Julia Lesky, BSc

# Test Data Generation for RFID

**MASTER'S THESIS**

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Computer Science

submitted to

**Graz University of Technology**

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Institute for Software Technology

Graz, May 2016

## AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

# Abstract

Testing is an important, quality-assuring factor in the software development process. Especially when it comes to highly heterogeneous, distributed systems, components often cannot be tested without test data fitting to the tester's need. However, in these systems test data has to fulfill numerous specialized requirements to bring the system into a desired state and therefore data is generated manually, which is time-consuming, error-prone and non-reproducible. In this master's thesis, the problems of manual data generation for various test cases in the challenging field of RFID supply chain management, one prime example for a distributed system with heterogeneous components, are discussed. The general challenges of the distributed system architecture as well as the goals of testing in this field of application are presented. To overcome problems of manual data generation, two data generation models are introduced: A sequential model and a stochastic model. Both models are evaluated regarding their usefulness for two different use cases for supply chain management. First, to bring the system into a desired state and second, for the generation of massive amounts of data for various performance tests. To test their usability and suitability, an implementation of both models is applied to a deployed RFID system for a realistic use case scenario.

# Abstract

Testen ist ein wichtiger Qualitätssicherungsaspekt in jedem Softwareentwicklungsprozess. Vor allem im Bereich von vielseitigen, verteilten Systemen können einzelne Komponenten oft nicht ohne passende Testdaten getestet werden. Gerade in diesem Anwendungsfall müssen Testdaten oft verschiedenartige Spezifikationen erfüllen, um das System in einen gewünschten Zustand zu bringen, daher werden Testdaten manuell generiert. Dies ist jedoch zeitaufwändig, fehleranfällig und häufig nicht reproduzierbar. In dieser Masterarbeit werden die Probleme von manueller Datengenerierung für verschiedene Testfälle im anspruchsvollen Gebiet von RFID-Lieferketten-Management vorgestellt, einem Paradebeispiel für eine verteilte Systemarchitektur mit vielfältigen Komponenten. Die allgemeinen Herausforderungen von verteilten Systemen sowie die Ziele des Testens in diesem Bereich werden aufgezeigt. Um die erwähnten Probleme von manueller Datengenerierung zu überwinden, werden zwei Modelle zur automatisierten Datengenerierung präsentiert, ein sequentielles Modell und ein stochastisches Modell. Beide werden anhand ihrer Verwendbarkeit für zwei Anwendungsgebiete analysiert: um das System in einen bestimmten Zustand zu bringen und zur Generierung von großen Datenmengen für Last- und Performancetests. Um die Eignung und Benutzerfreundlichkeit in einer realistischen Umgebung auszuwerten, werden Implementationen beider Modelle in ein laufendes RFID-System integriert.

# Acknowledgments

First, I would like to express my gratitude to my supervisor Prof. Wotawa, for all the valuable feedback and guidance during the work on this master's thesis.

Thanks also to Michael Goller, who made this work in cooperation with Enso Detego GmbH possible. Thank you for all the support, whether it was an insightful discussion, constructive criticism or proofreading. I would also like to thank all my colleagues for their understanding and patience during stressful times - you're an amazing team and it's always a pleasure to work with you!

I would like to thank my mother Sabine and my grandparents, Harald and Herta, for their guidance and support throughout my whole life. Without you, my university career wouldn't have been possible and I wouldn't have achieved what I've done until now. Thank you for your unconditional love and encouragement whenever I need it.

Furthermore I want to thank my best friend Christina, for being here for me since more than twenty years. You're the best friend one can imagine and I'm grateful fo your understanding and help in all aspects of my life.

# Contents

# List of Figures

# 1. Introduction

This master's thesis aims to discuss the challenges highly heterogeneous, distributed systems face when it comes to the need of test data. Test data generation is a complex process which needs much of the testing time if done manually. This is why it is important to establish ways to generate test data automatically. In this work a model-driven test data generation framework for distributed software systems with applications in supply chain management is presented.

## 1.1. Motivation

Software testing is a very important, quality-assuring factor in every field of information technology. Yet testing is still expensive and time-consuming if done manually. S. Dustdar and S. Haslinger [11] even state that testing software needs an average of 40% to 85% of the whole development process. To reduce time and costs, testing techniques suitable for the specific considered field of application need to be found. One possible way to minimize the effort is test automation with known techniques, such as regression tests, code-driven testing or graphical user interface (GUI) testing, to name a few examples.

The mentioned factors and the complexity depend heavily on the considered system architecture and required system components. Especially distributed systems are a challenging field of application due to their heterogeneous architecture with different abstraction layers, such as front-end, back-end, different types of hardware and software, all of these components communicating with each other. RFID (Radio Frequency Identification) systems are

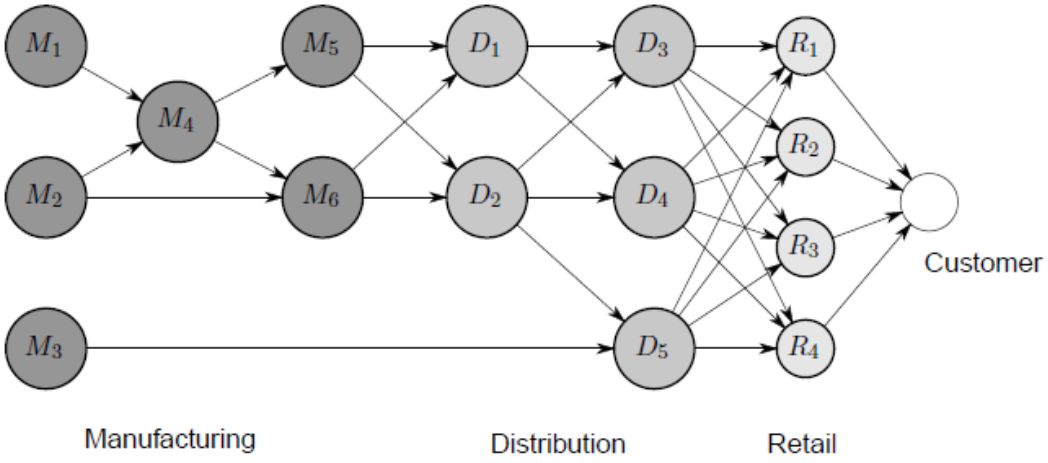


Figure 1.1.: Example for a typical supply chain in the fashion industry by M. Goller [18]. A supply chain consists of multiple connected nodes representing various stages, such as manufacturers, distributors and stores, with the customer as endpoint.

one prime example for a complex, multi-layered and distributed architecture with a close hardware integration, with their main characteristic being the contactless object identification over radio waves. The peculiarities of systems using this technology add even more complexity to the system, for example by unforeseeable behavior due to environmental influences during the identification process.

One field of application for such a distributed architecture is supply chain management. In Figure 1.1, an example for a supply chain is shown, spanning over manufacturers, distribution centers and stores. Additionally, it is possible that a system could also span only over parts of the supply chain. This could mean that RFID is only used in DC and stores and not at the manufacturers. Regarding testing and test automation, this makes the scaling of test data difficult.

Considering the mentioned factors, such as the different system layers and the functional principle, test automation in this field faces several challenges compared to traditional software systems, such as:

- **Distributed system issues:** As a sample for a distributed architecture, RFID systems face the same testing challenges, such as comprehensive testing rather than only tests of the single components, or tests to ensure correct system behavior in aspects of heterogeneity, scalability or failure handling. The topic of challenges and design requirements will be evaluated in more detail in Section 2.1.
- **Environmental issues:** When testing an RFID system, the tester needs to consider the peculiarities of the contactless identification of objects with applied tags/transponders regarding the environment. The main issues here are reflections by different media or the distance between tag and reading device. When test automation should cover aspects such as reading performance or detection rates, thoughts must be spared on how to get a test environment as stable as possible.

While the problem of test automation for individual RFID system components was already discussed by C. Pichler [31], it is not always possible to test particular components without additional test data. Typically, test data is generated manually by means of simple, deterministic models that cover the most important aspects of the investigated software module. A realistic scenario to show the problems caused by the complexity of distributed systems is their usage in the fashion industry. The system is used to monitor the whole supply chain of a company. Such a supply chain starts with the items being created at the manufacturer and covers typical processes of logistics and retail in this sector, meaning items leaving the manufacturer and being sent to a distribution center (DC) for further distribution to retailers (stores). Processes in a distribution center include the sorting and packaging of items for the redistribution or the sending process to the retailers. When items arrive at the retailer, they are added to the store's stock (goods in) and are involved in processes such as stock takings, movements on the store area or sales. In all these processes, RFID hardware is used to detect and identify the single items and the corresponding data is processed by the underlying software. Identifiers (in this industry EPCs, Electronic Product Codes) and article data of tagged items are stored in a database, and the procedures are performed with application software using the RFID hardware.

The following example will outline the complexity of the manual test data modeling. For every use case and functionality to test, a suitable data set

has to be created. At all stages of the supply chain, various processes are performed, for example the typical sale in a store. Each of these processes causes data transactions in the back-end and has influence to one or more parts of the supply chain. Even for a small test case scenario, different use cases have to be considered and considerable large data sets have to be generated to get expressive results.

## 1.2. Problem Definition

To test a supply chain management system, it is essential to know which data is needed to perform certain tests and to bring the system into the desired state at any time. As testing is a time-consuming factor in the development process, test automation and methods to establish fixed, reusable test routines are very important techniques to ensure a fluid test process. However, even with a high test coverage due to automated tests and suitable test environments, manual testing still has to be done to a certain extent. Whether the performance and manageable data load of a system should be tested or just to bring the system into a certain state for one particular test case, data should be at hand as quickly as possible.

In this master's thesis, we will discuss the requirements and challenges of testing distributed systems, such as heterogeneity, concurrency or scalability, with the focus on test data generation for aforementioned problems. We present a model-based test data generation framework for application in distributed, heterogeneous software systems in supply chain management. The developed model utilizes empirical data from productive systems for the estimation of model parameters and the model evaluation with respect to various functional and non-functional system tests .

With the model application, it should be possible to simulate supply chain processes in the most possible realistic way. Examples for the usage of a data generation model are the simulation of a sequence of consecutive store processes to either bring a sample test environment into a desired state, or to apply performance or load tests onto the system. Before the actual modeling process, the characteristics and properties the model should have must be defined. This means one must be aware which test types are covered with

the data created by the model and where the limits are. In this work, the focus lies on two aspects:

- The tester wants to bring a test environment into a specified state to perform one particular test case or needs to verify wrong behavior of the system. The application of the data generation model should fill the test system with a sequence of actions and processes over a fixed amount of time.
- To evaluate the capacity and performance of a system and find possible bottlenecks or other wrong behavior when confronted with a large amount of data, the tester wants to fill the system with a reasonable amount of realistic data. Rather than performing all the actions manually to get large amounts of data into the system, the model should provide an easy and fast way to apply these tests.

The goal of this thesis is to create a generative data model to cover the previously mentioned aspects. To achieve this, different modeling approaches, such as deterministic or stochastic methods, as well as simulation frameworks or engines with focus on their usability for heterogeneous, distributed systems will be evaluated and compared regarding their benefits and drawbacks for test data generation for this challenging field of application. The created model should cover a reasonable complexity and number of parameters to ensure easy application and integration into supply chain management systems.

In a following step, the model will be implemented and included into a framework so that it can be used in real test environments for RFID supply chain management systems. The application should provide interfaces for an easy integration into the systems as well as a simple user interface to set the parameters which the data model should use when the test data generation process is executed.

As a practical part, the developed modeling framework is applied to an existing software solution with application in the fashion retail supply chain management with particular focus on the processes on the store level, detego SUITE [17]. The usability of the created data model will be evaluated within this productive environment containing software components, such as database, web service and different types of application software (one of them running on a mobile RFID device), and interfaces for different types

of stationary and mobile RFID reading and printing devices. The effort of automatic test data creation will be compared to manual test data creation to get insight to the model's applicability to test various aspects of supply chain management systems.

## 1.3. Overview

In Chapter 2 we will discuss the properties of the system to test, starting with challenges and requirements of distributed systems. The focus lies on how these challenges and requirements can be covered with the created data generation model. After that, we will go into more detail and specify which aspects are important for an RFID system, a special type of distributed system. This will also cover a general overview about what RFID systems are as well as peculiarities different from other distributed systems. In the last part of Chapter 2 the model requirements will be specified more clearly. This section discusses which previously presented requirements or challenges can be tackled with the created data generation model. This also covers functional and non-functional model requirements, such as the test types for which the model should be able to generate data as well as non-functional requirements, such as scalability or performance.

In Chapter 3 the modeling is presented. The first part describes how a supply chain management system works in general and presents exemplary process steps on the store level of such a supply chain. Related work shows the state of the art regarding model-based data generation with their advantages and limitations of test data generation in the special case of supply chain management. The modeling process covers two different models. The sequential model is a simple-to-use model based on the lifecycle of a single item in a supply chain. As this model has some limitations, a more sophisticated, stochastic model is presented, which generates test data based on empirical data taken from a productive supply chain management system.

Chapter 4 describes the implementation details of the two developed modeling approaches presented in Chapter 3. The functionalities as well as the

usage is described, as are implementational characteristics of the sequential and the stochastic model.

The application of the designed models and implemented framework on a deployed supply chain management system with focus on the store layer of a supply chain is presented in Chapter 5. The interactions of the framework and models with the deployed system will be described. We will evaluate the model framework's execution time compared to manual test data generation as well as the quality of the generated data. The evaluation also covers the application to different use cases, such as data generation for particular test cases or for performance tests. In addition, the usability and adaptability of the model framework in different use cases is discussed.

This work concludes with Chapter 6, where all important aspects of the previous chapters will be highlighted again for a final summary. This covers advantages of test data generation with the help of a model as well as limitations of the used models and an outlook for future work.

# 2. Prerequisites

This chapter discusses the various requirements a generative data model should fulfill in a distributed environment. We will start with a short overview about general properties and challenges of distributed systems in Section 2.1 with the focus on the required data to effectively test these systems. In Section 2.2, a more detailed of additional peculiarities of RFID systems will be given. Finally, Section 2.3 discusses the requirements the model to design should fulfill in the given environment.

## 2.1. Distributed Systems

To line out the test data requirements of a distributed system, this section covers details on the properties and challenges of distributed systems. Tanenbaum and van Steen [35] describe a distributed system as a collection of independent computers which appear as a single system to its users. This means a distributed system contains a various number of applications on different computers, possibly devices of different types, to perform its tasks by communication via message exchange. The concept of a distributed system spreading over a network faces a few challenges.

Coulouris et al. [8, Ch. 1.1, p.3] describe the most important issues of a distributed system compared to a single system as the ability for resource sharing between concurrently executed processes, the need to synchronize the clocks of different components due to the lack of a global clock and failures of single components. Including a few thoughts on data modeling, the most important challenges faced when working with distributed systems are discussed as follows (see [8, Ch. 1.4] for more details):

- *Heterogeneity*: Distributed systems consist of various different hardware and software types. Communication between various devices, operating systems and pieces of software has to be ensured. Regarding testing, it should be possible to create considerable large test data which can be used to test the different system components.
- *Concurrency*: In a distributed system, resource sharing between concurrently executed programs must be ensured to prevent blocked processes because of resources locked for only one program. Test data must be designed in a way to be able to check if resource sharing works correctly. In addition, awareness of the limited ability to synchronize clocks between devices and other components of the distributed systems is needed for correctly timed message exchange.
- *Scalability*: A system is scalable if it still works stable and efficiently even with a great increase of resources and users accessing components. Regarding test data creation, this means with the model and a later framework it should be possible to create a large amount of data to test the scalability and to detect performance losses and bottlenecks in a distributed system.
  Performance is a challenge for every system, more so if it is heterogeneous and distributed. Users expect the system to be responsive in every situation, regardless the performance and load of a network or the size of data to be transmitted. The differences in response time can be seen with the example of access to a cached page (fast) or a not cached image of larger data size (slower) from [8, Ch. 2.2.5, pp.43-44]. In addition to responsiveness, two other performance aspects are throughput (the amount of work processed over an amount of time) and the ability to handle computational loads (for example by distributing one process over more than one component for load balancing). The system should be able to perform its tasks for all users accessing it at the same time. One way to react to performance issues is the *usage of caching and replication techniques* to achieve better responsiveness and a smaller computational load on the system.
- *Openness*: Openness is an important factor for distributed systems to ensure correct system behavior when a system is expanded by new components. Also a created data generation model should be able to be expanded by newly introduced processes at any time.

A few other challenges are listed as well. These are for once the topic of *failure handling*, which is the ability of a distributed system to handle faulty behavior or failures of a system component by introducing sensible failure policies. Another point is *transparency*, meaning that the distributed system must act as one entity to the user. The last challenge is *security*, where access rights, especially for sensible data, needs to be handled correctly. However, these challenges are not relevant for this work and will not be discussed in more detail.

## 2.2. RFID Systems

Radio Frequency Identification (RFID) systems are one special example for a distributed architecture. Therefore, they also share many of the previously presented challenges and architectural requirements. In this section a general overview of RFID technology will be given.

The main concept of RFID is contactless unique identification of objects with radio waves. The two main components of an RFID system are the *transponder* (also called tag) and the interrogator, commonly referred to as *reader*. The transponder is a device, usually without its own power supply, which holds data relevant to the object it is placed on. While various types of transponders exist, their main elements are a coupling element and a microchip. The minimum elements of a reader are a radio frequency module which acts as transmitter and receiver, a control unit and a coupling element (antenna). However, many of them have additional interfaces to be coupled with other systems to forward data received from transponders. In order to collect data from the transponder when it comes into the interrogation zone of the reader, the transponder has to be activated. The required power is supplied contactless through the coupling units on reader and transponder [12, Ch. 1.3].

Although the transponder and reader can be seen as a small RFID system, a setup of only these two components is of little practical use. An appropriate description of the components needed for a productive RFID system is shown in Figure 2.1.

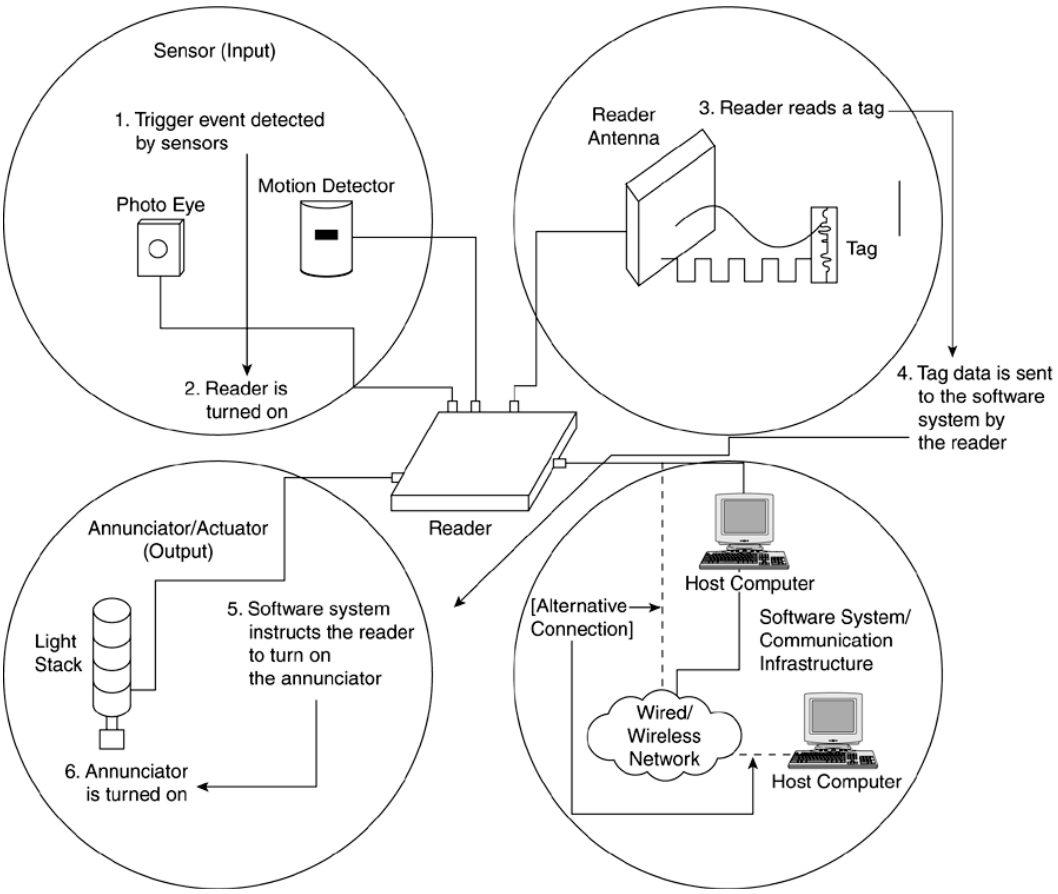


Figure 2.1.: Sample components of an RFID system by S. Lahiri [26]. This image describes the main parts a typical RFID system consists of, which are a computer system, RFID reading units, tags as well as possible sensors (inputs triggering the RFID elements) and annunciators (output produced from the RFID elements).

Not only can an RFID system consist of more than one of the mentioned tags, readers and sensors, especially in large systems the underlying software system typically consists of a set of distributed host devices and a multitude of software applications. Therefore, an RFID system qualifies as a distributed system with heterogeneous components. Considering the particular application in supply chain management, a software system typically spans over various manufacturers, distribution centers and stores, all of them equipped with an appropriate infrastructure to cover the relevant processes at the particular stage. For a smooth work flow, communication between all components must be ensured. To achieve that, the previously listed general challenges and requirements for distributed systems can also be applied on RFID technology and the testing process, as they share the peculiarities of distributed systems.

While these are probably the most important points regarding the productive software of an RFID system, there are a few other things to consider in a productive RFID environment, which are for one different environmental influences regarding the main RFID components, and second, interfaces to other systems, for example external ERP systems. When using RFID technology, one must be aware that various things can influence the behavior of the reader-tag interaction. Not only the distance between reader and transponder is an important factor for the behavior, where the detection rate decreases with increasing reader-to-transponder distance, but also the wave transport media (air/water) or reflecting/blocking elements in the surroundings of the stationed RFID technology.

As we will focus on the challenging distributed and heterogeneous aspects on the software side of RFID systems to reduce the need of RFID devices for data generation, the technical details of RFID hardware components, for example frequency ranges for various usage fields or transponder types, will not be discussed in detail. Refer to the RFID Handbook [12] or the RFID Sourcebook [26] for more information on the physical aspects of RFID technology.

## 2.3. Model Requirements

In this section, the requirements of the data generation model will be described. Basically, requirements can be divided into two two types: *functional* and *non-functional* requirements. Functional requirements describe how well the model meets the specification, meaning what the model does, in contrast to what it is supposed to do. In this work, the main functional requirement of the model is that test data should be generated according to the test definition, which will be discussed in Section 2.3.1. The functional requirements also cover the test types where data should be generated for with the created model.

Non-functional requirements are other factors that should be fulfilled, regardless of the defined functionality. Examples for non-functional requirements are a certain degree of scalability or usability of an application or framework. These and another two important aspects, controllability and observability, are discussed in the following sections. How well the model framework fulfills the listed requirements is evaluated during the usability study in Chapter 5.

### 2.3.1. Functional Requirements

In the best case, a data generation model should be versatile enough fulfill the data needs of a wide range of test types. However, as this would be out of the scope of this work, we will concentrate on a few selected test types and challenges for which test data should be generated. Basically, every single piece of software needs a certain amount of data to work with. While it can be sufficient for simple applications to cover all possible test cases and scenarios with unit and/or module tests with simple and easy-to-create data types as parameters, this is not always the case.

Already when looking at the simple example presented in Myers' *The Art of Software Testing* [30, Ch.1], one can see that even for this very simple application to verify if three input parameters build a valid triangle a quite comprehensive data set is needed to cover each test scenario. From that we can assume that the effort to think of meaningful test data sets to use as

either input data, calculation parameters or to fulfill any other test need grows exponentially the larger an application or system to test is.

According to [15], data input generation for testing can be divided into two sections: test sequence generation and test data generation. This work explains test sequence generation as dealing with what method sequences to test, test data generation on how to generate the data for these methods. With the data generation model, we aim to do both. Actions in the supply chain are methods to test in the software. Sequences of these actions should be generated and performed accordingly to test workflows and we want to generate meaningful input data to execute these methods with.

We can distinguish the following test types the data generated by the model should cover: generate data to bring the system into a particular state to test one component or method of the system, whether it is only a single action for an encapsulated test or a sequence of actions to simulate system behavior over a certain amount of time, or to generate data loads for stress and performance tests on the underlying system. The system under test SUT will act as a black box, meaning based on given input we will examine the output of the system without knowing anything about the internal system structures (see [30, Ch.2] for details). The test types are described in more detail as follows:

**Data generation for particular test cases:** Basically, when we want to bring the system into a desired state, this means that test data should be created in order to be able to perform one particular test case. Especially in distributed systems with an underlying database, it is not always possible to perform these test cases independently from test data. In simple test cases, this can even mean that generating the data manually might take more time than verifying that the system works correctly. In this case we want to use the framework to do the work - give it some parameters to reach the desired state, start the simulation and perform the test cases as needed. The created model framework should provide ways to fill the system with the data needed for test cases, whether they are tests for one module of the system or comprehensive tests of system components interacting with each other. In the framework, it should be possible to set parameters such as the size the generated data set should have (such as the stock size), the different number or types of supply chain actions that should be performed.

One standard use case for this type of generated test data could be the testing of one module of the system (which is for example the execution of one action in the supply chain lifecycle). The tester wants to verify the correct behavior (correct output based on the data input) on the software side of the system and exclude possible faulty behavior due to the peculiarities and possible interferences when using RFID technology.

**Data generation for performance testing:** Performance testing describes testing how well a system performs under different workloads, such as the number of users accessing the system or data sent over the system. The data generation model should be used to test the system's performance under extreme cases of workload over various amounts of time to find out the behavior of the system in these cases. The topic of performance testing with usage of the data generation model is divided into two sub-topics (after [30]). In addition, the topic of endurance tests will be covered as well. The performance tests can be described as follows:

- *Load testing or volume testing*: The system is filled with large amounts of data or needs to handle an absurdly high workload at once. When performing load tests, the goals are to find the data limit the system can handle and how responsive the system still stays in case of such a high amount of data.
- *Stress testing* is the test of a system's behavior during peak times, meaning the system is confronted with a heavy workload over a short amount of time. This type of test is performed to determine whether a system still fulfills the performance specifications when facing these peak amounts of data.
- *Endurance testing* describes testing the system's behavior when it has to handle a particular workload over a long amount of time.

With the manual creation of test data, it is often difficult to get data sets to test the performance of a system, especially when it comes to stress or load tests, where the system is confronted with a large amount of data at once. When this is done manually in such a distributed system for store supply chains, this could mean that a large testing environment is needed with more than one tester, multiple RFID devices and large amounts of physical tags. In this case it can be difficult to time the actions performed by each tester accordingly in an reproducible way. In addition, due to interferences

when using RFID devices or different testing behavior of the testers, it can be difficult to control the system at every moment of time to make qualitative assumptions about the system's behavior in such a test case.

A second aspect for performance testing is that in many cases the system should not be confronted with any, possibly random data just to find the amount it can handle, but rather it should be meaningful data according to specifications of the supply chain or the lifecycle of items in the supply chain. The system's behavior should be observed with large amounts of realistic data to see how well it performs during times of extreme workload and to find the limits of data the system can handle.

### 2.3.2. Controllability and Observability

Controllability and observability are two important aspects when testing a system, more so if it is distributed over various components. It is important that the tester knows the exact state a system is at any time to verify correct system behavior or prove the existence of errors or failures. In this context, the problems of controllability and observability are discussed in various works about testing distributed systems [7, 16, 22, 36].

However, this might not be that easy to accomplish in a distributed systems. In the model to design we still want to provide the best possible controllability and observability. Summarized, the two problems are described as follows:

**Controllability:** When testing a system, the tester wants to be able to control a program's inputs, outputs, operations or behavior at any time [16]. Especially in distributed systems, when testers are situated at various parts of the system, maintaining controllability is not always possible due to the concurrent nature of these system types with the lack of global clocks. For the testers it is not always clear when to send an input with no or delayed responses from other parts of the system [22]. This means that, especially in a distributed system with inputs from different sources, the problem of time dependencies can arise if no notifications are given as to who submitted messages to the system or when new messages can be sent. As the data generation model is set upon a distributed

system, it should be tried to avoid this scenario. In an RFID supply chain management system in particular, it is very important to coordinate inputs so that no race conditions occur which would lead to data inconsistencies.

**Example:** Any two actions are performed nearly simultaneously. Assuming that the second action would reverse the first one, should this be allowed or not? Scenarios similar to this must be thought of in the system specification to define a correct workflow. However, for the data generation model controllability would mean that we can exactly track what is executed at which time, meaning the data created with performed actions, locations, items moved with actions, as this will be the input to the test system to bring it into a desired state.

**Observability:** The observability problem occurs if it is not possible for a tester to find out which input caused a certain output [36]. To overcome the observability problem, the tester wants to trace back every output to a unique input from the data generation model. In the model, there will be no possibility to force the system to produce an output as expressive as possible, it rather depends on how the system under tests presents and makes the produced data accessible.

The easiest way to provide a certain level of controllability and observability when using the data generation model in this work would be the logging of each input or calculation done. This should include timestamps and other important runtime information to trace the messages sent to the test system.

### 2.3.3. Performance and Scalability

When applications are used, the question of their performance and scalability arises. K. Wiegers and J. Beatty [37, Ch. 14, Table 14-1] describe the *performance* requirement as the time a system needs to respond to user inputs or other events, and how predictable these responses are. *Scalability* is described as the ability of the system to grow in order to handle extensions, such as more transactions, servers or users.

Generally, when talking about how well a system performs, this aims at the response and calculation times of an application under a particular,

pre-defined workload. Whether this workload is the expected amount of data and information a system or application has to handle during usage, or a larger workload over a longer amount of time or during peak-usage, the application or system should maintain a reasonable performance. The model hence should be able to generate representative, large-scale data sets to test a system's scalability or performance.

When an application or system is scalable, this means that it is able to handle an increasing workload due to a higher amount of data to process, users accessing the system or calculations to be performed with respect to the performance specification. The system qualifies as scalable if the higher workload can be handled in a time not crossing a specified threshold. In case of this work the model framework should be able to adapt to needs for large amounts of data to generate, for example for stress testing the system.

Summarized, when stress tests or other tests with the need of a large amount of data should be performed on the SUT, the user should be able to generate these data sets with the model framework in a reasonable amount of time. How well the model framework adapts to the performance and scalability requirements will be evaluated in Chapter 5 by comparing the manual effort of test data generation to data generation with usage of the model framework for extreme cases of workload.

### 2.3.4. Usability, Adaptability and Extensibility

The requirement for usability mostly applies to the model's application. Even if the complex structure of supply chain data is modeled, the model should provide means of an easy and intuitive usage and fast applicability to the system under test to the create test data. This means the model should be as powerful as possible regarding the test data generation, but should still maintain a certain degree of usability, which also covers easy adaptability and extensibility. To achieve this, the complexity with respect to the number of model parameters and general model size has to stay within a reasonable range which can be understood by the tester.

As an example, consider any data generation model which is able to create a wide range of test data for a large, distributed test environment. For a tester, it would be of little to no practical use considering the mentioned factors of usability, adaptability and extensibility, as the model probably consists of thousands of nodes, methods or calculations. Maintaining or extending such a model is practically impossible for a human tester.

For the user it should be possible to understand the data generation model's structure at any point to be able to add, remove or change parts of the model. In case of data generation for supply chain management it should be easy to change or add properties, such as actions or locations, to the model for it to be applicable to various parts of the supply chain based on the use case. In addition, the previously mentioned factors controllability and observability also increase the model's usability.

When designing a data generation model, not only the use cases for the data creation need to be defined. Additionally, also various non-functional requirements have to be considered to increase the model's quality. These could be summarized as the ability to control and observe the model during the data generation process, as well as a certain degree of performance and scalability for various types of use cases. Finally, the model should be easy to use with respect to application as well as adaption and extension to different parts of the supply chain.

# 3. Modeling

For the test data modeling, two model types were considered and evaluated regarding their usability for supply chain systems: a sequential model and a Markov model, one type of stochastic model. In the next sections, we will describe both models in more detail and discuss how they will be used to generate test data. In Section 3.1 the processes of a supply chain are described. This section covers the properties of a supply chain and introduces the relations of sites, locations, actions and items in a supply chain to describe the system's functionality to model. Section 3.2 presents related work on data-modeling, such as already existing data generation tools and popular modeling types like state machines or evolutionary algorithms. In Sections 3.3 and 3.4 we present two models which were considered to be suitable for the purpose of test data generation.

## 3.1. Process Model

In this section, the basic structure of a supply chain is described. A typical supply chain consists of three basic elements: *locations*, *items* and *actions*. *Locations* describe different areas of sites (nodes in the supply chain, such as different stores or distribution centers). Prime examples for locations are sales floors, back rooms or shop windows in a store, or different storage rooms at a distribution center. An individual *item* is one element of a product, for example a shirt. The sum of all individual items in a supply chain, spread over the locations, form the stock $I$, with length $\|I\|$ being the total number of items. On items, *actions* are performed. These are basically procedures to change or detect the physical location of an item in the store, such as stock takings or relocations, or to change the number of items in stock, for example with a sale action. In Section 3.1.1, the most important

actions describing the everyday behavior in a supply chain actions are presented.

In a supply chain, different locations on various parts of the supply chain (sites) exists, each of them containing items. The sum of all of these items form the stock $I$ over all considered sites of the supply chain, which can be defined as

$$L_1 \cup L_2 \cup ... \cup L_n = I,$$

with

$$L_1 \cap L_2 \cap .. \cap L_n = \emptyset$$

meaning that items in stock can only be present at one location at a time.

$I$ and $L_1, L_2, ..., L_n$ are represented as list of present items, containing the items' Electronic Product Codes (EPC), a unique identifier, to clearly identify each item.

**Example:** With every action/transaction in the supply chain, a subset $L_k$ of items are potentially changing their location/state. Figure 3.1 shows a simplified example within a store with two locations, back room $BR$ and sales floor $SF$. The nodes describe the mentioned locations, meaning where an item is present at one moment of time, being the empty node before it is added to the store's stock, the back room, the sales floor and the customer. The edges/transitions describe the location-changing actions from the first moment an individual item is detected in the store until it reaches its final destination, the customer ($C$).

As shown in the previous example, it is not a realistic scenario that actions are only changing the number of items on any location $L_1, L_2, ..., L_n$. It is more likely that at some point either a set of new items is introduced to add it to $I$ or any other location, or that actions cause sets of items to leave the supply chain and decrease $I$. At a moment of time $t_k$, an action $a_k$ is performed either on $I$ or on a subset of $I$ or $L_1, L_2, ..., L_n$, respectively. With that, we have a sequence of actions $a_1, ..., a_k, ..., a_n$ to simulate the activity in an supply chain during the time interval $(t_1, t_n)$.

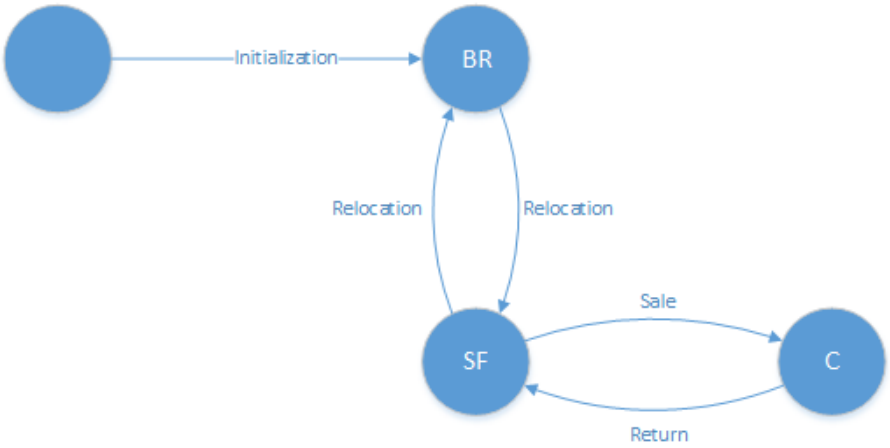


Figure 3.1.: Abstract process flow for an individual item, starting from where an item first enters the system's back room location. Possible location-changing actions are then relocations, until its final destination, the customer is reached (with the possibility to be returned to the system).

For each action, a meaningful item subset needs to be selected. In most cases, this should typically be an amount of items bound by a minimum and maximum number of items $[b_{low}, b_{up}]$ of $\|I\|$, $\|L_1\|$, $\|L_2\|$, ..., or $\|L_n\|$. We need not only to define the size of the subset, but also the items contained in the set. Both steps need further knowledge, and can, for example, be selected based on domain knowledge or by learning from empirical data analysis.

### 3.1.1. Exemplary Process Steps

In this section a few typical supply chain actions are described with focus on the store-level. These exemplary process steps are also used for the model evaluation in Chapter 5. We present the following actions in detail: initialization, stock take, relocation, inbound, outbound, write off, sale and return, which all have their own characteristics regarding the subset size and location-changing properties. The initialization, stock take, relocation and inbound are actions which can be performed on either location, while for sales only items on the sales floors are considered. Returns can only

happen for items not in the stock $I$, with a sale as last action. The outbound process is independent from where items are located.

- **Initialization:** When items arrive at a store, they must be recorded in the database and given a location. The initialization step is used to either initialize the first items in a new testing environment in a database with 0 items so far, or to simulate the arrival of new goods in a store at a given time. The number of items to initialize can either be defined manually by the framework user with the need to fill a database with items, or by selecting a random number inside predefined lower and upper bounds $[b_{low}, b_{up}]$, whether they are set based on knowledge of the application field or estimated by training the model with existing real-life data.
  An initialization action adds items to a location $L_k$ and therefore to $I$ as well and increases $\|I\|$ by the amount of items initialized.
- **Stock take:** Stock takes are used to detect the amount of items on a store's location, and can be done on any of the locations $L_1, L_2, ..., L_n$. The stock take process updates each detected item's location to the location the stock take is performed on. In an example, this means that an item with last location back room still has it as a location if it is detected at a stock take on the back room, but the location changes to sales floor if it is detected at a stock take on the sales floor.
  The goal is to have a coverage as high as possible on each floor, that is, in the best case a stock take should detect 100% of the items on a specific location. In addition, it is also possible that items from other locations are detected - this could be the case if the are placed on a location without the proper location-changing action, or due to peculiarities regarding the RFID interrogation zone of a reader (shielding, reflection). As an example, we define the number of items for a stock take $ST$ on a location $L_1$ in a store with two locations $L_1$ and $L_2$ as

  $$ST = p \cdot \|L_1\| + q \cdot \|L_2\|$$

  where $p$ and $q$ are selected from the percentage interval for the number of items to select from each floor. The exact determination of the intervals for $p$ and $q$ as well as the selection technique for the items is described in detail for each approach in the corresponding sections.

However, it still can be said that the number to detect from the location other than the stock take's should be modeled as small as possible, as only a small amount of items, mostly located in the border area between locations, will be detected.

- **Relocation:** A relocation is performed when an item is carried from one location to another, meaning that the action changes the item's location to the one it is relocated to. When an item is relocated to the same location it already was registered on, the action is still performed. After the relocation, the item will still have the same location, but has the relocation as its last action. The number of items for a relocation *Rel* is is calculated with

$$Rel = p \cdot \|L_k\|$$

with the percentage interval $(p_{low}, p_{up})$ forming boundaries for the number of items to select. Typically, this is a rather small number of items, as with a relocation only a few individual items are carried from one location to another.k
- **Sale:** For a sale, only items on sales floors are available to a customer. This is the reason why only items from the sales floors are selected for the *Sale* action. Again, in comparison to the number of items present on the sales floor, the amount of items in the typical shopping basket of a customer is rather small. More details on how this amount is selected can be found later depending on the approach.
Each sale decreases $\|L_k\|$ by the number of sold items.
- **Return:** Only items already sold qualify for a return, and only a small amount of items will be returned and added to stock $I$ again. This amount is determined, depending on the approach, either by setting it to a small amount of only a few sold items, or from an empirical data analysis.
With every return, $\|I\|$ is increased by the amount of items returned.
- **Write Off:** From a process point of view, there are numerous reasons when particular items need to be removed from the stock in a given location. The write off process provides this functionality and depending on the use case, a selected number of items is removed from the stock $I$.
- **Inbound:** An inbound or goods in process is performed when new

items are delivered to a store by either the manufacturer, DC or another store. With an inbound process, the delivered items are added to the store's stock system. Usually, a store is informed of the arrival of new goods beforehand. This means that a delivery notice containing the advised items and their quantity per product is added to the store system before these items arrive at the store.
After their arrival, they need to be added to the store's stock. This is done using the delivery notice to check which items actually were delivered and which were possibly missing. For the inbound process this means that the number of items to add corresponding to the number in the list $d$ can be defined as

$$In = p \cdot \|d\|,$$

with $p > 0$ being the percentage of the actually delivered items.

- **Outbound:** When items leave the store to be transferred back to a distribution center or the supplier, this is done with an outbound or goods out process. Each item in stock $I$ potentially qualifies for an outbound process, and usually cartons containing more than one item leave the store. The number of items for an outbound depends on the store's size and can be formulated similar to the relocation process as

$$Out = p \cdot \|I\|$$

with the percentage interval $(p_{low}, p_{up})$ forming boundaries for the number of items to select from the stock.

## 3.2. Related Work

To find the most suitable model, research was done on how the topic of test data generation can be approached and which model types and simulation frameworks to create test data already exist. In this section, the results of the literature research are presented. This will cover a general overview about selected model types as well as an evaluation of advantages, drawbacks and limitations of already existing modeling approaches regarding the intended purpose of the model to design. The most remarkable test data generation types and models are described in the next sections.

### 3.2.1. Existing Tools

There exist quite a few test data generation tools, which create test data according to specifications or by code analysis. To name examples, Doungsaard et al. [6] present a framework to generate test data from software specifications, namely UML diagrams, and S.J. Galler and B.K. Aichernig present an overview about popular test data generation tools for various programming languages [14]. However, more research into the topic of existing frameworks and tools showed two main problems for their suitability for this work.

One problem is the dependency on a certain programming language of most of the tools described in [14]. While this work also covers the implementation of the model and integration into a running supply chain system's test environment, the sole usage of one of these tools would not be sufficient. In addition, most tools need to be integrated or set upon existing source code to analyze methods and workflows for test data generation depending on the specific system under test.

The second problem is the need of a program or workflow specification to generate test data. The software to test is based on specifications and defined workflows. Yet the model to design in this work should simulate a supply chain in general for various test data needs, independent from one specific system description in form of workflow diagrams, for example to infuse erroneous data contradicting this workflow. This and the need for program code are the reasons why none of the frameworks found in various works during the research process was taken into consideration.

Additionally, not much research work or presented frameworks are available, as supply chain management with the usage of RFID technology is a quite new field of application. Thus, only a few different systems for this special type of distributed application exist. Testers of these systems still need to rely on specialized test methods. Tools directly developed for the peculiarities are used to assure the quality and confidence of their products. This is why more general approaches must be evaluated regarding their usability for this technology.

### 3.2.2. State Machines

State machines are one prime example covered in current publications on model-based testing of distributed systems and test data generation. The distributed system behavior is often described with a collection of finite state machines for test case generation, for example in [19, 21]. When looking at Figure 3.1 and at the definition of a state machine as a set of $n$ states and $m$ state transitions (a more detailed explanation can be found in [21, Ch.2.2]), we can see that a single item's movement cycle in a store could be easily modeled as a state machine, where the states are the locations an item could be on and the transitions are the actions with which the location is changed. This is why an abstract state machine approach was taken for one of the presented models in Section 3.3, where store processes are modeled with simple state transitions.

However, this model has a few limitations due to the static structure of states and transitions, which need to be designed manually based on workflows from specifications or expert knowledge. This limits the flexibility of the model regarding test coverage, as a generalized definition of states and transitions in addition to a specified workflow makes this model type too complex for sensible user interaction.

### 3.2.3. Probabilistic Models

Probabilistic or stochastic models are a popular way for test generation and are for example shown in [4, 40], as a probabilistic model is a flexible, powerful model type for this field of application. In this context, especially Markov chains are considered, one prime example for a stochastic model, where transition probabilities between system states at observed times are used to simulate system behavior. In contrast to other approaches, for example state machines, a training process with realistic data can be applied to this model type to estimate the transition probabilities between states. This adds flexibility regarding the application of the data generation model, as such a model is easily extensible and can be varied to cover unlikely cases as well for a high test coverage. In Section 3.4, a stochastic data generation model will be presented.

In addition, other probabilistic models were considered and research on alternatives was performed. One type appearing in the results were so called Hidden Markov Models (HMM [24, 32, 33]. In addition to the definition of states of time and transition probabilities, in an HMM unobservable "hidden states" exist and add more complexity to the Markov model. This makes it more powerful, however, HMMs have other specialized fields of application, for example speech or facial recognition.

### 3.2.4. Evolutionary Algorithms

Evolutionary algorithms (EA), often also called genetic algorithms, take advantage of evolution theory for algorithm design.This means evolutionary algorithms make use of data populations and evolutionary process structures like mutations to solve modeling or optimization problems. More detailed, this can be described after D. Ashlock [3] as operations on populations. Data structures are chosen to represent populations, quality measures and different methods to vary data structures are applied. In addition, stopping criteria need to be introduced to know when the data structure has a satisfying quality.

While there exist many approaches on how to use EA for test data generation, such as [10, 27, 38, 39], research shows that for this work, an approach with evolutionary algorithms would be similar to the selected stochastic model. With both it is possible to generate test data a by simulating a supply chain and analyze path coverage with various data sets. However, the definition of parameters and initial data sets for mutations is a very complex process. In contrast to the learning process of a stochastic model, an evolutionary algorithm is very sensitive regarding parameter changes and the mutation and comparison process might change the results significantly so that the initial data set and mutation parameters must be set very carefully. As this additional complexity is considered unneeded in this work's scope, a stochastic model is preferred over an evolutionary algorithm.

## 3.3. Sequential Model

Considering the exemplary process step description, an intuitive modeling approach is to consider the individual locations in terms of a standard state machine and the individual actions in terms of a simple, sequential process flow. Naturally, this approach describes the lifecycle for individual items. This model is used to handle store processes for test data generation in a randomized way. Based on expert knowledge of the workflow for single items, transitions between nodes (locations) are defined. Basically, the sequential model is an abstract representation of a single item's lifecycle in the supply chain, mapped to a sequential representation of actions covering a subset of items in stock $I$.

In a sample model instance, we will assume that a store has only two locations: the back room $BR$ and the sales floor $SF$ with the process flow and the actions displayed in Figure 3.2. In this model, a number of $l$ store processes are performed consecutively according to the transitions in the graphic. The possible actions covered by this model are initialization, stock take, relocation, sale and return and with regards to the locations can be further divided into:

- $InitBR$: Initialization on the back room.
- $InitSF$: Initialization on the sales floor.
- $STBR$: Stock take on the back room.
- $STSF$: Stock take on the sales floor.
- $RelBR$: Relocation to the back room.
- $RelSF$: Relocation to the sales floor.
- $Sale$: Sale.
- $Return$: Return.

Any sequence length $l$ (the number of actions performed) can be assumed as this model contains no mechanisms to limit the length or generate the sequence length from any empirical analysis. For the model instance depicted in Figure 3.2, this means the execution of such a sequence always starts with an initialization on either the back room or the sales floor. The initialization step could contain any number of items and is usually defined beforehand. From that point on, the next step is performed based on the last action according to the defined transitions.

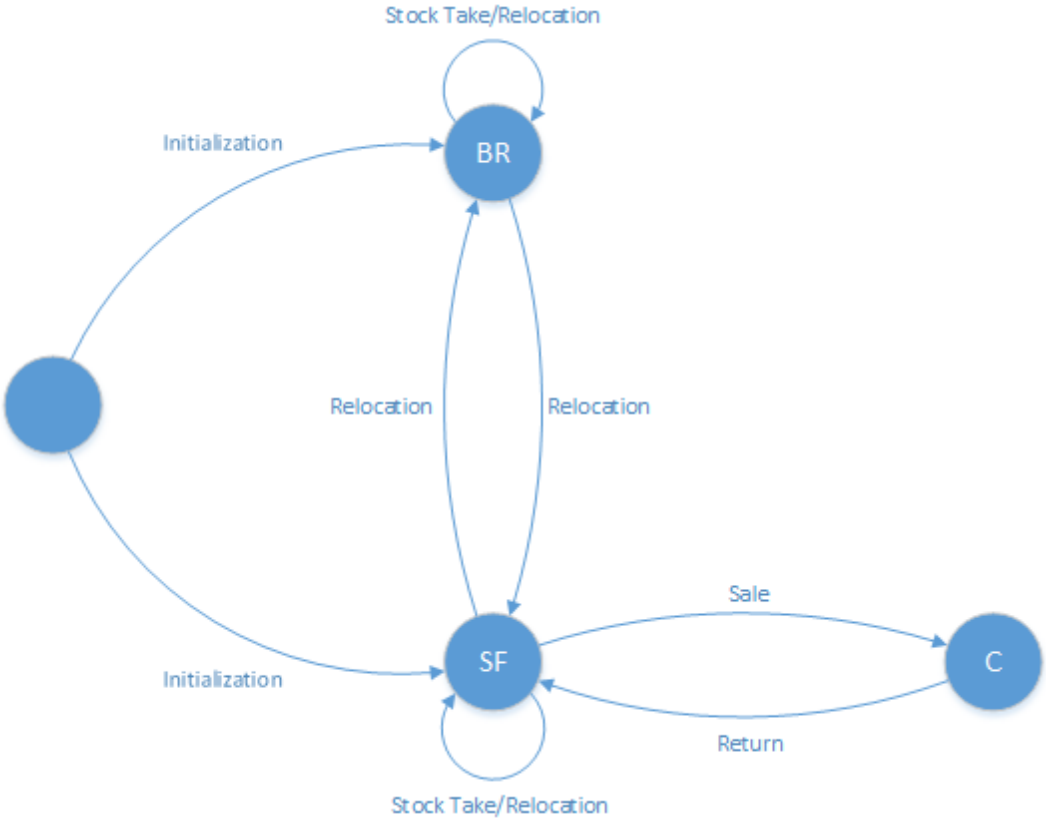


Figure 3.2.: Process flow for an individual item in the sequential approach. This example covers the item being added to the stock with an initialization as well as various actions such as stock take, relocation, sale or return. The item leaves the stock when it reaches the customer.

Each action in the sequence changes the number of items of at least one location in the supply chain. For example, stock takes and relocations change $\|BR\|$ and $\|SF\|$, initializations and returns increase $\|I\|$ and sales decrease $\|I\|$. The amount of items per action is selected randomly, but is limited by predefined boundaries inside the following intervals for each action:

- **Initialize:** The number of items to initialize is any integer number $i > 0$.
- **Stock Take:** For the stock taking process, the number of items considered is defined as

  $$STBR = p \cdot \|BR\| + q \cdot \|SF\|$$

  or

  $$STSF = p \cdot \|SF\| + q \cdot \|BR\|,$$

  depending on the location the stock take is performed on.
  The usual coverage $(p_{low}, p_{up})$ is defined with $p_{low} = 0.7$ and $p_{up} = 1$ for the location of the stock take and $(q_{low}, q_{up})$ with $q_{low} = 0$ and $q_{up} = 0.1$ for the items detected from other locations. This means between 70% and 100% of the items of the stock take location are detected, plus an additional amount of between 0 and 10% of items on the other location. The item selection follows a uniform distribution, where every item on a location has the same probability to be picked.
- **Relocation:** With a relocation, between 1 and 5% of $I$ is selected for a relocation, that is, $p_{low} = 0.01$ and $p_{up} = 0.05$ for the interval $(p_{low}, p_{up})$, with the random selection following a uniform distribution on $I$. This is based on the domain knowledge that only a small amount of items will be relocated, for example to bring items from a back room to the sales floor.
- **Sale:** The average shopping basket can be defined as one to ten items from the sales floor, randomly selected from $SF$ by a uniform distribution.
- **Return:** As already mentioned, only a very small amount of sold items is returned again. Therefore, either one or two items are picked randomly from the set of sold items.

Basically, the presented sample model can be extended to a various amount of nodes and transitions to represent a more complex item lifecycle in the supply chain. The sequential model is intended to execute one action after another for a possibly infinite amount of steps, following a very strict order of defined transition rules according to a single item's lifecycle. To generate meaningful data with this model, expert knowledge is needed to define states and transitions between these states manually. Based on the structure of this model, it also faces a few limitations. Therefore, it is not possible to model store actions independent from each other, like different actions taking place at the same time on different locations in the store. As example taken from the sample model, a sale *must* be followed by a return action from items back to the sales floor, which does apply to a single item, but not to the definition of actions performed in the supply chain. The sequential model is an easy-to-use model for quick test data generation with a few limitations regarding extensibility and coverage based on the process flow with respect to specifications.

## 3.4. Stochastic Supply Chain Model

A more sophisticated approach to tackle the challenges of test data generation is the usage of a stochastic model. Different to the sequential model, instead of defining boundaries and process flows based solely on knowledge of the system, real-life data from deployed store supply chain management systems is used to train the behavior of a data generation model. In this section, we will discuss general properties of Markov chains and Markov models and how they can be applied to the considered scenario of supply chain management with a particular focus on the fashion retail supply chain. This stochastic model is a generalization of the sequential model to allow a wider range of transitions independently from a defined workflow.

### 3.4.1. Transition Model

A Markov model is the model of a so called discrete time Markov chain (DTMC) or discrete Markov process, one special type of stochastic process. A

stochastic process $\{X_i\}$ is described by Cover & Thomas [9, Ch. 4.1, p.71] as an indexed sequence of random variables $X_1, ...X_n$ with a possible arbitrary dependence among them.

In a Markov chain, the random variables are called *states* and the characterizing property of a memory-less or first-order Markov chain is that at any time $t$, the directly following state at time $t+1$ depends only on the state at time $t$ and no other past states. A simple explanation of a Markov chain is given by M. Kuperberg [25] as a set of discrete states which, at any point of time, is in exactly one of these states.

Defined by L.R. Rabiner [32], a discrete Markov process is a system which in any time is in one of $N$ distinct states $S_1, S_2, ..., S_N$ where $S_1, S_2, ...S_N$ are random variables describing each state. Depending on transition probabilities between the states, the system changes its state at given times $t = 1, 2, ...$ with the state at time $t$ being called $q_t$ (random variables). As mentioned before, in a first-order Markov chain the current state is only depending on its predecessor state, hence we can describe the probability of the system being in state $S_j$ as

$$P\left[q_t = S_j | q_{t-1} = S_i, q_{t-2} = S_k, ...\right] = P\left[q_t = S_j | q_{t-1} = S_i\right].$$

According to [32], we can write the probability distribution as a transition matrix $A$ with dimensions $N \times N$ using the right side of the previous equation as

$$a_{ij} = P\left[q_t = S_j | q_{t-1} = S_i\right]$$

such that

$$a_{ij} \geq 0$$

and

$$\sum_{j=1}^{N} a_{ij} = 1.$$

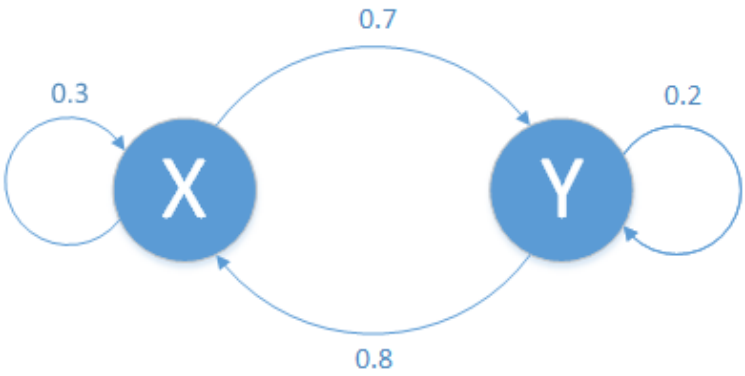


Figure 3.3.: A simple Markov process with two states and four possible state transitions.

For the model in this master's thesis, the challenge is to find the initial vector $\nu$, as well as the transition matrix $A$. $\nu$ contains the probabilities for the system to be in a state $S_i$ at time $t_0$, therefore with this vector the "start state" is defined. To obtain $\nu$ and the model's probability distribution and transition matrix $A$, it is trained with a given set of state sequences from practical usage. By analyzing these sequences, $\nu$ and $A$ are updated. To find a meaningful model, sufficiently large training sets need to be selected. More details on how the training works in the special case of this work can be found in the definition of the test data generation model in the next Section 3.4.2. However, examples to train Markov models are different EM algorithms (expectation maximization) such as the Baum-Welch algorithm discussed by L.R. Rabiner and B.H. Juang [33], as well as techniques for so called hidden Markov models (HMM), one special type of Markov model containing unobservable states, presented by Khreich et al. [24].

With these parameters defined, a sequence of state transitions can be generated from the model. Basically, in each state the next state is selected randomly depending on the probabilities given in $A$. This is usually done using algorithms suitable for the purpose (examples for different types of Markov chains were discussed shortly in lecture notes by K. Sigman [34]).

**Example:** Figure 3.3 shows a simple Markov chain with two random variables/states $X$ and $Y$ and their transition probabilities. A possible initial

vector for this state machine can be

$$\nu = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix},$$

where each of the two states is equally likely when the process is started. The corresponding transition matrix is defined as

$$A = \begin{array}{c} \\ X \\ Y \end{array} \begin{array}{c} \begin{array}{cc} X & Y \end{array} \\ \begin{pmatrix} 0.3 & 0.7 \\ 0.8 & 0.2 \end{pmatrix} \end{array}.$$

For test data generation, a Markov chain is used to model supply chain actions, such as were explained in Section 3.1. For the model's set of states, we need to define the set of actions as

$$M = \{a_1, a_2, ..., a_m\}$$

and the set of locations as

$$L = \{L_1, L_2, ..., L_n\}$$

for all locations of the store to model. With $M$ and $L$ defined the model's state set $S$ can be written as the Cartesian product

$$S = M \times L,$$

with the number of states

$$|S| = |M| \cdot |L|.$$

This means that theoretically each action can possibly be performed on each of the locations. With these definitions, the initial vector $\nu$ is defined with length $|S|$ and the dimensions of the transition matrix $A$ as $|S| \times |S|$.

**Example:** If we consider a sample system with the actions

$$M = \{Initialization, Stocktake, Relocation, Writeoff, Sale, Return\}$$

and two locations $L_1$ (back room) and $L_2$ (sales floor), we get 12 states for the model: Initialization on back room or sales floor, stock take on back room or sales floor, and so on. In this example, the initial vector $\nu$ would be of length 12, the transition matrix $A$ of dimension $12 \times 12$.

At this point it needs to be said that possibly not all of these actions are depending on a specific location. However, we might need to create faulty data like this with the model, so these cases still need to be considered. This is why we calculate the Cartesian product of actions and locations instead of defining some exceptions or constraints. This can be simply done by adding noise to $\nu$ or $A$ after the training process described in the following.

Previous to the training, the initial vector and the transition matrix have to be initialized. For both $A$ and $\nu$, a zero matrix or zero vector, respectively, is assumed and will be updated by training the model with a sequence obtained from a productive system. For the stochastic model to work and generate meaningful test data, it must be trained with sequences consisting of a consecutive list of $n$ actions with the following information:

- The store **action** performed.
- The **location** the action was performed on.
- The **number of items moved** with the action. Note that the number of items is not used for the model training but to obtain a meaningful number of items for each action in the later execution of a state transition sequence generated by the stochastic model.

In the update algorithm for $A$, every transition from one action in the sequence to the next is considered. The update algorithm depending on the given training sequences works as described in Algorithm 1.

In addition, the initial vector $\nu$ needs to be trained based on the first action of each sequence. While the training process of $A$ and $\nu$ needs the same training data as input, we describe the calculation of $\nu$ with Algorithm 2 for better readability.

**Algorithm 1** Training of transition matrix $A$

INPUT: training sequences with $n$ $action \times location$ pairs

set $A = 0$ at each index
**for all** sequences **do**
  get $action \times location$ of elements $i_k$ and $i_{k+1}$ for $k = 1$ in the sequence list
  increment value of $A$ at index $action \times location(i_k)$, $action \times location(i_{k+1})$

  **while** $k + 1 \leq n$ **do**
    set $k = k + 1$
    get $action \times location$ of $i_k$ and $i_{k+1}$
    increment value of $A$ at index $action \times location(i_k)$, $action \times location(i_{k+1})$
  **end while**
**end for**

**for** $j = 1$ to $\#rows$ in $A$ **do**
  $s_j = \sum values$ in row
  **if** $s_j > 0$ **then**
    $\forall$ values in row: $v = \frac{v}{s_j}$
  **else**
    set $value = \frac{1}{row.length}$ for each value in row
  **end if**
**end for**

return $A$

**Algorithm 2** Update initial vector $\nu$

INPUT: training sequences with $n$ *action* $\times$ *location* pairs

set $\nu = 0$ at each index

**for** $\forall$ sequences **do**
  get $a$, first *action* $\times$ *location* pair of sequence
  increment value of $\nu$ at index $a$
**end for**

**for all** values $v$ in $\nu$ **do**
  $v = \frac{v}{\#sequences}$
**end for**

return $\nu$

Adding noise to distort the results from the model is a simple task after $\nu$ and $A$ are trained. This is achieved by adding an additional relative factor $\epsilon$ to the transition matrix or initial vector, such that values are altered and possible zero values are considered when generating an action sequence from the model. A sample matrix of dimension $3 \times 3$ with an additional $\epsilon$ for noise is

$$A = \begin{pmatrix} a_{1,1} - \frac{\epsilon}{2} & a_{1,2} - \frac{\epsilon}{2} & \epsilon \\ a_{2,1} - \frac{\epsilon}{2} & \epsilon & a_{2,3} - \frac{\epsilon}{2} \\ a_{3,1} - \frac{\epsilon}{2} & \epsilon & a_{3,3} - \frac{\epsilon}{2} \end{pmatrix},$$

where $\epsilon$ replaces zero values in the matrix. The same could also be applied to the initial vector $\nu$ to generate faulty start states. However, training the model and the possible addition of faulty behavior only covers the selection of actions to perform on particular locations, but has no information about the number of actions to execute or which amount of items are considered for an action. The characteristics of sequences and item distributions are covered in the next section.

### 3.4.2. Time and Item Set Characteristics

In addition to the estimation of $\nu$ and $A$, the average size of a generated transition sequence and the average number of items for each action need to be estimated. Taking another look at the training sequences, we count the number of actions for each sequence and take the mean value as $\lambda$ to model a Poisson distribution as described by N. Henze [20, Ch. 24, pp.189-194]. The same must be done for each $action \times location$ pair in $S$ relative to the training data set's stock size. Thus, in the Poisson distribution, $\lambda$ will represent the average percentage of items taken for each action. This is why we define $a$ as the mean value of the number of items for each $action \times location$ pair and can calculate

$$\lambda = \frac{a}{size} \cdot 100.$$

**Example:** A stochastic model is already trained with only two actions and two locations, called $a_1$ and $a_2$ and $L_1$, $L_2$, respectively. We have the following four training sequences and a store size of 30 items:

1. ($a_1$, $L_1$, 3 items), ($a_1$, $L_2$, 4 items), ($a_2$, $L_2$, 6 items)
2. ($a_1$, $L_2$, 10 items), ($a_1$, $L_2$, 8 items)
3. ($a_2$, $L_1$, 3 items), ($a_2$, $L_1$, 4 items), ($a_1$, $L_1$, 5 items), ($a_2$, $L_2$, 8 items)
4. ($a_1$, $L_1$, 10 items), ($a_2$, $L_2$, 7 items), ($a_1$, $L_2$, 3 items)

We get

$$A = \begin{array}{c} \\ a1,L1 \\ a1,L2 \\ a2,L1 \\ a2,L2 \end{array} \begin{array}{c} \begin{array}{cccc} a1,L1 & a1,L2 & a2,L1 & a2,L2 \end{array} \\ \begin{pmatrix} 0 & \frac{1}{3} & 0 & \frac{2}{3} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

and

$$\nu = \begin{pmatrix} 0.5 \\ 0.25 \\ 0.25 \\ 0 \end{pmatrix}.$$

For the sequence length Poisson distribution, we calculate

$$\lambda_{seq} = \frac{(3+2+4+3)}{4} = 3.$$

For the actions we calculate $\lambda_{a1,L1} = \frac{(3+5+10)\cdot 100}{3\cdot 30} = \frac{1800}{90} = 20$

And following from that $\lambda_{a1,L2} \approx 20.83$, $\lambda_{a2,L1} \approx 11.67$ and $\lambda_{a2,L2} \approx 23.33$.

With this knowledge base, we can generate transition sequences with the model and its supporting distributions. The first step is to get the length the generated sequence should have. To get the number of actions, a random number $r$ following the calculated Poisson distribution is selected. This random number is most likely to be similar to the calculated $\lambda_{seq}$, but more flexible than defining a fixed value for the sequence length. With the stochastic model, a state $q_0$ at time $t_0$ is picked randomly based on the distribution of $\nu$. From this start state the model knows from which row in $A$ the next transition is selected based on the transition probabilities. All $q_1, ..., q_r$ states of the sequence to generate are then picked randomly according to the probability distribution of $A$.

The generated sequence of length $r$ contains a list of $action \times location$ pairs and suggest the actions and locations to perform for test data creation to simulate a store's everyday procedure. The actions will be performed consecutively at the given location with a random percentage of items (based on the determined distribution from the training).

Again, it is possible to add noise for faulty behavior, for example to alter the expected results and confront the system to test with unexpected, possibly erroneous data. To achieve that, the Poisson distributions for actions as well as the sequences can be altered by adding a relative $\epsilon$ to change the expected number of items or sequence length.

Summarized, in contrast to the sequential model, the stochastic model can be trained using empirical data from productive environments. This adds more flexibility, as the modeling process is not only limited to specifications and domain knowledge. By means of analyzing the process structure provided by the store data, a set of realistic probabilities for store actions can be estimated with the presented Algorithms 1 and 2. Depending on the data, it is possible that each action can occur at any location and at any point of time during the sequence generation, based on the previously estimated probability distribution for the supply chain actions. Additionally, from the training data the number of actions performed over an observed amount of time as well as the distributions of the item set size for each action can be estimated.

The stochastic model can be generalized even more with the addition of noise to any model parameter, whether it is the initial vector, the transition matrix or item distributions for actions and sequences. This causes the incorporation of unlikely state transitions in order to increase test coverage, as they still occur in practice and often cause unexpected or faulty system behavior.

# 4. Implementation

This chapter covers details on the implementation of the test data generation framework. In Section 4.1 we give a short introduction to the underlying system for the implementation. Section 4.2 contains all implementational details, such as used programming languages, libraries and the general structure of the implementation. In Section 4.3, we will present the implementation of the sequential model described in Section 3.3. Section 4.4 covers the stochastic model described in Section 3.4.

## 4.1. System Overview

The underlying system for the implementation is a distributed, service-based RFID system on the store level of a supply chain. Figure 4.1 shows a sample structure of a service-based RFID test system with its main components. Basically, the displayed Windows Communication Foundation (WCF) [28] web service acts as a communicator between the different components of the distributed system, namely a database server as well as devices with running software applications, possibly communicating with RFID devices.

For the implementation, an interface to the web service is used to communicate with the database in two ways: to get required data from the database and to perform actions and write them into the database. As the execution of these service calls should simulate the usage of the store system's applications to create test data, these applications are negligible. More information about a store process in general was already given in Section 3.1.

Directly at the web service is where the data generation model will be located. As with the software applications in the system, communication is possible in two ways:

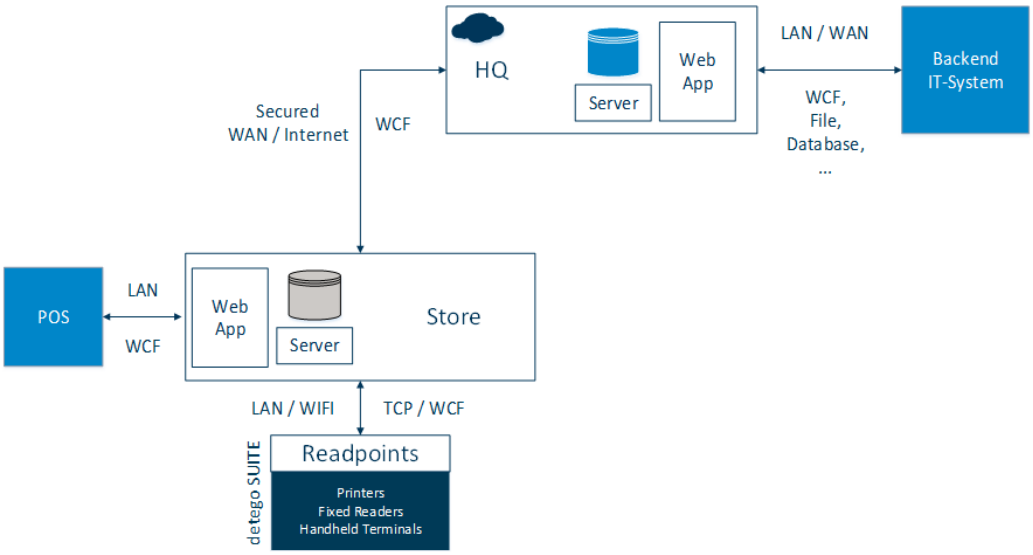


Figure 4.1.: Sample architecture for a productive RFID solution in the fashion industry by the Enso Detego GmbH [17]. This architecture consists of the databases for each component in the supply chain, communicating with each other over WCF web services. In addition to the communication with IT-backend and database, applications such as point of sale (POS) or various stationary or web applications use said database and web services to perform actions.

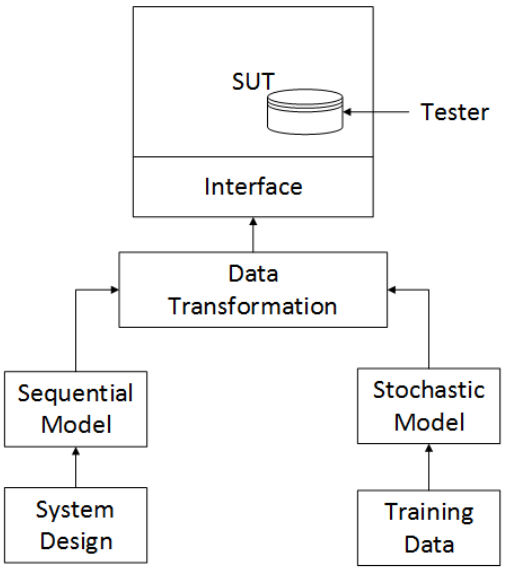


Figure 4.2.: Framework architecture presenting the model architecture and its application onto the system under test. Based on the model, either specifications or training data is used for the data generation. Before the data can be applied to the SUT, it has to be adapted to meet the SUT's requirements.

1. For the simulation of store actions, it is necessary to have information on the data already stored in the stock, such as the stock size, the items' EPCs or the item distribution over the store's locations. To be able to perform the actions according to the process model of Section 3.1, it is necessary that the required information can be retrieved with a request to the web service.
2. When the generated data should be sent to the test system by the model, the interface should handle the correct conversion from the information the generated test data contains to data the service can process. The goal is that the web service stores the data in the database and makes it accessible for the other system components which need to be tested with the generated data.

The communication of the model framework with the system's web service is done over an interface and is displayed in Figure 4.2. The framework covers the data conversion for the models as well as the models and data

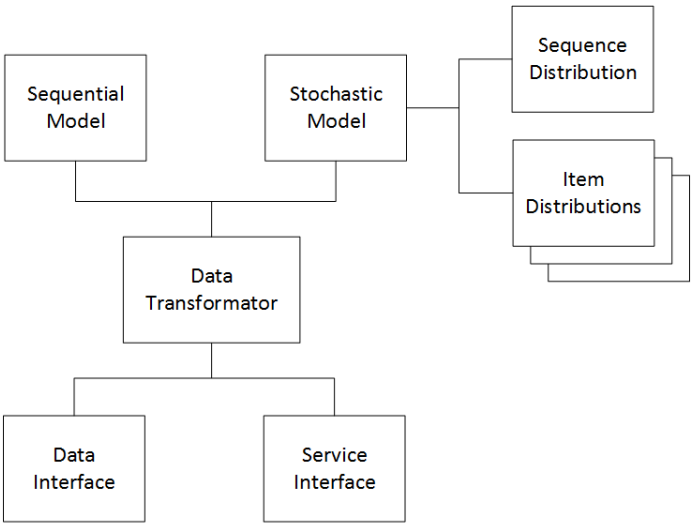


Figure 4.3.: Abstract class diagram of the most important parts of the model framework. These are both the sequential and the stochastic model, the sequence and item distributions, as well as the transformation class to prepare the generated data to be sent to the test system over the provided interfaces.

transformation to be used by the interface communicating with the system under test.

## 4.2. Implementation Details

The implementation of the model-based test data generation framework is a C# WPF application [29] with a graphical user interface (GUI). In this section, all general functionality of the application, any used libraries and the usage independent from the implemented test data generation models are described. The next sections cover the interfaces to the underlying system, data handling and other useful functionality besides the sequential and stochastic supply chain model.

As introduction, Figure 4.3 shows a diagram of the most important classes of the model framework, such as the models and distributions, as well as the data transformation class connected to SUT's interfaces. In the following,

the graphical interface, the main classes and the logging mechanisms are described.

### 4.2.1. Graphical Interface

The `MainWindow.cs` class contains all the application's superficial functionality. Here the graphical user interface such as button functionality, user input and output to the user as well as file dialogs, is located. In addition to the model implementations, the following functionality is included:

1. **Sequences:** It is possible to import a semicolon-delimited CSV file with unique pre-defined headers to convert it into a training sequence for the stochastic model. The input CSV needs to contain the fields *TimestampEvent*, *LocationName* and *BusinessStepName* (the action performed). All other possible fields in the table are not needed and are ignored. From this data, all entries with the same timestamp, location and action are counted and written into the training sequence file (also a CSV file). After the conversion, the training sequences contain a list of consecutive actions on different locations with a varying number of items per action. Example: stock take on the sales floor with 100 items.
2. **Masterdata:** The functionality to open and execute a .bat file to re-instantiate the database where test data needs to be created with predefined masterdata is provided. The masterdata could contain anything from the definition of locations, product masterdata and all other connectivities for a productive system to work. In addition, it is possible to initialize the database size, that is, the number of items which should be present in the test database before applying any model functionality.
3. **Single Processes:** In some cases it can be useful that not a whole sequence of store actions is executed, but only data from a single process, for example one sales process, is needed for a test. Therefore, functionality to select a process from a dropdown menu, add the required data (location and/or number of items) and perform only this single action is added.

Summarized, to give the tester the possibility to prepare the system for the model application, a selection of additional functionality is included in the framework. This includes the conversion of a database extract to a sequence file, the possibility to reset the system under test as well as to perform single process steps independently from the models.

### 4.2.2. Classes

In addition to the model implementations, the application uses four other classes to provide necessary functionality. These are a class for general functionality, two interfaces for data and service access as well as a class for supply chain functionality to communicate with the interfaces.

In the `GeneralFunctions.cs` class, all the system's functionality which does not fit to other, more specified classes is handled here. Methods of this class contain methods to import or export data from/into files as well as calculations or data conversions. The most important methods are listed in Appendix A.1.

In the underlying system, all processes regarding store actions are handled by the web service. The application calls the system's web service via an interface to get access to its functionality and methods to send data into the database. The interface is implemented in the `ServiceCalls.cs` class and each method uses a data client to reach the desired web service URL for the methods listed in Appendix A.2.

`ServiceCalls.cs` provides these methods via communication with the referenced web service to ensure that, depending on the underlying system, the functions with the correct parameters are sent to the system. The implementation of these methods must handle that the passed model parameters and variables are possibly not enough to perform the service call and add reasonable additional data.

All calls to the system regarding already stored data in the database is handled by the `DataCalls.cs` class. In the application, the web service's data is accessed via HTTP requests, to get a list of values in the desired table to use in other methods of the application. In the `DataCalls.cs` class,

the methods described in Appendix A.3 can be called to get data from the test system's database to perform various store actions.

The `StoreFunctionality.cs` class is the main class to pass all the actions performed by the model to the web service interface. Basically, this class prepares the given data from the model execution such that the underlying interface can handle it better, being it by performing additional calculations (such as the selection of a given number of EPCs from the database) or adding static parameters as the interface call requires them, but are not required at this state of the model design. The method descriptions can be found in Appendix A.4.

### 4.2.3. Logging

To all important processes and methods in the implementation, log output is added using the Apache log4net library [13]. By including log4net in the desired classes and initializing the logger it can be used all over the class to define different types of log output at any time and will be saved to log files which can be found in the *log* folder of the deployed application. The log file contains a timestamp, the logging level, the class which called the corresponding logger as well as the log output.

As the log file should contain important runtime information about the application and to maintain a certain level of observability, the generated log output can be summarized as the executed processes, generated sequences and execution times. The executed processes cover the successful execution of actions as well as logging about the training process or completed sequence execution. In addition, the generated sequences are logged before their actions are executed to be able to identify possible discrepancies or faulty behavior. Another thing logged is the time needed to execute various processes. This is important to evaluate results of load or performance tests.

## 4.3. Sequential Approach

The `StoreAutomaton.cs` class contains the sequential model's implementation. For this approach, the Stateless .NET State Machine Framework [5]. The Stateless framework can be used to implement simple state machines or workflows similar to state machines. This is the reason why it seemed useful for this purpose and the sample distribution matrix from Section 3.3 was transformed to a Stateless state machine, where the actions are executed on entering the next state, based on the transition rules.

**Example:** Basically, a very simple state machine with two states *StateA* and *StateB* and the actions *actionA* and *actionB* would be implemented as follows:

```
State init = State.StateA;
var stateMachine =
        new StateMachine<State, Transition>
                (() => init, s=> init = s );

stateMachine.Configure(State.StateA)
        .Permit(Transition.A, State.StateB)
        .OnEntryFrom(Transition.B, t => actionB());

stateMachine.Configure(State.B)
        .Permit(Transition.B, State.StateA)
        .OnEntryFrom(Transition.A, t => actionA());
```

In this code, the state machine is configured with the two states and state transitions *Transition.A* from *StateA* to *StateB* and the other way round. Each time a state is entered with a transition, either *actionA* or *actionB* is performed. One state transition can be fired manually with a call such as

```
stateMachine.Fire(Transition.A);
```

### 4.3.1. Usage

The functionality of the sequential model is spread over two tabs: *Automatic* for the execution of a sequence of a manually defined length and *Manual* to execute actions of the sequential model step by step with the possibility to get direct information about the performed actions.

**Automatic Execution:** For an automatic sequence execution of the sequential model, the user needs to select the number of items to be initialized in the database and how many randomly selected actions should be performed (how many transitions are performed, with method execution on entering or leaving). As the state machine works too fast for a human user to follow the steps, no direct output or information is given to the user, all performed actions with the number of items can be found in the log output file instead.

**Manual Execution:** A manual execution is possible with the sequential model. Basically, the functionality is the same as the automatic execution with the difference that the user only defines the number of items to initialize the database with and from that on fires every state transition and action execution manually. Information about the actions performed is displayed in a text box.

## 4.4. Stochastic Supply Chain Model

To implement the stochastic supply chain model and probability distributions defined in Section 3.4, the functionality of the Accord.NET framework [1], a C# library for machine learning and statistics, licensed under the GNU Lesser General Public License v2.1, is used. The store's stochastic model functionality is implemented in the class *StoreMarkovModel.cs*, the distributions for sequence length and number of items in *SequenceDistributionModel.cs* and *ItemDistributionModel.cs*.

In general, the `HiddenMarkovModel` class of the Accord.NET framework is used to define the Markov model parameters $\nu$, the initial vector, and $A$, the transition matrix. As an HMM supports unobservable states (see [32] for

more information), for this purpose it is assumed that the number of states is equal to the number of observable symbols, meaning both are equal to the set $S$. Based on this definition, the HMM matrix $B$, the emission matrix for observable output probabilities, is not needed, as every state can have exactly one output. This is the reason why $B$ is initialized as identity matrix of dimension $|S| \times |S|$, because with this emission matrix, an HMM acts the same way as a Markov model without hidden states.

However, a few drawbacks of the framework were encountered while implementing the model functionality. Without going further into detail, the `HiddenMarkovModel` training methods do not support the update mechanism of the transition matrix $A$ alone with a predefined initial vector $\nu$ and emission matrix $B$. This is the reason why, among other things, the update algorithm from the model definition was implemented independently from the used Accord.NET framework and passed to the model afterwards. In addition, the functionality to either train the model with training sequences provided by a running store solution or to import previously saved model parameters (transition matrices, distributions and other needed data) is contained in the application. The stochastic supply chain model's implementation spans over three different classes, the stochastic model, the sequence distribution and the item set size distributions.

In the `StoreMarkovModel.cs` class, methods to create and train the Markov model as well as the functionality to generate and execute sequences is implemented. Basically, the first step before any model can be created or trained, the store actions which should be covered by the model need to be defined. In this work, these are initialization of items, stock take, relocation, sale, return, write off, inbound and outbound, all of them possible on different locations in a store. For the `HiddenMarkovModel` class, the possible $action \times location$ pairs (the states) need to be edited in a way the model can understand, which means they need to be defined as integer values. To translate the states we use a codebook to map each state to a unique integer value. After that, the Markov model can be created, trained, and sequences can be generated. Thus, the generated sequence will have the form of an array of integer values. Only at that point the codebook is used again to translate the integer values back to the $action \times location$ pairs for the execution. Details on the methods contained in the class can be found in Appendix A.5.

The two classes for the sequence distribution and the item distributions contain distribution calculations for each store action to generate a fitting sequence length or number of items used for each action automatically. For the distributions, the `PoissonDistribution` class provided by the Accord.NET framework is used, as this library provides all needed functionality to learn a distribution and generate observations. Based on the methods summarized in Appendix A.6, the average sequence length and number of items to use for an action can be estimated when applying the stochastic model.

### 4.4.1. Usage

With the stochastic supply chain model, the first step is to train the model to define the model's parameters, the transition matrix and distributions. For the training the interface provides three possibilities:

- **Train with predefined sequences:** To get a quick and simple model definition to work with in case no other training data is available, a small training sequence is deposited and can be used to train the model. A short message informs the user if the training was successful.
- **Import model parameters:** As learning the parameters with training sequences can be time-consuming, it is possible to import an already trained Markov model as well as sequence and item distributions. A file dialog prompts the user to select the Markov model file (.hmm) as well as the distribution files for the average sequence length and the item distributions for the store actions (.dist). As these three files are saved by default after every successful training, it is easy to store them as a backup.
- **Training with a sequence file:** The standard training is done with the selection of a sequence file from a file dialog and then train the model. A short message informs the user if the training was successful.

After the training, the model is ready to generate test data. Again, it is possible to either define a sequence length manually (in case one wants to test how many consecutive actions the system can handle) or select *Automatic Length* to get the sequence length based on the calculated Poisson

distribution. Again, all actions performed produce log output for better observability and maintainability of the process.

# 5. Empirical Analysis and Usability

In this chapter, the designed and implemented data generation model will be analyzed and evaluated regarding its usefulness and usability on a real-life test system for RFID supply chain management. In collaboration with the Enso Detego GmbH, the framework was included in their system for store supply chain management, detego SUITE, with the intent to relieve the testers from the tedious task of manually creating test data such that certain test cases can be executed. This chapter will cover a short overview about the system where the model is integrated, and afterwards the analysis on how well the model framework performs its tasks according to the specified test types and requirements. Results regarding module functionality, performance and data generation times will be presented and compared to the manual creation of test data. In addition the topic of coverage in combination with the data generation models will be discussed.

## 5.1. System under Test

The system under test, detego SUITE, is a web-service-based distributed system for supply chain management using RFID technology and is displayed in Figure 4.1. The database contains all system information, in case of store management locations, article and item information, item tracking information and so on. With usage of the software applications and the additional hardware, the processes of the supply chain and actions of the everyday store life (goods availability, for example) are performed. For that, the web service is called to either retrieve data from the database to use or display in one of the applications, or data collected by the application is given to the web service to process them to the database.

While the system covers applications for the various parts of the supply chain, for example solutions for logistics, store management or central components, for the evaluation of this work we are concentrating on testing selected actions in the store system. Namely, these are the sample process steps which were presented in Section 3.1.1.

The test system contains three locations to perform the previously named store actions on: sales floor, back room and shop window. According to this information, the interfaces were implemented to include the model framework in the SUT. Both the sequential and the stochastic supply chain model are applied to the store system to evaluate how well they meet the requirements. The generated data can be used in the various components of the system. In this case these are an application for a mobile RFID device, a desktop application and a web application. However, in this work the focus for the analysis lies mainly on three system aspects to analyze the framework functionality:

- *Direct access to the database*: In order to verify that the generated data was correctly added to the system, the corresponding tables are accessed. The content will be extracted and analyzed for various test cases.
- *Web service log output*: The log output of the web service will be inspected for possible failures, exceptions or wrong handling of the generated data from the models.
- *Web application*: The web application of the system under test contains information regarding a store's stock. The web application mainly consists of reports for stock information, for example the item distribution over the store's locations, as well as reports for the different actions performed in the store, which is information about the time and number of items for performed actions. For the framework analysis, the web application will be the main application used for the verification of the generated data.

With this information base, the processes to test the two models can be discussed. As an outlook, the following example will describe one possible test scenario and how it would be performed manually.

**Example:** The user wants to test one specific report of the web application, in this case the inbound report. This page contains a list of all inbounds,

regardless if they are open (delivery numbers and information about the items contained in the inbounds) or already performed. For each inbound, information about the status can be displayed. That is, if they were completed with all items detected, missing items or additional items. For manual generation, the following steps are required:

1. Use a central component or whatever it needs to add deliveries to the store system.
2. Use an RFID device to encode the corresponding tags with the correct EPCs for all of the deliveries.
3. Verify in the database that the deliveries are added correctly.
4. Verify that the deliveries are displayed correctly in the web application.
5. Use the mobile RFID application to perform three different types of inbounds:
    - Complete one inbound with all its advised items.
    - Complete one inbound with missing items.
    - Complete one inbound with more items than advised.
6. As the inbounds were performed manually with an error-prone application, verify in the database that the inbounds are added correctly according to the specifications.
7. Verify that the graphical interface of the web application shows the correct data.

We can now see that, when the test data needs to be created manually, there exists a high time factor for the creation of the deliveries, the tag encoding and the actions performed with the mobile device. In addition, all these processes performed manually are prone to cause erroneous data, whether this happens due to wrong handling of the user, faulty behavior of the used applications or simply due to environmental issues causing interferences when using RFID technology. Summarized, one simple manual test of this type could easily take 10 minutes and more. This knowledge will later be used to evaluate the measured execution times of the model framework.

To overcome these problems, the model framework for data generation is applied and the two implemented models will be compared to each other and the manual data generation process. On the one hand, we want to test how well the models cover the functionality of the previously listed actions.

| | Set 1 | Set 2 |
|---|---|---|
| **Store Size** | 10000 | 1800 |
| **#Sequences (Days)** | 114 | 12 |
| **Avg. Sequence Length** | 225 | 55 |
| **Total #Actions** | 25925 | 620 |

Table 5.1.: Used data sets for evaluation with information about store size, the number of sequences and the number of actions.

With the method to test called by the framework, we want to verify that the service works as specified. In addition, coverage regarding the selected actions will be discussed in form of path coverage in Section 5.3.3.

To compare the created data and train the stochastic model, we have two different data sets from deployed supply chain management solutions. These data sets are recordings of the item movements over days. One sequence in the sets describes the behavior of the system over a day and contains a list of actions covered by the models' implementations, including the number of items for each action. We refer to them as training set 1 and training set 2. The data sets' basic information can be described as displayed in Table 5.1. More details are listed as follows:

**Training set 1:** The larger one of the data sets is equal to a large database size for the test data generation. Figure 5.1 shows the distribution of the executed actions for an overall amount of nearly 26,000 measured actions relevant for the test data generation. Counting the number of executions per action shows that in a deployed system, sale and relocation are the two most-executed actions. In relation to that, Figure 5.2 shows the number of items used in each action. From this we can conclude that, while sale and relocation are executed many times a day, only a small amount of items is used, an average of one to two items is sold at once, and relocated is an average of about ten items - this corresponds to the previous explanation that only what can be carried by one staff member is relocated at once.
After that, other regular actions are inbounds and stock takes (also referred to as *inventory*). Inbounds explain the arrival process of new goods in a store. This is usually a higher amount of items at once and this can also be seen in the graphic. The same applies for the stock

take/inventory, however, other than expected, the number of items is not "most of the stock on one location", as in a store of medium size, this would be definitely more than approximately 370 items. This can be explained that usually more than one staff member performs stock takes with one device each on different parts of a location and reads only a subset of the items present. The number of items detected is later concatenated by other processes running in the system.
One other significant value is the average number of items used for an outbound. As the number of performed outbounds is rather low over the selected amount of time, we can assume that items leaving the store for other reasons than sales are items which are not needed anymore. This could be for example outdated seasonal ranges sent back to the distribution center or manufacturer when the items in stock change and would explain the high amount of items combined in a single outbound action.

**Training set 2:** Figure 5.3 and Figure 5.4 show a similar behavior for the smaller data set. In training set 2 the most-performed actions are again sale and relocation, followed by stock takes and inbounds. One interesting thing can be seen in the items per action chart in Figure 5.4 - the amount of used items is near to the numbers shown in Figure 5.2 from a data set nearly 5 times larger. One possible explanation is that regardless of the store size, customers tend to buy only a small amount of items and that one staff member can only perform a particular workload.

As a comparison for the training sets, Figure 5.5 shows the similarity of both training sets regarding the frequency of actions. To evaluate the models' functionality, both the sequential and the stochastic model were applied to the system under test for different use cases. The results of execution times, data quality, performance tests as well as usability will be discussed in the next sections. Basically, we will apply tests to three different test store sizes, covering the previously mentioned three locations and eight actions. The store/database sizes are defined as a Poisson distribution of small, medium and large size. The distributions have the following average size set as $\lambda$:

- **Small store:** A small store is defined with a stock size of around 2,000 items. Therefore, $\lambda = 2000$.

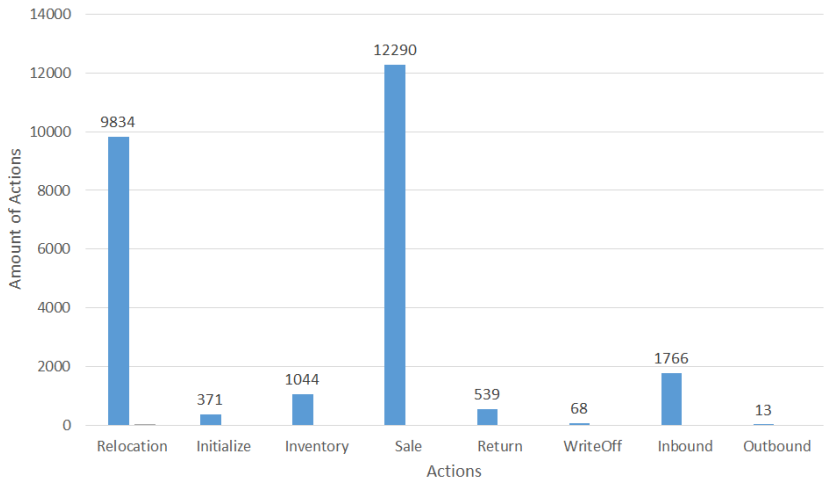


Figure 5.1.: Number of action executions in training set 1 (total amount of 25,925 actions).

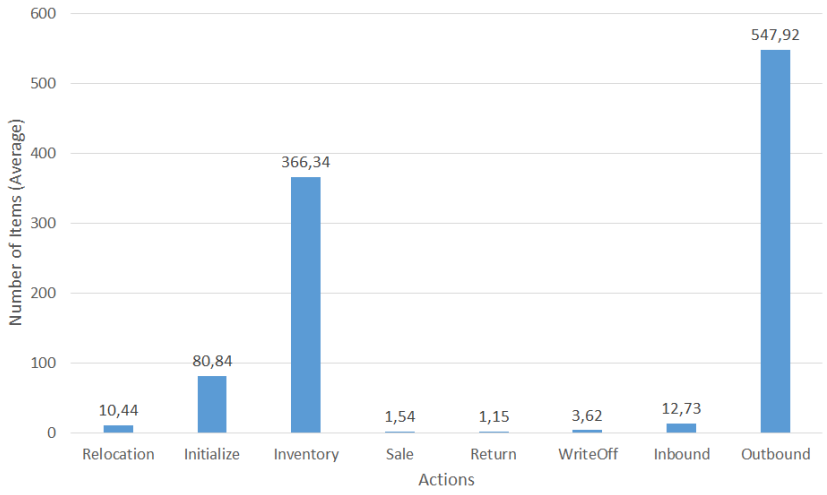


Figure 5.2.: Average number of items per action used in the first training set (data over 114 days, sequence length about 225).

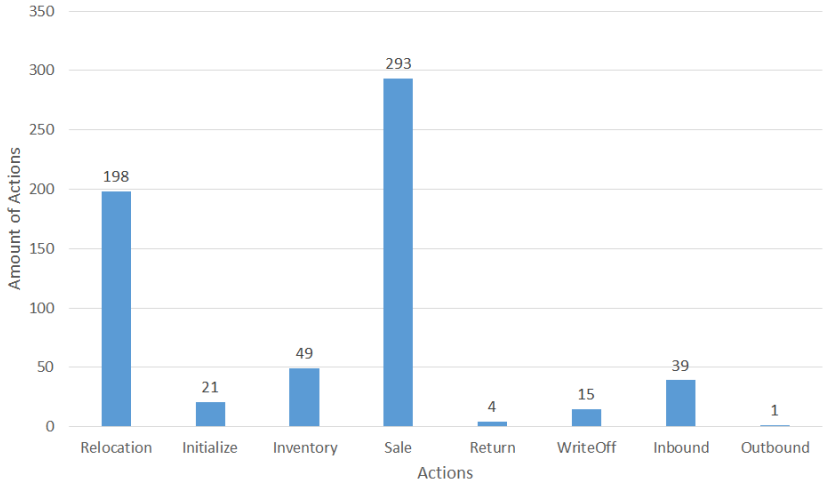


Figure 5.3.: Number of actions, training set 2 (total amount of 620 actions, store size around 1).

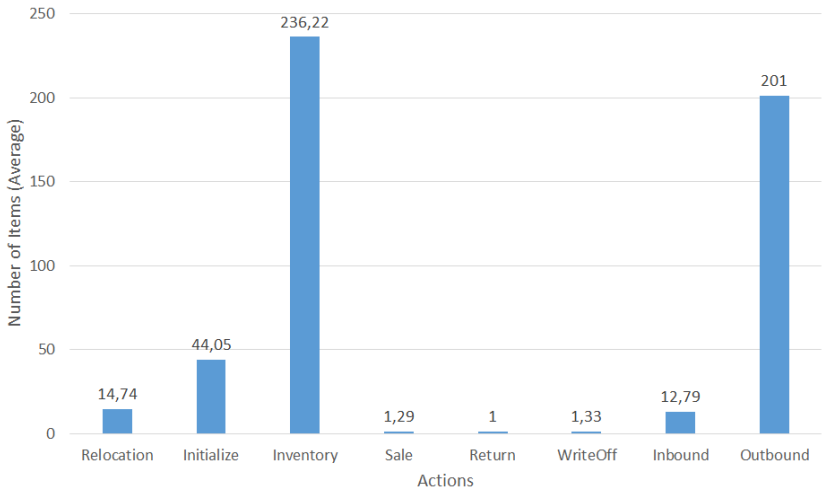


Figure 5.4.: Average number of items per action used in the second training set (data over 12 days, sequence length about 55).

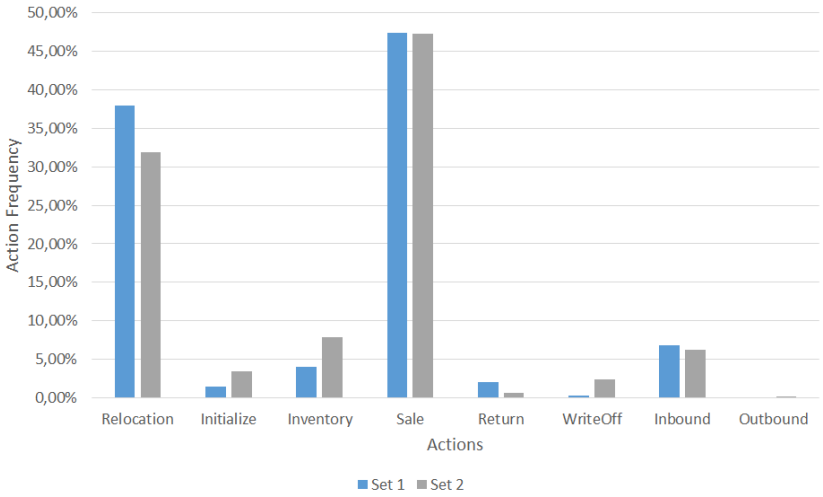


Figure 5.5.: Comparison of the relative number of actions in training set 1 and training set 2.

- **Medium store:** A medium store is defined with a stock size of around 5,000 items. Therefore, $\lambda = 5000$.
- **Large store:** A large store is defined with a stock size of around 10,000 items. Therefore, $\lambda = 10000$.

The exact size for a test run is a Poisson number, drawn from one of the defined distributions.

## 5.2. Execution Times

In Table 5.2, the results for the average execution time for a sequence of 100 actions is displayed for both models. The time presented is the average in seconds over 10 experiments each. However, preparations need to be made for both models before the actual execution time for data generation can be measured. The sequential model must have the number of items to initialize in the first step as input parameter, and the stochastic supply chain model must be trained before a sequence can be executed. The preparation was performed as follows:

| | Sequential Model | Stochastic Model |
|---|---|---|
| **Small** | 127.26s | 53.98s |
| **Medium** | 288.90s | 142.76s |
| **Large** | 480.03s | 415.84s |

Table 5.2.: Execution times for a sequence of length 100 for each store size in seconds. The larger the store, the more items need to be handled when performing actions and therefore the time needed increases.

- **Sequential model:** Input parameter for the first initialization step was set with 100 items.
- **Stochastic supply chain model:** Before any executions can be done with the stochastic model, it needs to learn its probability distribution from training sequences once. For this experiment, we used a rather small set of 12 sequences with an average length of 55 actions per sequence. The training of the model with these sequences additionally takes between 7 and 45 seconds additionally, depending on the store size the training data must be fitted to. However, this training process needs to be done only once and therefore this time is not included in the comparison.

Figure 5.6 shows a visualization of the average execution time per store size for both the sequential and the stochastic model. In the chart it can be seen that the execution time of the sequential model shows a linear growth with the store size. Figures 5.7 to 5.9 show the results of the single executions - again both models are compared to each other. Especially with the large store size, it can be seen that the execution times of the stochastic supply chain model are varying very much and the results are exceeding the execution times of the sequential model. After analyzing the detailed log output of the execution times per action, this can be easily explained due to the reason that the actions take more time to be executed depending on the action performed and the number of items used. For both models the execution times can be explained as follows:

- **Sequential model:** In the sequential model, the amount of items is taken from a fixed interval, corresponding to the stock size. Depending on the actual size, especially the number of items for stock takes or relocations is rather high compared to the stochastic model, as the

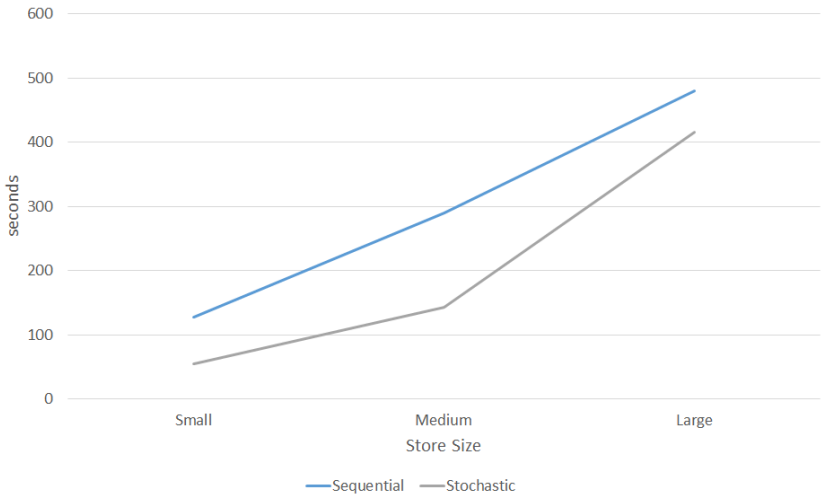


Figure 5.6.: Average execution times in seconds per store size. The larger the store, the more items need to be handled when performing actions and the higher is the amount of time the execution needs.

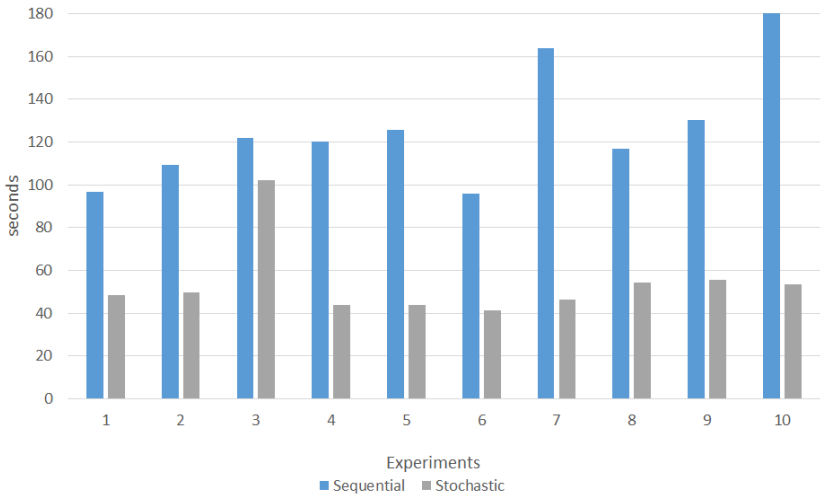


Figure 5.7.: Execution times per experiment for a small store. Due to the differing item set sizes, the stochastic model performs better than the sequential model.

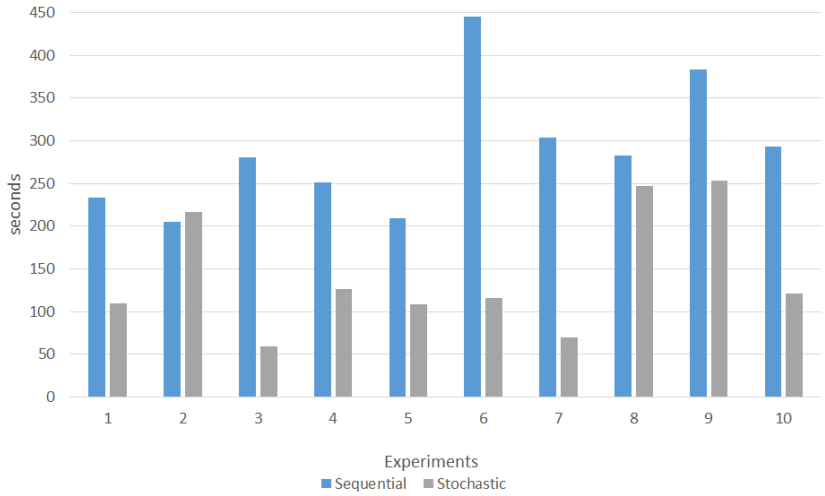


Figure 5.8.: Execution times per experiment for a medium store. Some actions take more time than others, and this explains the varying execution times of the stochastic model.

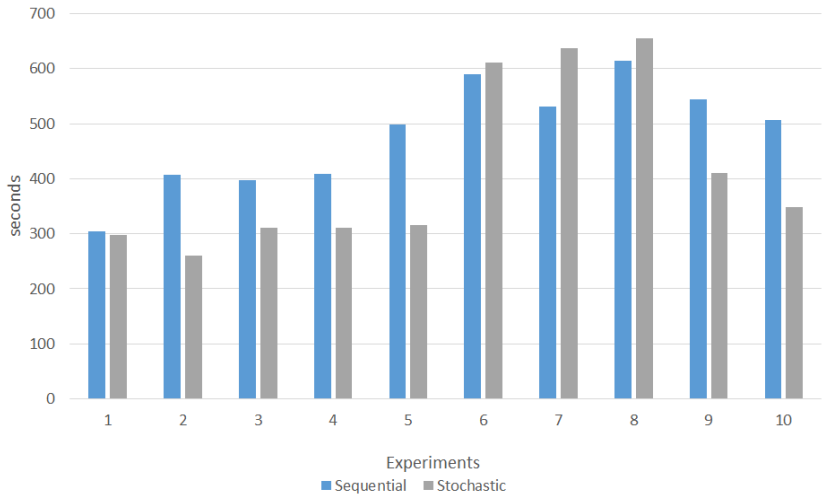


Figure 5.9.: Execution times per experiment for a large store. Again, some actions need more time than others, so the execution times of the stochastic model are in some cases even higher than the sequential model for a large store size.

amount of items used is much higher than in the stochastic model. The used training set showed that, different to the sequential model, only a small amount of items is relocated, usually only what one staff member can carry at once. Also the stock takes show that with one stock take not most of the stock is detected on one location, but rather only a small part. This can be explained with the usage of more persons working on a stock take with different devices to be merged after all persons have finished their RFID reading processes.

- **Stochastic supply chain model:** In general, it can be said that the execution of the action sequence generated by the stochastic model is faster than the sequential model. However, one can see in Figure 5.9 that the execution time is even longer than the sequential model in three cases. Analysis of the log output of the executions of the stochastic model showed that these long execution times are due to a high amount of performed inbounds, which take approximately ten times longer to execute at the web service than the other processes. Why the trained stochastic supply chain model suggested such a high amount of the same action consecutively will be discussed in the next section.

In addition to the execution times of sequences with the sequential and the stochastic model, Figure 5.10 shows the average execution times measured in the implementation of the models for each action with an amount of 100 items, including the calculation of the item sets to use. It can be seen that, except inbound and initialization of items, all of the implemented actions have approximately the same execution time for the same amount of items. Only initialization and inbound take much longer. These two actions are different to the others, as with these actions new EPCs must be calculated and added to the database, which means more workload than changing already existing data, as for the newly created EPCs it must also be checked if the same EPC already exists. The graphic also shows that especially the inbound takes more than twice as long as the initialization process for the same amount of items. Looking back to the description of a typical inbound process in Section 3.1.1, this action is also announced before the actual intake of the items is performed. The additional creation of the delivery note in a somewhat randomized way explains the longest execution time of all actions.

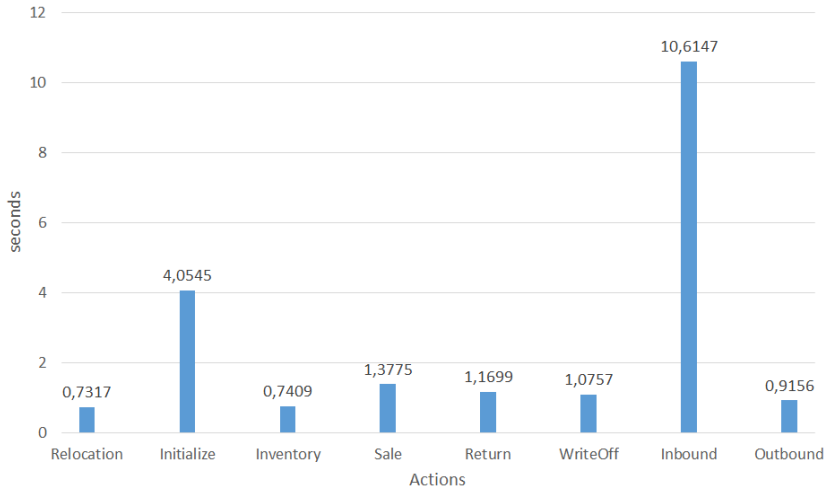


Figure 5.10.: Average execution time for each action in a database with medium size (approximately 5,000 items).

Compared to the manual creation of data to apply particular test cases, both models proved to be faster. The process of preparing only one inbound manually (see example in Section 5.1) would take at least 5 minutes alone, regardless of the size of the system's database. In that time, both models would be able to execute a sequence of at least 100 actions in a large database. However, there might be reasons when a manual execution is preferred, even when the execution of one of the models would be significantly faster. This would be in case very specialized data is needed which cannot be covered by the data generated with the model framework, for example when particular EPCs are needed.

## 5.3. Application

In this section, the data created by the models is discussed. The models will be applied to both use cases explained in Section 2.3.1, to fill the system with needed data as well as for various performance tests on the supply

chain management system. The models will be evaluated regarding the quality as well as their suitability for performance tests.

In the following, both models will be applied to the system for the following tests:

**Data quality:** We will compare the generated data to the analysis of the two previously presented data sets and evaluate the quality and drawbacks of test data generation using the models.

**Performance tests:** The system will be confronted with both a large database size as well as long sequence sizes to simulate large workloads to find out the limits of the system and the data generation framework. This also covers the execution of multiple instances of the implemented framework on the system to examine the web service's and database's responses.

**Coverage:** The achievable path coverage of both models will be discussed.

### 5.3.1. Data Quality

To compare the data generated by the models to the realistic training set, on both models experiments under the same conditions were applied. For this analysis, the stochastic supply chain model was trained with training set 1. As training set 2 with 12 sequences showed nearly the same results regarding the action distribution, the number of experiments was set to 12 experiments each in a medium-sized test database. The sequence length was generated automatically by the stochastic model based on the Poisson distribution for the sequence length, that is, the average lies around 255 actions per sequence, but varied between 200 and 270 actions. The number was recorded and the same sequence lengths were used for the sequential model, giving an overall number of approximately 2,700 actions to be executed for each model. In addition, the number needed to initialize in each experiment was set to 100 items for the sequential model.

Figure 5.11 shows the distributions of the executed actions during the experiments, compared to training set 1. Training set 2 is not shown in the chart for an easier interpretation, as training set 1 and 2 proved to be very similar (see Figure 5.5). It can be seen that the results of the sequential

model are different to the presented training sets. This was to be expected and can easily be explained as assumptions were made for the actions based on a single item's lifecycle and not on a realistic workflow regarding a full stock. We can see that the mostly performed action is a relocation, followed by stock takes and exactly 12 initialization actions. When looking back at the abstract workflow in Figure 3.2, it can be explained that relocation has four and stock take two vertices instead of only one with sale and return and is therefore expected to be performed more often. Write off, inbound and outbound are executed zero times as these actions are not included in the sequential model. The number of initializations corresponds to the number of sequences executed, as an initialization is always only the first action in the sequential model.

Some experiments while using the stochastic model not depicted showed surprising results - with the model being trained with data set 1, it was expected that action distribution would be similar to the chart of Figure 5.1. However, this was not the case and the experiments with the stochastic model showed a nearly equal amount of stock takes, sales and especially inbounds. To find the reason for that, a closer look at the trained initial vector and transition matrix was taken. The initial vector showed a rather high probability of about 32% for inbound and 31% for relocation as a start state for the model - this means that especially inbounds and relocations were the first actions in the sequences of the training sets. Additionally, in the transition matrix it becomes visible that in the training set, a few selected actions are executed after one another very often and therefore are weighted with a higher percentage. Especially for inbound the probability to perform another one in the next step as well is over 80%. Another example is the stock take, with the probability to perform another one on any location is in sum even over 90%. This explained the unexpected behavior, as the model is very likely to stay in the same state for the whole sequence. One possible reason why this happened is that the training set consists of a very limited number of sequences, where one sequence could be all actions over one day. Some actions are probably happening only at a certain time, for example any number of inbounds is often the first thing performed in the morning.

To overcome this problem the initial vector was modified to test how the results vary when noise is added to distort the probability distributions. For

that, in the initial vector the probability of the actions responsible for the unexpected results, such as inbounds, were decreased and the probability of frequent actions, such as sales, increased. With these changes applied, the better results presented in Figure 5.11 can be achieved, as the probability of of the model being stuck in a loop of the same action or limited action set is decreased. With this modification, the action distribution is very similar to the training set. This also shows that adding noise really is an easy and quick way to change the model parameters for a wider range of use cases.

For the future this means that the set to train the stochastic supply chain model must be selected even more carefully instead of only taking it from the database of a deployed system and adding a reasonable amount of noise. Possibly some sensible alterations have to be made as well to prevent the stochastic model from being stuck in these loops with extraordinarily high probabilities for the same action following each other if selected as initial state. In addition it could also be helpful to train the stochastic model with a much larger training set, as 115 sequences are probably not expressive enough to get a meaningful and realistic model.

Figure 5.12 shows the average number of items per action in the sequential and the stochastic model. While the results of the stochastic supply chain model are fitting very well to the item numbers of training set 1 (see Figure 5.2), the item distribution of the sequential model shows a completely different result. This is again due to the reason that the percentages used for this model are predefined - they do not fit to the results of the training sets, but they are correct according to the defined boundaries. For example, a stock take on a location detects between 70% and 100% of all items on a location. As this was a database of medium size (approximately 5,000 items), we can assume that on sales floor and back room are approximately 2,500 items each. The results were pretty close to that and the same applies to the other actions as well.

### 5.3.2. Performance Tests

As already discussed in Section 2.3.1, with the models it should be able to execute various performance tests on the system under test. With the model, we concentrated on two types types of tests: endurance tests and load tests.

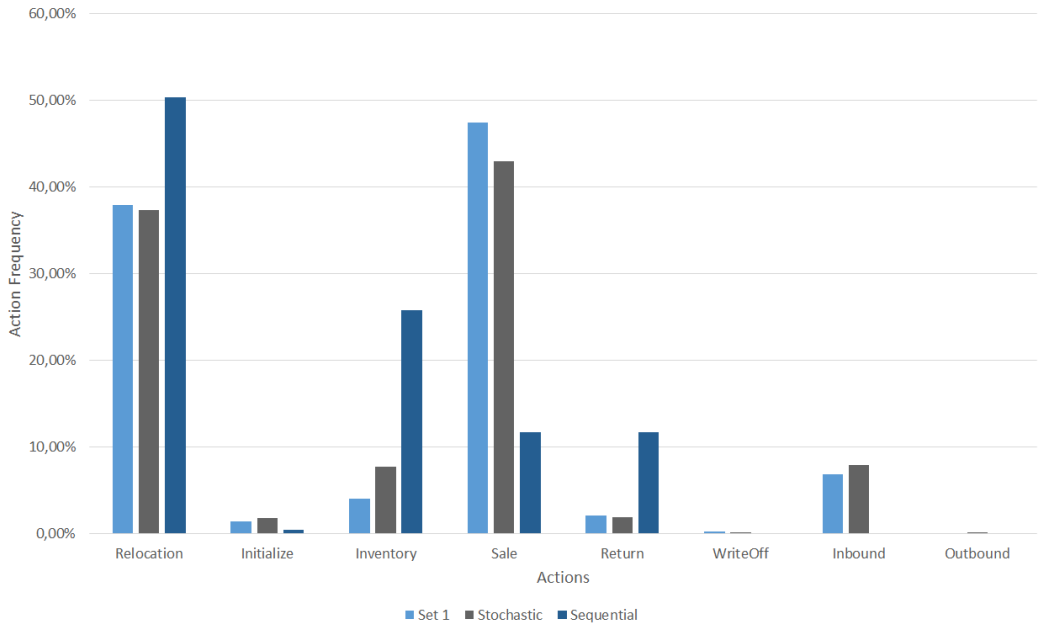


Figure 5.11.: Comparison of the relative number of actions executed. The chart shows that, for a well defined transition matrix and initial vector in the stochastic model, the action distribution is very similar to a realistic data set. Due to the structure of the sequential model, the obtained results are very different to training set 1 and the stochastic model.

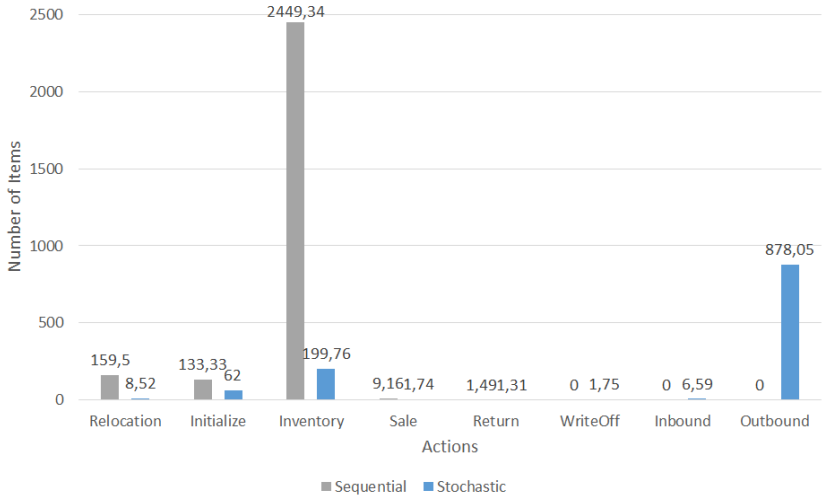


Figure 5.12.: Comparison of the average number of items used in both models for each action for a medium store.

The third discussed performance test type, stress tests, are also performed implicitly with the other two tests. In the following, the preparation and results of the performance tests are presented. Basically, with the test setup described in the following, statements about all three system properties can be made. As the goal was to find the limits and bottlenecks of the system, the idea was to confront the test system with data sets as large as possible and to find out what happens when the data generation models are applied. To find these limits, two approaches were used.

The first approach was to instantiate a database of a much larger size than the defined large store with around 10,000 items and then try to perform a sequence much larger than the calculated average of 255 of the stochastic model for training set 1, namely a sequence length of 5,000. Starting with a database size of 40,000 items, both the sequential and the stochastic model quickly reached a limit when confronted with timeouts from the test system's database, as such a high percentage of items could not be handled within the defined timeout interval of the system. As an example, in the sequential model, a stock take should have been performed with approximately 19,000 items, and this request could not be fully executed in

the defined timeout interval of one minute. While with the stochastic supply chain model the average amount of items was usually smaller, at some point, especially when already calculation-heavy actions such as inbounds should be performed with an unusually high amount of items, the web service's requests to the database still returned timeout exceptions. From the 5,000 sequence steps, the execution usually had to be stopped after a maximum of 100 steps in the sequential model and around 1,000 steps in the stochastic supply chain model.

From this point on, the data sets were halved until at least one of the models could execute a sequence of length 5,000. With a database size of 15,000 items at most the stochastic model brought reasonable results - the 5,000 consecutive actions could be finished in approximately 150 minutes. However, for the sequential model with stock takes of at least 70% of the stock's items this database size was too large for the web service/database to get a reasonable performance.

The second approach was to create at least two instances of the model framework and run them simultaneously. The test system used for these experiments was a database of medium size with approximately 5,000 items in stock. However, a few experiments with this setup, using both the stochastic and the sequential model, were not very successful. While a consecutive execution with only one action for the web service at a time was successful up to some point for a large data set, the maximum of actions in a sequence that could be executed consecutively was between 10-12 actions. At that point the response from the web service was again a timeout exception when it tried to access the underlying database. This was due to the reason that while the service could schedule the actions quite well, simultaneous requests to get data from the database led to performance problems and the actions could not be performed in a reasonable amount of time. However, adding wait times of a few seconds during the execution of each sequence brought improvements as in a realistic scenario it would be unlikely that in such a short amount of time, this means in a range of milliseconds to seconds, so many different actions would be performed in a deployed RFID system.

Looking at the results from both performance test approaches, it can be said that the stochastic supply chain model as well as the sequential model can

handle large amounts of data quite well. While the duration of over 2 hours may sound like a long time, it needs to be said that a manual execution of 5,000 different actions would never be possible in that time. More so, this would probably need days. In addition, setting up a test environment for this type of tests would be very tedious and time-consuming when RFID-specific issues, such as tag preparation or the setup of applications on multiple devices need to be considered.

### 5.3.3. Coverage

During the evaluation of the model framework a few aspects on coverage were considered, namely if sensible measurements of coverage are possible when running the models on the system under test. After research on different coverage criteria, such as graph coverage, statement coverage [2] or coverage regarding lines of executed code, it became clear that testing the code coverage of the SUT would be of little use because of two reasons. First, the system under test acts as a black box and other than the interfaces, we have no information about the underlying code. Second, with the evaluation only a limited action set on one level of a supply chain management system is considered - even with code available to test for coverage, it would be difficult to make a qualitative statement about coverage.

This is why a different approach was taken. One possible criterion for coverage tests was identified as path coverage regarding the actions executed in the model. This means the models are analyzed regarding their ability to cover all theoretically possible action paths in a given number of executed actions. We can define the number of possible action permutations as $n^k$, where $n$ defines the size of the action set and $k$ the path length, meaning the number of actually executed actions.

Consider a simple example with two actions $a_1$ and $a_2$, which for example could be two $action \times location$ pairs in the stochastic model. For a path of length $k = 3$ this would give the following $2^3 = 8$ path permutations:

1. $a_1 \rightarrow a_1 \rightarrow a_1$
2. $a_1 \rightarrow a_1 \rightarrow a_2$
3. $a_1 \rightarrow a_2 \rightarrow a_1$

4. $a_1 \rightarrow a_2 \rightarrow a_2$
5. $a_2 \rightarrow a_1 \rightarrow a_1$
6. $a_2 \rightarrow a_1 \rightarrow a_2$
7. $a_2 \rightarrow a_2 \rightarrow a_1$
8. $a_2 \rightarrow a_2 \rightarrow a_2$

Of course, this example case could be extended to any amount of transitions and path lengths as well as testing the paths leading to a defined action. The coverage goal with the models would be to pass through all these paths with as few experiments as possible to ensure that also unlikely or even impossible paths are considered and tested on the SUT to evaluate if the system can handle such cases correctly. Based on this information, the two models can be compared regarding their achievable coverages.

**Sequential model:** The sequential models executes actions in a consecutive order. The order in which these actions can happen is predefined based on specific system knowledge, whether this is achieved from specifications or other knowledge about the application domain. However, specifications often only cover "allowed" system states and transitions between this system state. Therefore, when defining and applying this sequential model, it is highly possible that not all paths in the system can be tracked, regardless of the number of performed experiments. This limits the achievable coverage, as some states in the path might not be reachable as, by definition due to the workflow specification, certain paths are not allowed and hence not shown as possible actions in the sequential model. Extending the model to cover these cases would be a tedious and complex task.

**Stochastic supply chain model:** In contrast to the sequential model, the stochastic model learns state transitions from given data sets, considering all possible action orders. Theoretically, with a sufficiently large number of experiments, a path coverage of 100% is possible for a finite path length, assuming the transition matrix consists of non-zero values. Based on the training data set it is likely that the transition matrix contains zero-values or values nearly zero with the effect that the stochastic model is not able to reach certain states and perform particular transitions. However, with the structure given due to the definition of the stochastic supply chain model, it can be easily adapted

to meet the requirements for path coverage with respect to unlikely or impossible transitions by adding noise to the transition matrix, the initial vector, or even both. With the added noise, unlikely paths get a higher probability to be passed and impossible state transitions will be reachable, which reduces the amount of experiments for high path coverage.

Summarized, coverage testing with the sequential model is highly dependable on the specifications from which the model is created. Extending it to cover cases not specified is a time-consuming, complex task which is not always possible. Contrary to that, the stochastic model provides a flexible structure to consider unlikely or impossible paths as well by modifying the transition probabilities between states. Therefore, with this model it is possible to reach a reasonable coverage with a sufficiently large set of executed experiments.

## 5.4. Usability and Limits

In the previous sections results of the data generated with the model framework and the two models were presented. With the previous statements it is possible to compare the models to each other and to the manual process of generating test data for various test types. The presented model framework with the implementation of both the sequential and the stochastic model proved to be an easy and quick way to generate test data to fill the underlying system with working data to perform various test cases. In the following, the usability of the two models will be described.

### 5.4.1. Sequential Model

With application of the sequential model to the system under test, it is possible to fill its database with data very quickly. With a few clicks to set how many actions should be executed, data is generated according to the presented action flow of Figure 3.2. For this limited set of actions, the model generates data following the presented abstract action graph on the

various locations, based on preset values for the number of items to use. In addition, these defined values (percentages) are easy to change for a different outcome after the execution. This model turns out to be effective in case only a particular amount of random data is sufficient to complete test cases. For the following example test case, no realistic data would be needed to verify the functionality.

**Example test case:** For a report in the web application, table filters exist. To test these filters, a random amount of data is needed to test if they work accordingly and filter out the correct data. A possible filter criteria could be that only items of a piece of clothing with size M is displayed in the report. However, the filters should still work if a large amount is present in the reports.

While this model turned out to be very simple to use, it also faces some drawbacks, mainly regarding the versatility of test cases it can be used for. In the following, the main problems of this model encountered during usage can be listed as:

*Adaptability and extensibility*: With the sequential model, only a strict consecutive execution of the given actions is possible. For each new action or location that should be added, new nodes and vertices have to be added manually in the workflow. Here the main problem is that probabilities for new actions, the locations on which actions are performed and the number of items used for these actions need to be assumed based on domain knowledge and added to the workflow by hand. In addition, knowledge about the system specification is needed to get a qualitative data generation model. This is not maintainable for larger workflows.

*Workflow does not reflect realistic behavior*: As this model does not make use of realistic data to simulate the process steps or estimate the stock size, it is difficult to make assumptions about the correctness of the generated data for test cases which aim to test the coverage of the modules or system components. For a realistic representation, the model must be changed according to the use cases each time. In addition, this model does not cover any time management. The actions are registered in the test system's database with the execution time - in a realistic scenario,

actions are performed throughout a whole day, not a large amount at nearly the same time.

*Missing flexibility*: As already mentioned, the sequential model is not flexible enough for different use cases. It is difficult to create test data for one specialized use case, for example when only relocation data is needed. Either the model workflow is changed in the source code or only so many steps are executed manually until the desired amount of data is present. The sequential model is suitable for simple use cases when any amount of random data is needed, but applying changes to the model for different use cases or high path coverage is tedious and not flexible enough. This is due to the reason that the workflow only allows the execution of actions following each other as set, but for example a sale action is not necessarily followed by a return in every case. Adding more variations to this model is time-consuming and error-prone.

### 5.4.2. Stochastic Supply Chain Model

With the stochastic model, data is generated in a more sophisticated and flexible way. After interfaces for more actions are added, these new actions only need to be added to the codebook for easy inclusion into the stochastic supply chain model and then they are automatically considered in the training process and the following data generation. By adapting the training data set to the needs of the tester, the data generation model provides a realistic amount of test data, scaled to the size of the test system. When other components of the test system should be tested with data as realistic as possible, only a sufficiently large sequence of actions needs to be executed to have realistic, but yet more flexible test data than a simple database backup. In addition, the stochastic model can easily be adapted in case only a subset of the actions is needed by only reducing the available actions in the codebook.

However, while this model seems to be suitable for a wider range of use cases than the sequential model, a few problems and limits can be listed:

*Actions are location-dependent*: Due to the design of the *action* $\times$ *location* pairs for the initial vector and the transition matrix of the stochastic

supply chain model, all actions are assumed to be executed on or for a certain location. However, this is probably not applicable for all actions of a supply chain and therefore exceptions have to be considered when calculating the Markov model's parameters. A simple workaround would be to add a location to the action nonetheless when analyzing the training data and to ignore it when the action should be executed. It would be desirable to find a better way to handle these in a better way than by defining a set of exceptional rules.

*"Stationary states"*: If the training data set is not analyzed carefully enough before using it to train the stochastic model, it can happen that the model is not likely to "leave" a certain state or in a loop of only a few states with a very high selection probability compared to all other states. As shown in Section 5.3, this can lead to results which are not conform to the expected data based on the used training set. One future goal would be to find better metrics for the training step of the model instead of adding noise to prevent these cases.

*Constrained training data*: To get expressive test data corresponding to a realistic data set, it was not enough to just take data from a deployed system and to limit it to only the actions covered by the model framework. This led to the problem of the stochastic supply chain model staying in a loop of consecutively executing the same action again and again. Therefore, it is not always sufficient to only train the stochastic model with the set. Some thoughts need to be spared to fit the training set to the tester's need to prevent this from happening, as in this field of application it might not always possible to get a very large set of training data for the model.

*Missing temporal behavior*: As already mentioned for the sequential model, the stochastic model does not support realistic temporal behavior as well, as this was considered out of this work's scope. Therefore, also with the stochastic model framework executions are only added with the execution time as timestamp. While the current execution behavior of the action sequences generated with the stochastic model is sufficient for the discussed test cases, an RFID system cannot be simulated realistically without the corresponding temporal behavior. For example, goods could be delivered twice a week in the morning. After that, an inbound is performed on the arrived goods. In future work, better metrics for temporal behavior should be considered.

# 6. Conclusion

This work described the challenges of distributed systems, like heterogeneity, concurrency or scalability, with a special focus on testing in the field of RFID systems for supply chain management, one prime example for such a demanding test environment. When discussing the structure and properties of RFID systems, it became clear that in this environment a wide range of test cases cannot be performed in a meaningful way without corresponding test data.

As manual test data generation usually is time-consuming and error-prone, the goal of this work was to find a model suitable for test data generation for a selected range of test cases in supply chain management systems: to fill the system with realistic data to bring it into a particular state for tests on various system components, as well as for the generation of large amounts of data for performance tests. To model the processes of a supply chain, two approaches were selected after research on already existing frameworks, evolutionary algorithms and various probabilistic models: a sequential model and a stochastic model.

The first model, the *sequential model*, is an abstract data generation model using the lifecycle of a single item mapped to a sequence of actions for sets of items. This is for use cases where just a random amount of the most important data is needed to test particular parts of a supply chain system. The second model, the *stochastic supply chain model*, is one kind of probabilistic model which analyzes data from deployed supply chain managements to calculate realistic action and item distributions for data generation.

In addition to the design, both models were implemented as a C# framework to be applied to a deployed test system to evaluate the quality and usability according to the defined use cases. As a comparison, a data set was taken

from a deployed supply chain management running in a fashion store was taken to compare the generated data to it. It was shown that the sequential model does not produce very realistic data, but can still be used for certain test cases without the need for that. One main advantage was the easy and quick application to get results. Before running the stochastic supply chain model, preparations need to be made to create a data set to train the model. As a result of that, this model performed better when the generated data is compared to the store data set. However, a few problems were encountered due to the selected training data set, sequences with only a limited set of actions were created according to very high transition probabilities back to the same state in the transition matrix. Overall, the stochastic model still showed a better runtime, coverage and a higher versatility for functional and performance tests, since no parameters had to be set based on expert knowledge.

While both models performed quite well for a few selected cases of application, possible future work can be discussed. As the sequential model follows a very strict structure of a single item's lifecycle, it would be very difficult to extend it to more realistic scenarios. However, the stochastic model provides much space for improvements to extend the data generation process for a wider range of use cases, for example by extending it to cover all possible actions in the supply chain system. By providing one model for each node in the system, such as manufacturers, distribution centers and stores, each with their own specific action set and communicating with each other, the complete supply chain could be simulated. Additionally, future work should also cover the inclusion of realistic temporal behavior to get a test system as close as possible to the later use cases as well as better metrics to test for the presented coverage tests.

Concluding it can be said that test data generation is a very challenging factor in the software development process, especially when the procedure should be automated and data should be applicable to a wide range of use cases. Model-based test data generation is one way to approach this topic, however, goals and use cases must be specified precisely to be able to create an appropriate data model fitting to the tester's needs.

# Appendix

# A. Method Summary

## A.1. General

*AddRandomEpcsToList*: EPC lists are needed to tell the system which items should be used for a store process. This method is used to add a given number of EPCs existing in the database to a list. The EPCs are selected from either one specified locations or all registered items in the database. By defining a random number with its maximum being the number of items on a location or in the store with C#'s `Random` method, the EPCs are selected until the desired amount is reached.

*GenerateSGtin*: This method is mainly used in case that new EPCs should be added to the store, for example with an inbound or initialization process. In RFID store management systems, each EPC is calculated from a GTIN, a unique identification number for articles. The `GenerateSGtin` method takes two input parameters, a string containing a GTIN (which, in this case, must exist in the test database as product masterdata) and an integer representing the serial number the generated EPC should have. The calculation of the EPC is done according to [23].

*SaveModelParameters*: Each time a training of the Markov model is completed, the trained parameters, meaning the model's initial vector and matrices as well as the Poisson distributions for the sequence length and number of items for each action are stored in three files, which are saved in the `Model Data` folder of the application.

*GetActions*: This method is used to get a list of all actions with the possible locations supported by the Markov model and is used in the model implementation to easily define the codebook (described in Section 4.4) used for handling the actions in the model and to define the dimensions of vectors and matrices.

*InitializeStore*: initializes a given number of items in the database to create a store environment with items on each location.

## A.2. Service Interface

*StockTaking*: Send a stock take action of $n$ EPCs on a location $L_k$ to the system.
*Relocation*: Send a relocation action of $n$ EPCs to a location $L_k$ to the system.
*WriteOff*: Send a write off action for $n$ EPCs to the system.
*Initialize*: Send an initialization request for a list of $n$ new EPCs to the system.
*Inbound*: Sending an inbound to the system consists of two steps:

- Preparing the inbound and registering it in the system. Items which are meant to be delivered to a store are advised as incoming.
- When the items arrive, they are added to the store's stock. The two possible scenarios here are that the inbound is either complete (all of the advised items are added) or incomplete (not all of the advised items added).

*Outbound*: Send an outbound with $n$ EPCs to the system.
*Sale*: Send a sale action of $n$ EPCs to the system.
*Return*: Send a return action of $n$ sold EPCs to the system.

## A.3. Data Interface

*EpcInDatabase*: Checks if a given EPC already exists in the test database.
*GetGtinsInDatabase*: Gets a list of the test database's GTINs / products.
*GetEpcsInDatabase*: Gets a list of all EPCs in the test store's stock.
*GetEpcsOnLocation*: Gets a list of all EPCs on a given location.
*GetEPcsWithProductData*: Gets a list of all items with the corresponding product data.
*GetGtin*: Gets the product GTIN of a given item.

*GetNumberOfItemsInStock*: gets the stock size, meaning the number of items in stock.
*GetSoldEpcs*: Gets all EPCs which are marked as sold in the test database.
*GetLocations*: Gets a list of all locations of the test store.
*GetBusinessSteps*: Gets a list of all actions which can be performed in the store system.
*IsBusinessStepInDatabase*: Is true if a certain action is possible in the test system and false otherwise. This is important as training data can contain more actions than can be handled by the test system and these need to be filtered out.
*GetRandomSupplier*: Gets a random supplier from the test system. This method is needed for the outbound process, as the supplier where items are sent back to needs to be defined.
*GetRandomGtin*: When a number of items need to be created, they need to have product data. This method gets a random GTIN registered in the database.

## A.4. Store Functionality

*PerformInbound*: Generates a list of $n$ EPCs with their corresponding GTINs in the database and performs an inbound of these items on a given location.
*PerformOutbound*: Gets a number of $n$ random items in the database via the `AddRandomEpcsToList` method described in Section A.1 and performs an outbound with these items.
*PerformSale*: Gets a number of $n$ random items from a sales floor location and calls the corresponding web service interface method.
*PerformReturn*: Gets a number of $n$ random items which are marked as sold in the test system and calls the corresponding web service interface method to return these items.
*PerformRelocation*: Gets a number of $n$ random items from the stock and performs a relocation to a given location.
*PerformInventory*: Gets a number of $n$ random items from a given location. If $n > \#items$ on the location, the difference $n - \#items$ is taken from

the other locations in the store. On the $n$ selected items, a stock take is performed.

*PerformWriteOff*: Gets a number of $n$ items on a given location and performs a Write Off.

*CreateInitialEpcList*: Creates a list with $n$ EPCs not in the test system's database, but with correct product data which can be handled by the test system.

*InitializeItems*: Calls the previous method to create a list of $n$ new EPCs and performs an initialization.

## A.5. Stochastic Supply Chain Model

*CreateTrainedMarkovModel*: In this method, the Markov model is trained based on given store data. In the first step, the sequence distribution and item distributions are calculated from the training sequences' lengths and number of items used for each $action \times location$ pair in the input file.

After the calculation of the distributions important for a realistic simulation, the data contained in the sequence file must be translated according to the previously mentioned codebook so that the Markov model can be trained. As Accord.NET provides a class called `Codebook` for exactly this purpose, after the initial declaration, we only need to translate the sequence with the `codebook.Translate(data)` command.

As described previously, we define the not needed emission matrix $B$ as the identity matrix. Then $A$ and $\nu$ are calculated based on the Algorithms 1 and 2. Then the model is initialized with the two matrices $A$ and $B$ and the vector $\nu$.

At last, the currently trained data (Markov model and distributions) is exported to the application's `Model Data` folder for future usage.

The most important methods of this class are:

*TrainTransitionMatrix*: This method gets the training data as input and creates the transition matrix $A$ according to Algorithm 1.

*UpdateInitialVector*: This method updates the initial vector $\nu$ based on the transition matrix $A$ according to Algorithm 2.

*ImportModelParameters*: This method is used to create a Markov model without training if data is already at hand from a previously training. Instead of training, data contained in this file will only be passed to the model and can be used instantly. However, this method has no metrics to fit the imported training data to a test store size different than the trained one. This means if the test store is not of an equal size compared to the imported training data, wrong system behavior or inconsistencies can occur.

*GetSequence*: This method generates a sequence of actions from the trained model, either of a sequence length based on the Poisson distribution or of manual length.

*ExecuteActions*: This method represents a dictionary for the execution instructions depending on the *action* $\times$ *location* input string. An example would be the input state *Relocation* $\times$ $L_3$, then the called method would be `PerformRelocation` on the location $L_3$. Naturally, this method only contains actions covered by the Markov model.

*ExecuteSequence*: This method gets the sequence generated by the `GetSequence` method and executes the contained actions consecutively based on the dictionary from the previous method.

## A.6. Distribution Classes

*CalculateAverageSequenceLength*: Gets passed the training sequences, counts the length of each and calculates the Poisson distribution from these values.

*GenerateObs*: Calculates a random integer value based on the underlying distribution.

*CalculateItemDistributions*: For each *action* $\times$ *location* pair, this method calculates the Poisson distribution's $\lambda$.

*FitDistToStoreSize*: As the test store's size can be different, the distribution is calculated relatively to the test store size.

# List of Abbreviations

**CSV** comma-separated values
**DC** Distribution Center
**DTMC** Discrete Time Markov Chain
**EA** Evolutionary Algorithm
**EM** Expectation Maximization
**EPC** Electronic Product Code
**ERP** Enterprise Resource Planning
**GTIN** Global Trade Item Number
**GUI** Graphical User Interface
**HMM** Hidden Markov Model
**HTTP** Hypertext Transfer Protocol
**RFID** Radio Frequency Identification
**SUT** System Under Test
**UML** Unified Modeling Language
**WCF** Windows Communication Foundation
**WPF** Windows Presentation Foundation

# Bibliography


[1] *Accord.NET Framework*. `http://accord-framework.net/`. last accessed 29th April 2016 (cit. on p. 51).

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN: 9780521880381 (cit. on p. 74).

[3] D. Ashlock. *Evolutionary Computation for Modeling and Optimization*. Springer Publishing Company, 2006. ISBN: 9780387221960 (cit. on p. 29).

[4] M. Beyer, W. Dulz, and F. Zhen. "Automated TTCN-3 Test Case Generation by means of UML Sequence Diagrams and Markov Chains." In: 12th Asian Test Symposium. IEEE, Nov. 2003, pp. 102–105 (cit. on p. 28).

[5] N. Blumhardt. *Stateless .NET State Machine Framework*. `https://github.com/dotnet-state-machine/stateless`. last accessed 29th April 2016 (cit. on p. 50).

[6] Doungsa-ard C., K. Dahal, A. Hossain, and T. Suwannasart. "Test Data Generation from UML State Machine Diagrams using GAs." In: vol. 4909. International Conference on Software Engineering Advances. IEEE, Aug. 2007 (cit. on p. 27).

[7] L. Cacciari and O. Rafiq. "Controllability and observability in distributed testing." In: *Information and Software Technology* 41(10-11) (1999), pp. 767–780 (cit. on p. 17).

[8] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. 4th ed. Addison Wesley, 2005. ISBN: 0321263545 (cit. on pp. 9, 10).

[9] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. 2nd ed. John Wiley & Sons, 2006. ISBN: 9780471241959 (cit. on p. 34).

[10] A. Deepak and P. Samuel. "An evolutionary multi population approach for test data generation." In: World Congress on Nature & Biologically Inspired Computing. IEEE, Dec. 2009, pp. 1451–1456 (cit. on p. 29).

[11] S. Dustdar and S. Haslinger. "Testing of Service Oriented Architectures - A Practical Approach." In: *Object-Oriented and Internet-based Technologies*. Lecture Notes in Computer Science Vol. 3263. 2004, pp. 97–109 (cit. on p. 1).

[12] K. Finkenzeller. *RFID Handbook: Fundamentals and Applications in Conctactless Smart Cards, Radio Frequency Identification and Near-Field Communication*. 3rd ed. John Wiley & Sons, 2010. ISBN: 9780470695067 (cit. on pp. 11, 13).

[13] Apache Software Foundation. *Apache log4net*. `https://logging.apache.org/log4net/`. last accessed 29th April 2016 (cit. on p. 49).

[14] S.J. Galler and B.K. Aichernig. "Survey on test data generation tools." In: *International Journal on Software Tools for Technology Transfer* 16 (2014), pp. 727–751 (cit. on p. 27).

[15] S.J. Galler, M. Weiglhofer, and F. Wotawa. "Synthesize it: from Design by Contract™ to Meaningful Test Input Data." In: 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM). IEEE, Sept. 2010, pp. 286–295 (cit. on p. 15).

[16] J. Gao. *Component Testability and Component Testing Challenges*. `http://www.engr.sjsu.edu/gaojerry/report/testabilities.pdf`. last accessed 4th May 2016. 2000 (cit. on p. 17).

[17] Enso Detego GmbH. *Detego Suite*. `http://www.detego.com/en/products/detego-suite.html`. last accessed 29th April 2016 (cit. on pp. 5, 44).

[18] M. Goller. "Probabilistic Modeling in RFID Systems." PhD thesis. Graz University of Technology, 2013 (cit. on p. 2).

[19] O. Henninger, M. Lu, and H. Ural. "Automatic Generation of Test Purposes for Testing Distributed Systems." In: *Formal Approaches to Software Testing*. Vol. 2931. Lecture Notes in Computer Science. 2004, pp. 178–191 (cit. on p. 28).

[20] N. Henze. *Stochastik für Einsteiger*. 8th ed. Vieweg+Teubner, 2010. ISBN: 9783834808158 (cit. on p. 40).

[21] R.M. Hierons. "Testing a distributed system: generating minimal synchronised test sequences that detect output-shifting faults." In: *Information and Software Technology* 43(9) (2001), pp. 551–560 (cit. on p. 28).

[22] R.M. Hierons and H. Ural. "Overcoming controllability problems with fewest channels between testers." In: *Computer Networks: The International Journal of Computer and Telecommunications Networking* 53.5 (2009), pp. 680–690 (cit. on p. 17).

[23] BAR CODE GRAPHICS INC. *EPC-RFID INFO*. `http://www.epc-rfid.info/sgtin`. last accessed 29th April 2016 (cit. on p. 85).

[24] W. Khreich, E. Granger, A. Miri, and R. Sabourin. "A Survey of Techniques for incremental Learning of HMM Parameters." In: *Information Sciences* 197 (2012), pp. 105–130 (cit. on pp. 29, 35).

[25] M. Kuperberg. "Markov Models." In: *Dependability Metrics*. Vol. 4909. Lecture Notes in Computer Science. 2008, pp. 48–55 (cit. on p. 34).

[26] S. Lahiri. *RFID Sourcebook*. IBM Press, 2005. ISBN: 0131851373 (cit. on pp. 12, 13).

[27] G.I. Laţiu, O.A. Creţ, and L. Văcariu. "Automatic Test Data Generation for Software Path Testing using Evolutionary Algorithms." In: Third International Conference on Emerging Intelligent Data and Web Technologies (EIDWT). IEEE, Sept. 2012 (cit. on p. 29).

[28] Microsoft. *Windows Communication Foundation*. `https://msdn.microsoft.com/en-us/library/dd456779.aspx`. last accessed 29th April 2016 (cit. on p. 43).

[29] Microsoft. *Windows Presentation Foundation*. `https://msdn.microsoft.com/en-us/library/ms754130.aspx`. last accessed 29th April 2016 (cit. on p. 46).

[30] G.J Myers. *The Art of Software Testing*. 2nd ed. John Wiley & Sons, 2004. ISBN: 0471469122 (cit. on pp. 14–16).

[31] C. Pichler. "Software Test Automation in the Field of RFID." MA thesis. Graz University of Technology, 2015 (cit. on p. 3).

[32] L.R. Rabiner. "A tutorial on hidden Markov models and selected applications in speech recognition." In: *Proceedings of the IEEE*. Vol. 77. 2. IEEE, 1989, pp. 257–286 (cit. on pp. 29, 34, 51).

[33] L.R. Rabiner and B.H. Juang. "An introduction to hidden Markov models." In: *ASSP Magazine, IEEE* 3.1 (1986), pp. 4–16 (cit. on pp. 29, 35).

[34] K Sigman. *Simulating Markov Chains*. Lecture Notes. 2007 (cit. on p. 35).

[35] A.S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. 2nd ed. Prentice-Hall, Inc., 2006. ISBN: 0132392275 (cit. on p. 9).

[36] H. Ural and D. Whittier. "Distributed testing without encountering controllability and observability problems." In: *Information Processing Letters* 88(3) (2003), pp. 133–141 (cit. on pp. 17, 18).

[37] K. Wiegers and J. Beatty. *Software Requirements*. 3rd ed. Microsoft Press, 2013. ISBN: 9780735679665 (cit. on p. 18).

[38] W. Zhang, D. Gong, X. Yao, and Y. Zhang. "Evolutionary generation of test data for many paths coverage." In: Chinese Control and Decision Conference (CCDC). IEEE, May 2010, pp. 230–235 (cit. on p. 29).

[39] Y. Zhang, D. Gong, and Y. Luo. "Evolutionary Generation of Test Data for Path Coverage with Faults Detection." In: vol. 4. Seventh International Conference on Natural Computation (ICNC). IEEE, July 2011 (cit. on p. 29).

[40] K. Zhou, X. Wang, G. Hou, J. Wang, and S. Ai. "Software Reliability Test Based on Markov Usage Model." In: *Journal of Software* 7(9) (2012), pp. 2061–2068 (cit. on p. 28).