{q}, then the model's results are as expected.

The experimental setup is as follows: the model is run with the same input parameters giving ρ_{an} for 4 different simulation end times. Per experiment 100 replications are run to account for stochasticity. The end times used are $t_{\text{end}} \in \{100, 1000, 10000, 10000\}$ s. The mean outcomes for ρ are compared to the analytical solution $\rho_{\text{an}} = 0.9$.

It is found that the solution converges for longer simulation times. This is illustrated in Figure 6.3: all replications approach the analytical solution over time, and the distribution of values for $\rho(t)$ becomes more narrow over time. It is still up to the tester which requirements on the steady-state value for ρ can be defined based on these results. A trade-off between a required level of accuracy and simulation time must be made, since the automated test package will be used to test many runs.

This is further illustrated in Figure 6.4, which gives the distribution of the same results. The test package could again use a bandwidth to define which range of outputs is accepted for ρ. With an accepted relative deviation of $d = 1\%$ from the analytical solution, 97% of runs of *M/M/1 queue* will be 'acceptable' if $t_{\text{end}} = 100\,000$ s. However, that takes a long time to run. For $t_{\text{end}} = 10\,000$ s, only 47% of runs is acceptable. This is of course worse for a more strict bandwidth of $d = 0.5\%$; then only 28% of runs is acceptable for $t_{\text{end}} = 10\,000$ s, versus 83% for $t_{\text{end}} = 100\,000$ s.

Input experiments for *M/M/1 queue*

With the stochasticity and convergence of *M/M/1 queue* defined, this model can be further analyzed for different sets of input parameters. The experimental setup for this is 7 experiments with different input parameters. λ is kept at 1.0 so that $\rho_{\text{an}} = 1/\mu$, while μ is varied: $\mu^{-1} \in \{0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4\}$. Per experiment 100 replications are run. This is repeated with $t_{\text{end}} = 10\,000$ s and 100 000 s.

The results for the mean ρ are shown in Figure 6.5. The *M/M/1 queue* model clearly gives a good estimation of the analytical solution for all inputs where $\rho_{\text{an}} < 1$ where this is possible. The variances in results are not constant over the input space, thus not the same percentages of runs would be within the accepted bandwidth for different inputs. With higher $t_{\text{end}} = 100\,000$ s, all experiments with $\rho_{\text{an}} < 1$ give at minimum 94% of replications with results within the bandwidths.

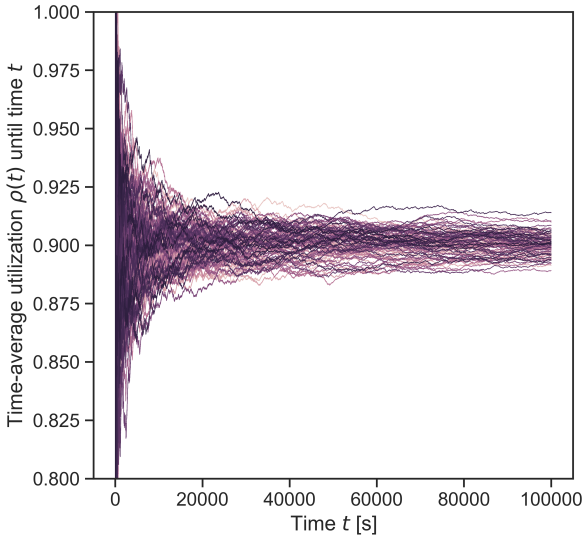


Figure 6.3: Time-average utilization $\rho(t)$ until time t for 100 replications of *M/M/1 queue*, with $\rho_{\text{an}} = 0.9$. The results converge to the a steady-state solution over time.

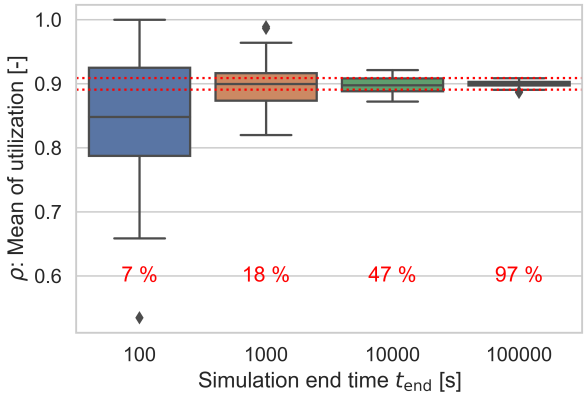


Figure 6.4: Distribution of mean utilization ρ for 100 replications with different t_{end}. The red dotted lines give the bandwidth for $d = 1\%$ deviation from the analytical solution. The red text gives the percentages of samples that is within this bandwidth.

Again, a lower number of runs has results for W and L that are within a $d = 1\%$ bandwidth. This is summarized in Figure 6.6. Clearly, the variance in mean W and L across runs is less for lower ρ_{an}. That is as expected: a lower utilization means less probability for random congestions. It is found that Little's law also holds for all inputs, even when $\rho_{\text{an}} \geq 1$. The difference $\delta = W - L/\lambda$ has a higher variance for higher ρ_{an}. The maximum $|\delta|$ is almost 0.5.

6.2.3. Decisions for verification of results in test package B

In conclusion, a test developer must make many choices in order to implement an idea of what acceptable outputs should be. Some considerations for implementing simple bandwidth tests are now listed, along with decisions made for test package B based on the stochasticity and convergence experiments.

- The *representative outputs* that can prove the SUT's functioning must be chosen. It is found that aggregates of a run's outputs are most convenient to use: the mean and/or standard deviation. It is further hypothesized that for using a bandwidth test, the mean of an output ideally has a normal distribution. For test package B, the mean utilization ρ and the difference $\delta = W - L/\lambda$ are thus chosen as outputs.
- An adequate *simulation end time* t_{end} must be chosen, that is known to give results that have

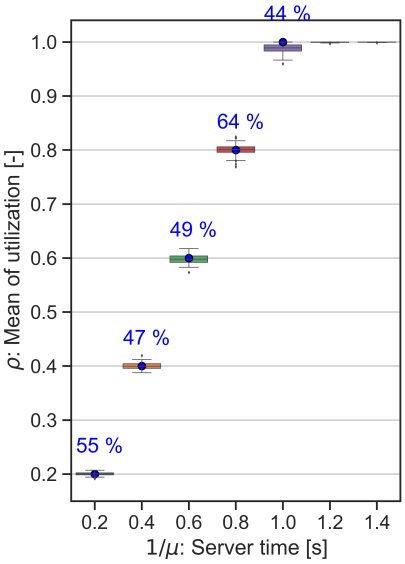


Figure 6.5: Distributions of mean utilization ρ from 100 replications of *M/M/1 queue* with $t_{\text{end}} = 10\,000\,\text{s}$, for different input values of $1/\mu$. The analytical solutions are given in blue where applicable. The blue text indicates the number of means that are within a $d = 1\%$ bandwidth from the analytical solution for ρ.

converged to a steady state. For test package B, it is chosen that $t_{\text{end}} = 10\,000\,\text{s}$ is a good trade-off between accuracy and execution time.

- The *number of replications* needed to account for the model's stochasticity must be defined. This is again a trade-off. The analysis of *M/M/1 queue* was done with 100 or 1000 runs per experiment, but it is decided to use 10 replications per experiment in test package B.
- The *acceptable relative deviation* d from the analytical solution must be chosen based on an analysis of the stochasticity of a validated simulating model across its input space, or if available from a known distribution that is found analytically. For test package B, $d = 1\%$ is used for the mean ρ. For δ, no relative deviation can be used, since its mean should be 0. Therefore, an absolute margin of $|\delta| = 0.5$ is used.
- The *acceptable number of runs with outlier results* can be defined based on the other criteria and experimental setup. For test package B, it is decided that 40% of runs should be within the bandwidth for ρ, combined with the requirement that the mean of means $\bar{\rho}$ should be within the bandwidth. Furthermore, 100% of runs should be within the bandwidth for δ.

6.3. Implementation of verification of results in MBT test packages

It is now explained how the aforementioned methods for verification from multiple runs can be implemented in an AltWalker test package with a test execution script, as is done for test package B.

6.3.1. Implementation in AltWalker

The experimental setup for batch runs and the oracles that do numerical tests on the results must be implemented in the MBT packages. It is now explained how this is done in AltWalker. The oracles are specified on two levels: there are bandwidth tests per test case (thus per simulation run), and there is an analysis of the results from multiple test cases.

Bandwidth tests per test case

The bandwidth tests can be easily implemented into the AltWalker test package. The analytical solutions must be defined for each set of inputs used in the test execution script. The lower and upper margins of the associated acceptable bandwidths are then calculated as well. These are passed to the test script for each test case run. The test script then evaluates the representative outputs at the

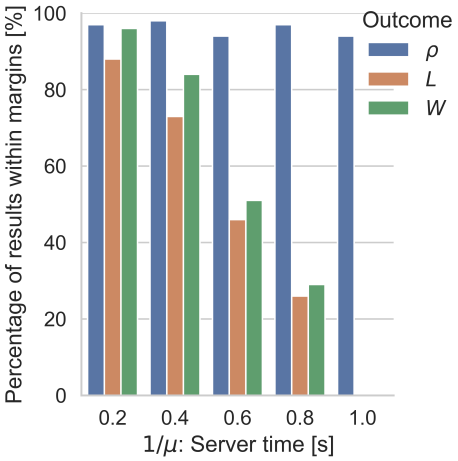


Figure 6.6: Percentage of mean outcomes that are within 1% of the analytical solution, from 100 replications with $t_{\text{end}} = 100\,000$ s.

end of a test run in a function associated with the end vertex, or in the `tearDownModel` or `tearDownRun` fixtures. A verdict is generated per representative outcome that summarizes that test case: within bounds, outside of bounds, or unexpected. This is passed again to the test execution script, before the next test case is run in a new AltWalker instance.

Analysis of results from multiple test cases
Once all experiments defined in the test execution script have been completed, the results from multiple test cases can be considered. This is done in the test execution script as well. Functionality is added that checks whether the number of runs within bounds is as expected. Furthermore, the mean of means is calculated for the representative variables. This finally results in a verdict on whether the SUT has the expected outcomes.

6.3.2. Combining verification of dynamic behavior and verification of results

Test package B with the experimental setup as just described takes a long time to run. This is because verification of the dynamic behavior is combined with verification of the model's aggregate outcomes. This first goal is done with more usual MBT techniques that are computationally expensive: tests are executed for each step of a simulation run. Long simulation times may not be crucial for this testing goal. It may be assumed for simple SUTs such as *M/M/1 queue* that if its dynamic behavior is as expected for a short run, it will be as expected for a long run. On the other hand, long simulation times are crucial for the goal of verifying a simulation model's steady-state outcomes, as is established in this section. The same holds for the use of multiple replications.

A hybrid can be achieved by developing an automated test package that combines model-based test cases of shorter simulation runs with (aggregate) output verification of longer simulation runs. The use of the test model and an MBT tool such as AltWalker is then not needed for this verification phase. This can again be put into terms of white-box testing vs. black-box testing: an MBT approach with a test model is shown to be feasible for white-box testing of a simulation model in Chapter 5, while it is now shown that simple black-box testing of only the inputs and some inputs can work for the verificaiton of simulation models if the expected outputs are known. These methods combined give insights into potential faults in the SUT; where in the code they occur, when during a simulation run they occur, and whether they affect the model's results.

7 Discussion

The results presented in Chapters 2 – 6 are now discussed. This is done to answer the four subquestions as defined in Chapter 1, which are repeated here:

1. What known options for model-based testing are relevant for testing simulation models?
2. How can test models be designed for step-by-step verification of the dynamic behavior of a simulation run?
3. How can test models be designed for verification of simulation models, with minimal knowledge of the SUT implementation and/or its internal variables?
4. How can automated test packages be developed, that use or integrate MBT techniques for verification of a simulation model's results?

In addition, the advantages and disadvantages of the MBT software tool of choice, AltWalker, are discussed.

7.1. Relevant options for MBT of simulation models

Options for different dimensions of MBT have been summarized based on taxonomies and case studies from literature in Section 2 to answer subquestion 1. The relevance of some options for testing simulation models has been hypothesized mainly by identifying which properties of simulation models make them hard to test, and by exploring which MBT options can address these properties.

Options selected for automation of testing

Some relevant options for MBT dimensions could be selected by simply adhering to a strict definition of MBT as a test technique that should involve the automation of the following steps of the testing process: (i) test path generation, (ii) mapping to executable test cases including oracles with expected behavior, (iii) test execution on the system under test (SUT), and (iv) oracle checking to assign verdicts to test cases. This means that the model scope is input-output, the modeled artifact is functional behavior, and the technologies of generation and execution are trivially automatic. Adherence to this strict definition of MBT is justified as automation is one of the main potential benefits of MBT over manual testing.

A step-by-step approach is also generally expected within this definition of MBT. This term is used in this report to indicate that for every input in a test case, an oracle with expected behavior is present as well. This approach is most in line with the purpose of verification of the dynamic behavior of the SUT. By assessing the SUT's response to every input, the test package can be used to precisely analyze when a failure occurs. With a more white-box approach, it can further identify what part of the SUT implementation produces the failure. The manual tester can use this information to see why a failure occurs.

Options selected for testing reactive systems and simulation models

Other options for dimensions have been adapted from previous research on MBT for reactive systems and simulation models. The test model should use a transition-based modeling paradigm. The test

model should mostly have the same characteristics as the SUT. A focus is made on non-real-time simulation models with a discrete-time base and inherent stochasticity. Therefore, the test model should have the same notion of time. Based on this idea, it is hypothesized that the simulation modeling paradigm used in the SUT may influence what design works best for the test model. Previous research indicates that a deterministic test model could be used for nondeterministic SUTs, and vice versa, so this option is left open for exploration.

The use of online, reactive test execution is another option taken from previous research. This is seen as the only relevant option for testing reactive systems such as simulation models. It cannot be known before a reactive system is executed what events it will produce exactly, as its response to inputs depends on its internal states. Therefore, the path generation should be aware of the SUT's states, in order to select paths that lead to relevant test cases. This need for reactive tests is even greater for nondeterministic SUTs, such as the simulation models with inherent stochasticity used in this project. Furthermore, simulation models are made to generate endogenous events upon time advancement. A test model must thus also know what event the SUT may or may not generate next, given the SUT's state.

Options left for case studies

It is now summarized which options are left open for exploration in the case studies. Firstly, how automatic test generation and test execution can be achieved is no topic of research, as existing MBT software tools provide this functionality. This means that this project has mainly focused on how test models can be designed for model-based dynamic verification of simulation models. It is left open whether test models should be nondeterministic, what test selection criteria should be used, and how paths should be generated through the transition-based abstract models. It was not clear how online, reactive testing ties in with path generation, and how the notion of time can be kept the same between the test model and SUT.

Furthermore, it was found from the case studies that a separation of the test model into an environment model and a behavior model could be beneficial. And lastly, it is known from the literature that the problem of test abstraction is fundamental to any MBT development process. This means: how can the SUT's expected behavior be represented and generated from the test model in an abstract manner. It is assessed to what extent these problems have been solved by looking into the other subquestions.

An option that is left open, that is relevant for the verification of a simulation run's results, is data coverage criteria for test generation. It has been shown in Chapter 6 that these can be implemented by simple sampling of (initial) input parameters for the SUT, and making multiple test cases based on these parameters.

Lastly, it was concluded in Section 2.4.2 that the level of oracle specification must be decided on. A choice can be made on whether oracles are specified for every step of the test path, and whether the test model gives exogenous inputs or only time advance instructions to the SUT. It is further discussed in Section 7.3 how oracle specification and inputs from the test model have been implemented in the developed test models.

7.2. Verification of dynamic behavior

To answer subquestion 2, test models have been developed for the case studies in Chapter 5 with three different simulation models used as SUT. The case studies demonstrate that MBT of simulation models is feasible. However, they do not provide a complete exploration of all MBT that were left open in the literature study. Some additional approaches to test model design are given in Chapter 5, but these are not further explored.

Usage of online, reactive test execution and test generation options

It has been explored what online, reactive test execution means for MBT of simulation models, and with what path generation options it can be integrated. Three options are defined in Section 5.3.1 for how a test model can trigger events in the SUT. The first option is named 'monitoring-only', where the test model only gives time advance instructions to the SUT during a test, and input parameters only at the start. Giving the same input over and over is unusual for MBT, but it is adequate here because the SUT is a simulation model that generates endogenous events upon time advancements. This 'monitoring-only' approach has led to three test models for different SUTs, with the common characteristic that

path generation closely follows the SUT's state. It is discussed later what implications this monitoring approach has for the test model characteristics and the usefulness of MBT.

Two other options defined for event triggering by the test model are 'mediated runs' and 'explorative runs'. Unfortunately, these options have not been further explored in this project. They would be useful for the second type of verification defined in Section 2.4.2; verification of responses to exogenous variables. In both approaches, specific inputs, along with time advancement instructions, are given to the SUT by a test model. These two approaches seem most in line with the existing MBT case studies from literature. A behavior test model (also called an input model) is often used to provide inputs. These inputs are then used both on the SUT and on a behavior test model (also called a SUT model) that models how the SUT should respond.

Because they are not explored, the distinction implied between 'mediated runs' and 'explorative runs' may be unclear. The ideas tie in with two benefits that triggering events in the SUT could have for testing simulation models. Firstly, a 'mediated run' could be useful for integration testing and component testing, where the test model generates inputs that a SUT would normally get from the environment that it is connected to, for example, other simulation model components. Secondly, an 'explorative run' could be useful to make more efficient and specified test cases, and for robustness testing. The test model rapidly causes events in the SUT that are suspected to give faulty behavior, instead of waiting for the SUT to generate such events by chance.

Implications of a monitoring approach

The use of a monitoring-only approach has some implications for the usefulness of MBT. Firstly, a problem left open for the case studies was the selection criteria, such as coverage criteria, and technology for test (path) generation. In the end, path generation algorithms and complex coverage criteria have not been used much in the case studies: only a 'random coverage' criterion has been used to run the test models. It is now further explained why complex path generation is not needed in a monitoring approach where the test models are designed to follow the SUT's state.

Stochasticity of test path generation, often the main reason to choose for MBT, has actually not been used in this project, because the SUT is now stochastic itself. With the use of online, reactive tests, the path generation is continuously influenced by the SUT's current state. The best example is the test model for *M/M/1 queue*. All transitions in the test model are completely 'locked' by guards: a transition in the test model will only be triggered if it is detected that a (relevant) variable has changed in the SUT. The test model for *Airport* works similarly: here transitions are triggered by messages about events in the SUT. This means that the test models themselves are in fact deterministic.

The fact that the test models are deterministic, is not necessarily bad design. MBT is usually applied when it is deemed efficient to generate a vast number of test cases with a stochastic test model. Here, similar techniques are used to automatically follow the states of a stochastic SUT during testing. Thus, test generation and execution are automated, meaning that the benefits of MBT over manual testing are still relevant. Alternative approaches that do not use test models would probably involve assertion functions in the SUT that are less clearly selected based on the SUT's state. Moreover, manual testing alternatives could mean that a tester has to look at a list of events produced during a simulation run.

The design choice to have the test model's state follow the SUT's state also means that eventual faulty transitions or events of the SUT should be implemented in the test model as well. The step-by-step oracles are thus specified in two ways: 1) Given the test model's current state, assertion functions are used to evaluate the SUT's current state. This is done by mapping the vertices of the abstract model to assertion functions. 2) Given the test model's current state, an unexpected state transition and/or message from the SUT triggers a transition in the test model that is defined as faulty. A 'failed' verdict is then assigned.

Selection of representative states

A problem that is relevant not only to a monitoring-only approach is the selection of representative states. It is found in the case studies that the problem of test abstraction is indeed fundamental to MBT. This requires creativity from the test developer, and how this problem can be solved highly depends on the inner workings of the simulation model under test. Given the test goal, the most important functional behavior should be taken from the functional requirements. For example, the testing goals for *M/M/1 queue* and *Airport* are different, while these SUTs have similar functional requirements. The test model for *M/M/1 queue* only tests whether entities are in the correct 'order' in the SUT, while the test model

for *Airport* also involves time constraints for the processing of entities. The test model for *M/M/1 queue* would thus not notice a faulty SUT where the entities stay in service longer than their maximum service time.[1]

An idea is mentioned in Chapter 5 that representative states could be defined by determining which system states of the SUT warrant that different assertion functions are used. The test model thus selects some unit tests that are particular given the SUT's state. It is found that for simple simulation models, such distinctions cannot always be made. *Two-way switch* for example could be 'tested' by asking one question during the entire simulation run: is exclusive-OR logic followed? What is missing in this research is therefore testing of more complex simulation models that can show different patterns of behavior. For these systems, what type of behavior is expected and what assertions should be run is even more dependent on the SUT's state. The *M/M/1 queue* has such a distinct pattern of behavior because it becomes unstable when $\rho > 1$. Test model B is (unintentionally) adequate for that situation in terms of step-by-step verification

On a related note, the process of selecting representative states has shown that the idea found in literature is true, that the validity of the test model itself should be continuously checked during test development. Even for the simple SUTs used in this project, it is a time-consuming process to make abstract test models and mappings that somehow model what the SUT should be doing. When one assumes that the test model could be invalid, a failure found in a test case could be either due to a fault in the SUT or a fault in the test model.

Options for time advancement and model notation

One novel idea is found for time advancement in the case studies. Normally, edges of the transition-based abstract test model are associated with inputs to the SUT. However, it can be useful to use the vertices instead of edges of the abstract models for time advancement, so that guards in the test model are aware of new state transitions in the SUT. This is again a consequence of the reactive and monitoring design of tests in this project.

The hypothesis that the simulation modeling paradigm used in the SUT dictates how the test model should be designed is only partly true. One relevant difference is found between agent-based modeling (ABM) and discrete event simulation (DES): the test model can model time on its own if the SUT uses ABM, while the test model must query the SUT's logical time if the SUT uses DES. That is no problem for online, reactive tests. Furthermore, it is found that step-by-step testing is easier to conceptualize for DES, as every time advancement will generally give one output from the SUT, namely one event. This is different for an ABM, where during one time step many events of different processes may have happened in unknown order, or potentially no event may have happened.

Regarding the modeling notation of the test model, a few new insights can be given. Only one type of notation for transition-based models has been used: extended finite state machines (EFSMs). EFSMs seem adequate for testing simulation models: time can be kept as an internal variable, the path generation can be guided with guards and coverage criteria, and elements of the abstract model can be mapped one-to-one to assertion functions. Furthermore, state machines are familiar to simulation model developers, they have graphical representations, and they are supported by most MBT tools. A disadvantage is the problem of state explosion: the number of states needed to represent a system increases rapidly with the system's complexity, and the number of state transitions needed increases exponentially. This is especially relevant for simulation models, which consist of multiple processes that all have states. Selection of representative states and encapsulation into system states is therefore essential for test model design, as explained earlier. A remedy against defining many state transitions by hand is to make submodels of separate processes in the SUT, and to compose these into larger state machines. This is supported even by some MBT tools, but it has not been explored in this project.

Some literature on MBT of real-time systems, which deal with similar problems as in this project, have used transition-based models where time is kept as a variable that is different from other variables. For example, the case study by Poncelet (2016) uses timed input-output systems with timed automata, where advancement of clocks is done separately from the state machine. It is advised for further research to focus on such model notations for tests that involve timing issues.

[1] Luckily, for *M/M/1 queue* such faults would lead to mean outputs that are different from the analytical solution, which may be noticed by the test package once it verifies the steady-state outcomes from multiple simulation runs.

7.3. Requirement-based testing and modular approaches

Subquestion 3 has been defined as was found in that functional, black-box testing is strongly recommended for MBT. It has been found in the literature study that the term 'black-box testing' can refer to two closely related concepts: test design based on the SUT's requirements instead of its implementation, and tests where the SUT's internal states are not accessible.

Feasibility of black-box tests

It seems that requirement-based test design is an ideal. It is hypothesized in literature that doing design of the abstract model only based on the functional requirements will improve the test quality. This hypothesis cannot be discussed based on this project's results, as test quality is not within the scope, and the validity of the developed test packages has not been assessed. However, it has been demonstrated that model-based black-box tests could need a different test model design compared to more white-box tests. The test package for *Airport* has been developed for this purpose. For further discussion, it must be noted again that what counts as the SUT's interface and its internal variables is arbitrary; this can be changed for testing purposes.

As said earlier, the requirements for *M/M/1 queue* and components of *Airport* are similar: both should simulate a server with a capacity of 1 and a queue. The test model for *M/M/1 queue* follows what state the SUT is in: it is made for a reactive test where the SUT's state determines what state the test model transitions to. The test model for *Airport* on the other hand models what state the SUT should be in. The test model's state transitions are based on the SUT's interface only, where 'arrivals' are the inputs that the SUT gets, and 'departures' are outputs of the SUT. The idea in this project is that this is a more black-box design, compared to the test package for *M/M/1 queue*, because the SUT is seen as a black box. The test model has access to fewer internal variables of the SUT.

This approach seems more in line with the approaches to MBT found in the case study review of Section 2.4: the same inputs are used on the SUT and on a behavior test model of the SUT. The difference to most existing case studies is that no environment test model had to be used for *Airport*: the inputs are taken from other components of the SUT. This is thus an example of integration testing, where a component is tested in its actual environment.

Three points can be made on the idea of more black-box testing. Firstly, the test model development was still done with knowledge of the SUT's implementation in mind, so it is not truly requirement-based testing. Secondly, the test model for *Airport* had to be made more complex than the test model for *M/M/1 queue*, as it is based on minimal information. This means that its test model almost has the same functionality as the SUT; it could be seen as a reference for the SUT. Of course, for testing more complex SUTs, one would be advised to develop a test model that focuses only on the core functions of the SUT. And thirdly, a developer may be more tempted to design the test models based on the SUT implementation, when reactive testing is used. In non-reactive tests, the SUT's states are only accessed for oracles. But in reactive tests, the SUT's states are accessed to influence path generation as well. This closer integration between the test model and SUT thus means that the test design may be more implementation-based, or more white-box. This idea is not only relevant for testing of simulation models, but for any type of SUT.

Modular approach to MBT

Another hypothesis was added in Chapter 2 that black-box testing may give two additional uses for MBT: one test model could be used to test SUT components, or to test SUTs with similar functionalities.

For the first use case, it has been demonstrated with the test package for *Airport* that one test model can indeed be used to test components of a SUT that are very similar. Unfortunately, an attempt was unsuccessful to develop a test package where multiple test models test multiple components of the SUT, ideally during one run. It is unclear how path generation should traverse the different test models, while the SUT's states change for every time advancement. Concurrent states and parallel execution may solve this problem, and much research on MBT with concurrent states is available. However, this will make test development more complex.

For the second use case, the test package for *Airport* has not been executed with other similar SUTs due to lack of time. It is likely that this is possible: the test model includes an impromptu message protocol that could be used on other SUTs with minimal effort. This alleviates the problem mentioned in literature, that development of mappings is time-intensive. In conclusion, using a test package on

similar SUTs could be an interesting step in further research. It can further demonstrate the usefulness of using abstract test models.

Approaches to verification based on oracle specification and types of inputs

The three approaches to verification, as distinguished in Section 2.4.2, are also relevant to discuss the difference between black-box and white-box testing. These approaches are now repeated: 1) step-by-step verification of endogenous variables, 2) verification of responses to exogenous variables, and 3) verification of a simulation model's results, given its initial input values. This classification was made with the idea that different levels of oracle specification are needed and that the SUT's state should be verified after every time advancement. These approaches are now linked to the levels of oracle specification mentioned throughout this report. An overview is given in Table 7.1

Table 7.1: Levels of oracle specification and approaches to verification as mentioned in this project

Level of oracle specification	Approach to verification	Done in test package
Oracles on many steps for internal logic of generated events	Verification of dynamic behavior	A, B, C
Oracles on some steps for correct response or time constraints	Verification of dynamic behavior, Verification of responses to exogenous inputs	C
Oracles at last step of test case, Oracle of entire test suite	Verification of (end) results	B

The first approach, step-by-step verification of endogenous variables, is the main idea behind the test models that have been developed. It seems that most existing MBT applications also use a step-by-step approach, with oracles specified to all steps of the test case. However, now that every step of the test case can lead to endogenous events in the SUT, because it is a simulation model, the test model could make verdicts on *every* event that SUT generates. Evaluating the SUT's state for each step and specifying oracles for many steps then seems more like a white-box approach. It can be explored in further research whether this step-by-step approach is useful for structural testing of simulation models.

The second approach, verification of the response to exogenous variables, is partly done in test package C. A component within the SUT is tested, of which the inputs from other components can be seen as exogenous events. The test model determines within what time the SUT should respond to these events, and checks this. This test model is designed somewhat differently for this purpose: it uses multiple implicit time advances associated with the vertices, see Section 5.3.2. However, test package C is made for monitoring only, while verification of the response to inputs would be most useful for mediated and explorative runs, where the test model gives inputs to the SUT during a simulation run. That is in line with controlling the exogenous stochasticity of a simulation model, in order to make dynamic verification more feasible, as proposed by Mihram (1972).

The third approach has been explored in Chapter 6, where oracles are only specified for the end results of (multiple) simulation runs, specifically for the means of results. This is clearly a black-box approach. It could be useful for more complex simulation models to consider the other options mentioned in Chapter 2, such as verification of intermediate results or time series analysis techniques. This approach is now further discussed.

7.4. Verification of results of a simulation run

Subquestion 4, the automated verification of a simulation model's results, has been explored in Chapter 6. A point must first be made on the research design itself. Functional requirements for verifying results should be specified differently than those for step-by-step verification of dynamic behavior. For dynamic verification, the functional requirements could thus specify which events the simulation model (being the SUT) may generate given its state and inputs. For verification of the simulation model by its results, the functional requirements should include some known solution or an expected range of outputs of the system that the simulation model is modeled to.

It can be concluded from the developed case studies that verification of the dynamic behavior is the goal that is most in line with the functionality provided by MBT techniques. Verification of the

results of multiple simulation runs, on the other hand, does not necessitate the use of test models as is done for MBT. Oracles only need to be specified to compare the SUT's results to the expected results, given the initial conditions. Analysis of the results of multiple runs is however an important step for the verification of simulation models, and after this for validation as well. One can think of a simulation model that generates the correct events and responds to exogenous inputs as specified, but that still gives outcomes that are not correct. This could be especially true for the stochastic simulation models used in modeling and simulation for decision-making (M&S).

Therefore, an attempt has been made to combine verification of dynamic behavior and verification of simulation run results in this project. Chapter 6 shows that this is feasible, but that it is best to separate test cases for dynamic verification from test cases for validation of results. This is because of a trade-off for how long tested simulation runs should be: step-by-step testing as is done in MBT is computationally expensive, while the statistic analysis used on the results often necessitates many long simulation runs.

7.5. Advantages and disadvantages of AltWalker

Throughout the project, the features and problems with the MBT software tool of choice, AltWalker, have been explored as well. It is found that the GUI editor is useful for quickly building test models, and that it is convenient that test mappings can be written in Python using common testing libraries. A disadvantage for practicality is that the functionality is scattered across two programs: Java and Python. A bug in communication between these programs makes AltWalker not suited for step-by-step testing of simulation models.

For efficiency, it would be nice if some conditions were continuously checked, regardless of which state the test model is in. This is useful for ending the test execution when a time value or error has been reached, or to temporarily disengage the entire test model until a condition holds. This last approach would be helpful for efficiently testing long simulation runs, where a different behavior pattern is expected later on.

Other missing features are concurrent states, conditional transitions between submodels, and parallel execution of test cases. Concurrent states would especially be useful for testing multiple components of one simulation model at once. It was found during the selection process that other open-source Python-based MBT tools lack similar features. It is recommended to reconsider these options and to check if batch execution and analysis of multiple test cases is supported.

8

Conclusion

This report has discussed how model-based testing (MBT) can be applied for the dynamic verification of simulation models. This has been done first by a literature review to find MBT options that may be relevant for testing simulation models. MBT taxonomies and case studies have been considered for this. These options have been further explored in new case studies, consisting of test suites for three simple simulation models. This is done to see what considerations a test developer may encounter during test development. An existing MBT software tool has been selected for this purpose. The requirements and preferences for the selection process of this tool have been based again on the relevant options for MBT of simulation models.

It is found that current MBT literature and MBT software tools are mostly not focused on testing simulation models. Most literature found on this topic consists of case studies, where it is often not described what options for test model design are considered, and how the choices can be justified. Moreover, often only some steps of the testing process are automated. Most existing case studies further do not describe the influence on test model design that some unique properties of simulation models may have: timed variables, many states at once, stochasticity, unknown output traces, and large input spaces.

Three main topics have been explored in this thesis: verification of dynamic behavior during a simulation run, verification of a simulation model's results against known solutions, and the distinction between black-box and white-box testing. This demarcation into topics is based on the idea that verification of dynamic behavior is in line with the step-by-step approach commonly used in SUT, while verification of (end) results, given the initial inputs, is a common step for testing simulation models. The implications of black-box testing are deemed important as well, because test design based on requirements only is often advised for MBT.

It is demonstrated how a test package can combine dynamic verification and verification of results, but that MBT techniques are only needed for the dynamic verification step. Moreover, verification of results may necessitate long simulation runs, while long simulation runs are taxing for dynamic verification as this is computationally expensive. It is therefore advised to use different test suites for these two purposes. Automated verification of end results can be regarded as a better choice than step-by-step dynamic verification for testing whether a simulation model's aggregate results or variance in results is conform an (analytical) solution or another validated model.

The development of different case studies has indicated that black-box testing, where only the interface of the system under test (SUT) can be accessed, may necessitate a more complicated test model. This is because test models for this purpose can only work if they model the SUT's state, whereas test models for more white-box testing can access and follow the SUT's internal states. Modeling the SUT's state implies that the test model needs more internal variables, state transitions, and guards.

It is found through literature search that some options for MBT are advised for SUTs that have properties that are similar to simulation models, namely timing constraints, nondeterminism, and a reactive nature where the response to inputs depends on the internal states. Online, reactive test execution is seen as the best option, as the generation of test cases can be directly influenced by the

SUT's internal states on a step-by-step basis. Furthermore, it is decided to automate as much of the testing process as possible, since automation is one of the main reasons to use MBT.

The technology needed for model-based test case generation and test execution is already provided by existing MBT software tools. Therefore, this project has mainly focused on another dimension of MBT: test model specification. It is found that transition-based test models are most commonly used, recommended, and supported for MBT. Other aspects of how the test models should be designed are left open for exploration in three case studies. Each case study shows the development of a test package for MBT of a simple simulation model. Simulation models that use different modeling paradigms have been to explore whether this has an influence on how test models should be designed.

By developing test models for the case studies, it is found that event triggering in the SUT is an important design choice. A property of simulation models is that they will generate endogenous events when their logical time is advanced. This means that some simulation models do not need exogenous inputs during a simulation run to show their normal behavior. With such simulation models as SUT, a test model only has to give 'time advance' instructions as input to the SUT during a run, along with initial values at the start of a run. This approach is named a 'monitoring-only' run. Alternative approaches named 'mediated' and 'explorative' runs are defined in this report, but have not been implemented in the case studies. These involve a test model that gives exogenous inputs to the SUT during a run, along with the instructions for time advancement.

The case studies demonstrate options for test model design using a monitoring approach. The test models are made to closely follow the SUT's current state. This demonstrates the usefulness of online, reactive tests for MBT of simulation models: the test model is aware of what the SUT does and can select relevant oracles accordingly. The problem of test abstraction can be approached with this in mind: the SUT can be abstracted into representative states or patterns of behavior, based on when different assertions would be relevant.

Combined with a step-by-step approach, where the expected behavior is generated and checked for each time advancement, this monitoring-only design can already give a modeler insight into when, where, and why faults occur. This can be seen as an advantage of MBT over manual testing for simulation models specifically: checking what happens at every time advancement by hand would be labor-intensive, especially as verification of simulation models often necessitates that many simulation runs are analyzed.

It is argued that this 'monitoring-only' approach is not in line with how MBT has been applied for testing reactive systems in case studies from previous literature. Usually, a test model simulates what the SUT should be doing, instead of following what the SUT is doing. One example is therefore developed of a test model that uses the same inputs as the SUT gets, in order to generate the SUT's expected behavior. It is shown that this necessitates a more complex abstract model. This approach can be applied to use MBT for another potential advantage: reusing test models to test similar components within SUTs, or to test similar SUTs. This use has not been successfully demonstrated in this project. An extensive effort into this topic may lead to composable test packages, that can test different compositions of SUT components and their interoperability.

The aforementioned alternatives to the 'monitoring-only' approach are be more in line with how MBT is often used, namely with a test model that gives exogenous inputs to the SUT during a run. This can be done to trigger specific events in the SUT, or to test its response to inputs. This approach may demonstrate additional benefits of using MBT. The approach can be useful for component and integration tests, where the test model simulates the environment that a SUT normally interacts with. It can also be used to generate more efficient test cases as specific events could be triggered in the SUT, without waiting for those events to be generated by the SUT itself. This approach is therefore recommended for further study.

Lastly, it is found that existing case studies from literature often do not mention on what level oracles are specified, while this is an important choice for test design. In the case studies developed in this project, the SUT's state is evaluated after every time advancement, and oracles are specified for relevant state transitions. This approach is called 'step-by-step' and is found useful for the monitoring-only testing of dynamic behavior as done in this project. It is hypothesized that verification of the response to exogenous inputs does not necessitate oracles for every step. Furthermore, it is found for the verification of a simulation model's results, only oracles are needed that compare the results at the end of a run to known solutions, given the initial conditions.

In conclusion, this project has demonstrated that MBT of simulation models is feasible and that it is most useful for verifying the dynamic behavior. Test suites generated with MBT techniques can be combined with other verification techniques to give a full assessment of a simulation model's functional behavior. A focus on test model design by developing case studies for different SUTs has highlighted an approach where a test model follows the SUT's states and selects relevant oracles, as assertions functions, accordingly. Other approaches where the test model provides inputs to the SUT during a run have been described, but not demonstrated.

Recommended further steps are the development of test models to test simulation models that show different patterns of behavior. Furthermore, test models should be developed that give inputs to the SUT during a run. And lastly, test packages should be proven useful by showing that they can detect faults in mutant SUTs, as is done in other research.

Bibliography

AltWalker. (2023a, July). AltWalker 0.3.1 documentation. Retrieved October 1, 2023, from https://altom.gitlab.io/altwalker/altwalker/index.html

AltWalker. (2023b, July). API Documentation. Retrieved November 13, 2023, from https://altwalker.github.io/altwalker/api.html

Banks, C. M. (2010). Introduction to modeling and simulation. In J. A. Sokolowski & C. M. Banks (Eds.), *Modeling and simulation fundamentals: Theoretical underpinnings and practical domains* (pp. 1–24). Wiley.

Banks, J., & Carson, J. S. (1986). Introduction to discrete-event simulation. *Proceedings of the 18th Conference on Winter Simulation*, 17–23. https://doi.org/10.1145/318242.318253

Berkenkötter, K., & Kirner, R. (2005). 13 Real-Time and Hybrid Systems Testing. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, & A. Pretschner (Eds.), *Model-Based Testing of Reactive Systems: Advanced Lectures* (pp. 355–387). Springer. https://doi.org/10.1007/11498490_16

Binder, R. V., Legeard, B., & Kramer, A. (2015). Model-based testing: Where does it stand? MBT has positive effects on efficiency and effectiveness, even if it only partially fulfills high expectations. *Queue*, *13*(1), 40–48. https://doi.org/10.1145/2716276.2723708

Diaz, R., & Behr, J. G. (2010). Discrete-Event Simulation. In J. A. Sokolowski & C. M. Banks (Eds.), *Modeling and simulation fundamentals: Theoretical underpinnings and practical domains* (pp. 1–24). Wiley.

Fu, M. C., & Gross, D. (2013). Simulation of Stochastic Discrete-Event Systems. In S. I. Gass & M. C. Fu (Eds.), *Encyclopedia of Operations Research and Management Science* (pp. 1410–1418). Springer US. https://doi.org/10.1007/978-1-4419-1153-7_959

Gerhold, M., Hartmanns, A., & Stoelinga, M. (2019). Model-based testing of stochastically timed systems. *Innovations in Systems and Software Engineering*, *15*(3), 207–233. https://doi.org/10.1007/s11334-019-00349-z

Gross, D., & Harris, C. M. (1974). *Fundamentals of queueing theory*. Wiley.

Herd, B., Miles, S., McBurney, P., & Luck, M. (2014). Verification and Validation of Agent-Based Simulations Using Approximate Model Checking. In S. J. Alam & H. V. D. Parunak (Eds.), *Multi-Agent-Based Simulation XIV* (pp. 53–70). Springer. https://doi.org/10.1007/978-3-642-54783-6_4

Hollmann, D. A., Cristia, M., & Frydman, C. (2012). Adapting Model-Based Testing Techniques to DEVS Models Validation. *Proceedings of the 2012 Sypmosium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*.

Huq, F. (2000). Testing in the software development life-cycle: Now or later. *International Journal of Project Management*, *18*(4), 243–250. https://doi.org/10.1016/S0263-7863(99)00024-1

Iftikhar, S., Iqbal, M. Z., Khan, M. U., & Mahmood, W. (2015). An automated model based testing approach for platform games. *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 426–435. https://doi.org/10.1109/MODELS.2015.7338274

Jacky, J. (2011). PyModel: Model-based testing in Python. *Python in Science Conference*, 48–52. https://doi.org/10.25080/Majora-ebaa42b7-008

Kazil, J., Masad, D., & Crooks, A. (2020). Utilizing Python for Agent-Based Modeling: The Mesa Framework. In R. Thomson, H. Bisgin, C. Dancy, A. Hyder, & M. Hussain (Eds.), *Social, Cultural, and Behavioral Modeling* (pp. 308–317, Vol. 12268). Springer International Publishing. https://doi.org/10.1007/978-3-030-61255-9_30

Keränen, J. S., & Räty, T. D. (2012). Model-based testing of embedded systems in hardware in the loop environment. *IET Software*, *6*(4), 364–376. https://doi.org/10.1049/iet-sen.2011.0111

Kleijnen, J. P. C. (1995). Verification and validation of simulation models. *European Journal of Operational Research*, *82*(1), 145–162. https://doi.org/10.1016/0377-2217(94)00016-6

Li, N., & Offutt, J. (2017). Test Oracle Strategies for Model-Based Testing. *IEEE Transactions on Software Engineering*, *43*(4), 372–395. https://doi.org/10.1109/TSE.2016.2597136

Lindvall, M., Porter, A., Magnusson, G., & Schulze, C. (2017). Metamorphic Model-Based Testing of Autonomous Systems. *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, 35–41. https://doi.org/10.1109/MET.2017.6

Marinescu, R., Seceleanu, C., Guen, H., & Pettersson, P. (2015, December). A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs. In *Advances in Computers* (pp. 89–140, Vol. 98). https://doi.org/10.1016/bs.adcom.2015.03.003

Marques, A., Ramalho, F., & Andrade, W. L. (2014). Comparing Model-Based Testing with Traditional Testing Strategies: An Empirical Study. *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, 264–273. https://doi.org/10.1109/ICSTW.2014.29

Mihram, G. A. (1972). Some Practical Aspects of the Verification and Validation of Simulation Models. *Operational Research Quarterly (1970-1977)*, *23*(1), 17–29. https://doi.org/10.2307/3008511

ModelTestRelax. (2023, September). *ModelTestRelax overview* (Software Documentation). Retrieved October 4, 2023, from https://gitlab.inf.elte.hu/nga/ModelTestRelax/-/blob/master/docs/user_guide.md

Poncelet, C. (2016, November). *Model-based testing real-time and interactive music systems* [Doctoral dissertation, Université Pierre et Marie Curie - Paris VI]. Retrieved May 12, 2023, from https://theses.hal.science/tel-01528954

Poncelet, C., & Jacquemard, F. (2016). Model-based testing for building reliable realtime interactive music systems. *Science of Computer Programming*, *132*, 143–172. https://doi.org/10.1016/j.scico.2016.08.002

Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., & Stauner, T. (2005). One evaluation of model-based testing and its automation. *Proceedings of the 27th International Conference on Software Engineering - ICSE '05*, 392. https://doi.org/10.1145/1062455.1062529

Pretschner, A., & Philipps, J. (2005). 10 Methodological Issues in Model-Based Testing. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, & A. Pretschner (Eds.), *Model-Based Testing of Reactive Systems: Advanced Lectures* (pp. 281–291). Springer. https://doi.org/10.1007/11498490_13

Richardson, D. J., Aha, S. L., & O'Malley, T. O. (1992). Specification-based test oracles for reactive systems. *Proceedings of the 14th International Conference on Software Engineering*, 105–118. https://doi.org/10.1145/143062.143100

Roungas, B., Verbraeck, A., & Meijer, S. (2017). A Framework for Simulation Validation & Verification Method Selection. *Proceedings of the 9th International Conference on Advances in System Simulation*.

Roungas, B., Verbraeck, A., & Meijer, S. (2018). A Framework for Optimizing Simulation Model Validation & Verification. *International Journal on Advances in Systems and Measurements*, *11*(1&2), 137–152.

Sabbaghi, A., & Keyvanpour, M. R. (2017). State-based models in model-based testing: A systematic review. *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, 0942–0948. https://doi.org/10.1109/KBEI.2017.8324934

Salabim. (2023, July). MMc [Example model]. https://github.com/salabim/salabim/blob/master/sample%20models/MMc.py

Schieferdecker, I., & Hoffmann, A. (2012). Model-based testing. *IEEE software*, *29*(1), 14–18.

Schmidt, A., Durak, U., & Pawletta, T. (2016). Model-based testing methodology using system entity structures for MATLAB/Simulink models. *SIMULATION*, *92*(8), 729–746. https://doi.org/10.1177/0037549716656791

Schmidt, A., Durak, U., Rasch, C., & Pawletta, T. (2015, 4). Model-Based Testing Approach for MATLAB/Simulink using System Entity Structure and Experimental Frames. *Simulation Series*, *47*, 828–835.

Shafique, M., & Labiche, Y. (2010, 4). *A Systematic Review of Model Based Testing Tool Support* (Technical Report). Carleton University. https://repository.library.carleton.ca/concern/research_works/4m90dv50r?locale=it

Sokolowski, J. A., & Banks, C. M. (Eds.). (2010). *Modeling and simulation fundamentals: Theoretical underpinnings and practical domains*. Wiley.

TNO. (2021). *Model Based Testing* (Internal Report).

Tretmans, J. (2008). Model Based Testing with Labelled Transition Systems. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, R. M. Hierons, J. P. Bowen, & M. Harman (Eds.), *Formal Methods and Testing* (pp. 1–38, Vol. 4949). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-78917-8_1

Ullrich, O., & Lückerath, D. (2017). An Introduction to Discrete-Event Modeling and Simulation. *SNE Simulation Notes Europe*, *27*(1), 9–16. https://doi.org/10.11128/sne.27.on.10362

Utting, M., & Legeard, B. (2007). *Practical model-based testing: A tools approach*. Morgan Kaufmann Publishers.
OCLC: ocm73993672.

Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, *22*(5), 297–312. https://doi.org/10.1002/stvr.456

Van der Ham, R. (2023, July). *Salabim. Release 23.2.1* (Software Documentation). https://www.salabim.org/manual/salabim.pdf

Van Osch, M. (2005). Automated Model-Based Testing of χ Simulation Models with TorX. In R. Reussner, J. Mayer, J. A. Stafford, S. Overhage, S. Becker, & P. J. Schroeder (Eds.), *Quality of Software Architectures and Software Quality* (pp. 227–241). Springer. https://doi.org/10.1007/11558569_17

Wilensky, U., & Rand, W. (2015). *An introduction to agent-based modeling: Modeling natural, social, and engineered complex systems with NetLogo*. The MIT press.

Zander, J., Schieferdecker, I., & Mosterman, P. (2011, September). A Taxonomy of Model-Based Testing for Embedded Systems from Multiple Industry Domains. In *Model-Based Testing for Embedded Systems*.

Zander, J., Schieferdecker, I., & Mosterman, P. J. (2012). *Model-based testing for embedded systems*. CRC Press. http://www.crcnetbase.com/isbn/9781439818459

A

Functionality of AltWalker

This appendix gives details on the functionality of AltWalker, in addition to Section 3.2. Mainly the features, problems, and added functionality that are relevant to this project are given. For a full understanding of AltWalker and its uses, the reader is referred to *"AltWalker 0.3.1 Documentation"* (2023).

The reader is again referred to the code of the simulation models and test packages as developed in this project, on https://github.com/montequercus/MBT-sim

A.1. Model specification

A.1.1. Additions to abstract models

The notation of abstract models, or *directed graphs*, in AltWalker is discussed in Section 3.2.1. It can be added that functions in the test scripts can be re-used by using elements that have the same name. Furthermore, AltWalker's documentation mentions that (only) vertices are used to execute assertions on the SUT's output, but that is not true in practice. Edges and their associated functions in the test script can be used for this as well. Thus, the only real difference between vertices and edges is that edges can have guards. The modeler can specify a start vertex where the path generation should always start. An end vertex is not mentioned in the documentation, but can be implemented by setting a requirement on one vertex, and running with a `requirement_coverage` option.

A.1.2. Stopping path generation based on variable

An important feature in this project would be to stop path generation and test execution based on the value of a graph variable or SUT variable, as explained in Section 3.2.1. That would enable path generation to stop once the SUT reaches its simulation end time. This cannot be specified by simply setting a length for path generation, because the path length needed in the abstract model to reach a certain time in the SUT is unknown before running a (stochastic) reactive test. AltWalker unfortunately has no built-in stop condition that checks the graph variables. Options have therefore been explored to stop test execution based on a variable, such as time, from either the abstract mode, the test script, or the test execution script.

Options on how this is possible are explored by use of an example: an abstract model and a SUT simulation model both have time variable t. The test case must end if t reaches 50. An option is to create an end vertex, for example named `v_end`, in the abstract model. The abstract model can then be run with generator option `random(reached_vertex(v_end))`, as was done for the example in Figure 3.1. However, for this option it is necessary that many vertices will have an edge to `v_end`, with a guard `t==50` on each edge. This will make more complex abstract models unnecessarily confusing and less adaptable. An advantage of this options is that it can be easily combined with other conditions (guards) for when a simulation run should end, for example due to an external event. For this reason, this option is used in some of the test packages developed in this project.

Unfortunately, the need to add many edges to an end vertex cannot be avoided by running with the `random(requirements_coverage(<int>))` option, and setting requirements on all edges or vertices where the time t is increased. This will not work, because a requirement will only be counted by AltWalker the first time that it is passed. Similarly, the simple `length<num-of-elements>` stop condition

is also not an option. It is unknown how long a test path needs to be for a given SUT simulation end time, because the SUT is stochastic.

Another solution is to force the execution to stop from the test script, instead of from the abstract model. This seems sensible, as the test script has direct access to the current simulation time of the SUT, so it can easily check if the end time is reached. However, no function has been found that will successfully end the AltWalker test case from the test script in a normal way. Calling the `tearDownModel()` or `tearDownRun()` functions from another function in the test script does nothing: the functions are not executed. An option is to force some function to throw an exception, so that AltWalker is forced to immediately execute `tearDownModel()` and end test execution. This works, but it has disadvantages: this will give the verdict 'failed' for every test case, and the coverage will not be indicated as 100% in the coverage report.

There is no method found as well to stop test execution from the test execution script, because this script cannot be 'accessed' from the test script or vice versa during test execution. AltWalker API functions like `planner.stop()` and `function.restart()` can therefore not be executed when relevant to stop a test case.

A.2. Test generation

AltWalker gives many options on how a path will be generated through the graph. A model can be run with different path generation options; this adds flexibility to the test design. A path generation option is given by a combination of a *Generator* along with a stop condition. These can be related to concepts known from the taxonomies of MBT (see Section 2.3): the generators give the test generation technologies, and the stop conditions give the test selection criteria.

The four available generators can be categorized as follows:

- *Random generation:* The `random` generator will generate a random path through the graph. If a vertex has multiple outgoing edges, an edge will simply be chosen with equal probabilities. These probabilities can be influenced with the `weighted_random` generator. For this to work, weights must have been set on (some) edges in the graph itself. Note that guards are respected by all generators.
- *Graph search algorithms:* The options `quick_random` and `a_star` try to generate the shortest path through the graph that satisfies the stop conditions. This is done with path search algorithms, respectively Dijkstra's algorithm and A*.

Thus, test generation technologies like (bounded) model checking and symbolic execution are not supported with the simple directed graphs that AltWalker uses. The stop conditions options are now related to the known test selection criteria.

- *Structural model coverage criteria:* The percentage of vertices or edges that must be covered can be set with `vertex_coverage(<percentage>)` and `edge_coverage(<percentage>)`.
- *Requirements-based coverage criteria:* The tester has options to 'steer' the path generation through certain elements. The option `requirement_coverage(<percentage>)` is used to give the percentage of requirements that must be covered. For this option to work, certain elements must be tagged with a 'requirement' field. Similarly, `dependency_edge_coverage(<threshold>)` is used to specify a threshold. Dependency values can be set on edges. The path generation will only stop if all edges with a dependency equal or greater than the threshold have been traversed. Lastly, `reached_edge(<name>)` and `reached_vertex(<name>)` are used to specify a specific element that must be passed. These can thus be used to specify an end element for the test.

Data coverage and random generation are thus test selection criteria that are not supported out-of-the-box by AltWalker. It is review in Section 3.2.5 and the case studies if and how functionality can be added for these criteria.

Note that certain path generation options can lead to (long) paths of unpredictable length. For instance, a large graph with run with `random(edge_coverage(100))` will generate a path of different length each time, and no seed can be set for this in AltWalker. If the graph would contain a deadlock, this test generation would even go on indefinitely. Endless path generation can be prevented with stop conditions `length(<no_elements>)` or `time_duration(<seconds>)` to specify the (maximum)

number of elements or time duration to do path generation. If needed, endless path generation can also be chosen by running with `random(never)`. If a vertex is reached from which no edges are available (given the guards), then AltWalker will stop path generation and mark the test as 'failed'. It is further explored in Appendix A.1.2 how SUT simulation runs of fixed length can be tested with AltWalker.

An example of a simple directed graph and its features is given in Section 3.2.3.

A.3. Test execution and APIs

The file structure used by AltWalker and the general test execution process are given in Section 3.2.4. This appendix gives further details on how the AltWalker API and GraphWalker REST API are used, and which problems are known with these. For a full understanding the reader is referred again to the documentation, see *"API documentation"* (2023b) .

A.3.1. Executing test cases from a Python script

The term *test execution script* has been introduced for a Python script that sets up and executes multiple test cases. It uses AltWalker API functions and classes from the `altwalker` library to do so. Three classes must be initialized: a `Planner`, a `Executor` and a `Reporter`. The purposes of these classes can be understood by considering the general function of an (online) MBT package again, which was shown in Figure 2.1. The scheme of how all programs and objects are used was given in Figure 3.2.

Firstly, the `Planner` is the test path generator. It specifies which model(s) should be run and with what generator option(s). AltWalker relies on the Java application GraphWalker CLI for path generation. In online testing, AltWalker will request the next step on the path from GraphWalker if it has finished its current step. Secondly, the *Executor* functions as a test case generator and test execution tool combined. For each step on the path, the associated function from the test script will be run. The test script thus gives the mapping of instructions. The test script is also used as the adapter in this model, because it sets up the SUT's run. This can be done by simply importing the SUT as a module, since all SUTs are written in Python for this project. Lastly, one or multiple `Reporter` are used to print or export the test verdicts and coverage report. The test verdict can be displayed step-by-step: each element on the path is displayed, along with the verdict 'pass' or 'fail'.

The `Planner`, `Executor`, and `Reporter` are combined into a `Walker` class instance. This is what is ultimately used to run tests with the command `Walker.run()`. The test execution script must also contain an instance of the class `GraphWalkerClient` that sets up the GraphWalker Java process. Communication between AltWalker's `Planner` and GraphWalker is done with TCP/IP messages, so a port must be opened.

A.3.2. Options for automated testing of multiple simulation runs

As established in Section 3.2.5, it is important that multiple simulation runs with different input parameters can be tested automatically with one test package. This functionality is not included in AltWalker itself, and has been built into the test execution script. This appendix explains three approaches that have been explored. Two options that were not chosen to be used in practice are discussed first.

Batch runs from one test case run

A first option considered is to use built-in functionality of AltWalker and/or GraphWalker for batch runs. However, no such functionality exists, or it is not documented thoroughly enough. The closest functionality in AltWalker is that "configuration options" can be passed as environment variables in the CLI to a test package. It is however unclear what is meant with configuration options, and no examples have been found where this functionality is used in practice. It appears to be a method to pass data to the directed graph before running a test. That would be useful to set initial parameters for the direct graph, and to pass the same parameters to the SUT.

Another option is to test a sequence of model runs using one test case. The input parameters for all runs are then contained or passed to the abstract model. Extra elements should be added to the abstract model, which indicate that a model run has been finished, and that the graph variables should be reset and a new model run should be started. This is a feasible option, but it has some disadvantages regarding practicality. The abstract model becomes more complex: it is then no longer a behavior model of the SUT, but it contains information about the test environment itself. Coverage reports will no longer relate to one simulation run, but to multiple, making them harder to interpret.

Furthermore, faults found in the SUT or errors in the abstract model will stop test execution. This would mean that subsequent simulation runs are not tested automatically, when a fault occurs before. And lastly, with this option it will no longer be possible to extend the test package's functionality with parallel execution of test cases in the future.

Options for automated batch runs from a test execution script
The option used in this project is to declare the experimental set-up in the test execution script, without using AltWalker functions. The abstract model is reset from the test execution script for each simulation run that is tested. Input parameters can be sampled by the test execution script, and passed to both the abstract model(s) and simulation run. For running replications, seeds can be passed to the SUT. This method alleviates all problems mentioned before of using one test case run to test multiple simulation runs: the coverage reports and faults are now given per simulation run and are thus easy to interpret, the abstract model remains a simple behavior model of the SUT, and parallel execution of test runs could in theory be implemented.

A disadvantage is that the test package now needs additional analysis functions that save and compare the output from multiple runs. Each test case run will still give the usual test results made with AltWalker: warnings throughout the run and a coverage report at the end of the run. However, additional analysis is needed the verify the results from multiple runs. Mostly statistical tests will be used for this.

A challenge for implementing this option is that both the AltWalker objects and SUT must be reset. Additionally, the GraphWalker process could be reset as well, or a new process must be started. Additionally, input parameters must somehow be passed from the test execution script to the abstract model(s). Various attempt are developed to achieve this, since no similar implementation has been found in previous literature or AltWalker code examples.

The general method developed for the test execution script is best explained with the simplified code, see Listing A.1. Options for the pseudo functions named are given further on, specifically for starting a new test case run, and for passing input parameters values to the test model and SUT.

Listing A.1: Pseudecode for sampling input parameters and making test cases per SUT run from the test execution script

```
gw_client = GraphWalkerClient(...)
gw_service = GraphWalkerService(...)

planner = OnlinePlanner(client=gw_client, service=gw_service)
executor = create_executor(...)
reporter = ClickReporter()
walker = Walker(planner, executor, reporter)

num_experiments = 3
num_replications = 5
input_data_dict = {'value1' : [1,2,3],
  'value2' : [1,2,3]}

for i in range(num_experiments):
for j in range(num_replications):
startNewRun()
passParametersToModel()
walker.run()
collectData()
```

Starting a new test case run The function `startNewRun()` in Listing A.1 indicates that the AltWalker run should somehow be reset once a test case has completed and a coverage report has been made. Two viable options have been developed to do this functionality. The first option will keep the GraphWalker Java process running. It uses the `restart()` and `reset()` options from the AltWalker REST API:

Listing A.2: Reset AltWalker run but keep GraphWalker Java process running.

```
for i in range(num_experiments):
for j in range(num_replications):
[...]
```

```
walker.run()
planner.restart()
executor.reset()
```

The second option will make a new GraphWalker Java process for each test case that is run. This is simply done by declaring all objects again for each experiment. The Java process of each new GraphWalker service can be closed with the `kill()` function. The implementation is as follows:

Listing A.3: Reset AltWalker run and make a new GraphWalker process.

```
for i in range(num_experiments):
for j in range(num_replications):
gw_client = GraphWalkerClient(...)
gw_service = GraphWalkerService(...)
planner = OnlinePlanner(client=gw_client, service=gw_service)
executor = create_executor(...)
reporter = ClickReporter()
walker = Walker(planner, executor, reporter)
[...]
walker.run()
gw_service.kill()
```

An advantage of the second option is that it could be adapted to allow for parallel execution, because separate GraphWalker services are used. The current implementation is unfortunately still sequential, because the next iteration of this loop will only be executed once `walker.run()` has finished. Note that while the service can be killed, the TCP/IP ports used by the client cannot be killed. This problem is discussed in Section A.3.3.

Passing parameters to the test model and SUT Two working options are made as well to pass (input) parameters from the test execution script to the test model(s) and SUT. AltWalker's aforementioned `data` functionality can be used again to pass parameters to the abstract model. This is a method to give initial values for graph variables if needed:

Listing A.4: Function in test execution script for passing input parameters to the abstract model.

```
for i in range(num_experiments):
for j in range(num_replications):
[...]
for key, value in input_data_dict.items():
planner.set_data(key=key, value=value[i])
walker.run()
```

The problem still remains on how to pass variables to the SUT. As mentioned earlier, the test script contains instructions that are executed on the SUT. Because all SUTs are made as Python scripts in this projects, the instructions are run on the SUT by simply loading the SUT as a module in the test script. Thus, the SUT and test script can be regarded as one program. The problem can therefore be rephrased as: how can data be passed from the test execution script to the SUT? Unfortunately, there is no direct connection between these two programs. A solution for this first method is to pass any data for the SUT first to the abstract model. This data can be saved in graph variables that remain unused by the abstract model. The test script can then access these graph variables with the `data` functionality and use them to initialize the SUT in the `setUpRun` function.

It is crucial that the SUT is reset as well from the test script. This is can be done with the `reload()` function from the `importlib` library. An example is given for the test script:

Listing A.5: Function in test script for loading input parameters from the abstract model.

```
import importlib
import SUT_script as SUT
[...]

class ModelName(unittest.TestCase):
def setUpModel(self, data):
# Load parameters from abstract model
self.seed = int(data['seed'])
self.val1 = int(data['val1'])
self.val2 = int(data['val2'])
```

```
# Reset SUT
importlib.reload(SUT)
# Make SUT objects with new parameters
[...]
```

The second option works the other way around. The test execution script exports the input parameter data to a JSON file. This file is loaded into the test script during the `setUpRun` or `setUpModel` function. The input parameters that are relevant to the abstract model are passed to the abstract model using the `data` functionality. The data is passed to the SUT by either loading the data JSON in the SUT script or from the test script. This method is implemented as follows:

Listing A.6: Function in test execution script for passing parameters to the test script directly via a JSON file.

```
import json
[...]

for i in range(num_experiments):
for j in range(num_replications):
# Get parameters for this experiment
params = {'val1': input_params_dict['val1'][i],
  'val2': input_params_dict['val2'][i],
  'seed': seeds[j]
}

with open(' SUT_settings.json', 'w') as f:
json.dump(params, f)
walker.run()
```

A clear disadvantage of the first method is that the abstract model must be changed: the variables to be passed to the SUT must be initialized as graph variables. An advantage of the first option is that it may be easier to adapt for parallel execution: there is no JSON data file that can be overwritten.

A.3.3. Problems with communication between AltWalker and GraphWalker

A problem with AltWalker was encountered while developing test packages for the case studies. The problem is that GraphWalker stops path generation and throws exceptions when test packages are run that have use long SUT simulation times and/or many experiments. It is found that this problem is caused by how communication is done between AltWalker and GraphWalker, which causes the computer to run out of TCP/IP ports.

Diagnosis of the problem

More specifically, a GraphWalker service is run in a Java process and uses one port. AltWalker's `Planner` class is a client that communicates with this service using TCP/IP request. Upon inspection, it is found that AltWalker will create a new port for every request made to the GraphWalker service. This means that if many requests are made in a short time, the system will eventually run out of ports. This problem occurs both when experiments are run using one GraphWalker service, or when a new GraphWalker service is started and closed for each experiment. It is unknown whether this is a bug; a more logical method seems to be using one port for all requests.

This problem is hard to solve. Declaring another IP address for the service does not work: the problem is with the client, and that will always use address `127.0.0.1`. The function `GraphWalkerService.kill()` is successful in closing the Java process, but this will not close the ports as well. The ports cannot simply be closed by closing another process or with system commands because they are in `TIME_WAIT` state. Microsoft Windows, the operating system used for this project, assigns a special process ID to ports in this state, that is different from the IDs of the associated Java and Python processes. It does not allow these ports to be closed manually. `TIME-WAIT` ports will automatically be closed a few minutes after the last message in all operating systems.

Temporary solution in test packages

Due to time constraints no true solution is developed. The problem is mitigated for now by pausing the execution in Python periodically for two minutes, so that all TCP/IP ports in `TIME-WAITING` state will be closed automatically. A convenient implementation of this in a test package, is to introduce the function

`time.sleep(120)` in the test execution script after each experiment has finished. Test package B will however run out of ports during the first experiment already. This is because it has an experimental set-up with high simulation end times, and the SUT *M/M/1 queue* needs many simulation steps, thus much communication between AltWalker and GraphWalker is done.

It is found that after approximately 1800 time advances of *M/M/1 queue* in a test case, the system runs out of ports. The simulated time is then almost 500 s. With some safety factor, the `sleep()` function is executed every 1000 time advances by the test script. A count of time advances is kept across different AltWalker runs by the test script. If a simulation run has ended, the number of time advances since the last pause are passed to the test execution script. The test execution script can then pass it to the AltWalker run of the next experiment as a graph variable.