FAKULTÄT FÜR !NFORMATIK

Testing the Performance of Complex System Simulations in the Production Automation Domain

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Gregor Dürr
Matrikelnummer 0527755

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Ao. Univ. Prof. Dipl.-Ing. Mag. Dr. techn. Stefan Biffl
Mitwirkung: Univ.-Ass. Mag. Thomas Moser

Wien, 30.11.2009

(Unterschrift Verfasser) (Unterschrift Betreuer)

Acknowledgement

My special thanks go to my supervisor Stefan Biffl, professor at the Institute of Software Technology & Interactive Systems (ISIS) at the Vienna University of Technology, for his support and advice. Particularly, I would like to thank Thomas Moser at ISIS for his encouragement and guidance throughout the work.

In addition, I would like to thank my parents for giving me the opportunity to attend the Bachelor Program at the TU Munich, the Master Program at the TU Vienna, and to spend a semester abroad at the NTNU Trondheim. Furthermore, I would like to thank my girlfriend Sigrid for organizing most of our upcoming journey round the world while I was writing my master's thesis.

Declaration

I declare that I have written my Master's Thesis independently. Only the indicated sources and aids were used. Parts of this work taken from other sources were cited in every case.

Wien, 30.11.2009 _____
Gregor Dürr

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit einschließlich Tabellen, Karten und Abbildungen , die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30.11.2009 _____
Gregor Dürr

Zusammenfassung

Bei komplexen automatisierten Produktionssystemen kann meist nicht vom Verhalten der Teilsysteme auf das Verhalten des Gesamtsystems geschlossen werden. Die zur Ermittlung des Gesamtverhaltens eines Systems notwendigen Erkenntnisse können aus Simulationen gewonnen werden. Neben der Güte der Simulationsergebnisse ist die Effizienz eines Systems ein ausschlaggebendes Qualitätskriterium. Dies gilt insbesondere bei einer großen zu testenden Parametervielfalt. Mögliche Parameter einer Produktionsstraße sind beispielsweise Produktionsstrategie, Fehlertoleranzstrategie und die Anzahl der zu fertigenden Produkte.
Allgemein dient Software-Testen als Instrument zur Steigerung der Qualität des zu testenden Produktes oder Services. Für einen vollständigen Test aller Anforderungen an ein Softwaresystem, müssten alle möglichen Szenarien in einem separaten Testfall abgebildet und getestet werden. Durch einen automatischen Testfall-Generator können zahlreiche Anforderungsszenarien in kürzester Zeit erzeugt werden. In der vorliegenden Arbeit werden die erzeugten Testfälle als Input-Daten für die Simulation von komplexen verteilten Produktionsstraßen mittels eines Simulation-Tools herangezogen.
Das Ziel der Arbeit ist, eine effiziente Methode zum Testen der erwähnten Testfall-Generatoren im Bereich der Simulation von Produktionsautomatisierungen aufzuzeigen. Als Testmetrik wird das Verhältnis der Testabdeckung zum hierfür notwendigen Aufwand festgelegt. Unter Testabdeckung bei einer gegebenen Parametermenge wird in der Arbeit das Verhältnis zwischen den erzeugten Testfällen und den möglichen Testfällen verstanden. Alle verfügbaren Parameter eines Testfalls sind in einer GUI durch den Anwender auswählbar.
Zur Erzeugung der Testfälle werden zwei unterschiedliche Ansätze betrachtet. Ein Ansatz ist der statisch spezifische Ansatz, welcher den Nachteil aufweist, dass zusätzliche Parameter nur mit erhöhtem Aufwand und mittels Programmierkenntnissen hinzugefügt werden können. Der zweite Ansatz verwendet ein dynamisch generisches Skript, welches auf einer Ontologie als Datenmodell basiert und die Testfälle abhängig vom gewählten Parameter-Setting generiert. Durch die Verwendung einer Ontologie kann diese mittels Werkzeugunterstützung ohne jegliche Programmierkenntnisse erweitert werden. Die lose Kopplung zwischen der Ontologie und dem Generator-Skript ermöglicht, dass Änderungen an der Ontologie keine Änderungen am dynamisch generischen Skript nach sich ziehen.
Das dynamisch generische Skript erzeugt Testfälle entsprechend der gewählten Parameter. Anschließend werden die erzeugten Testfälle in eine XML-Datei exportiert. Zusätzlich kann das Simulationsergebnis als Feedback in der Ontologie gespeichert werden, wodurch Erfahrungswerte nachfolgenden Generierungsprozessens zu Verfügung stehen.
Neben der zur Laufzeit dynamisch generierten grafischen Oberfläche, der Ontologie und dem dynamisch generischen Skript, wurden auch Produktions- und Fehlertoleranzstrategien implementiert bzw. umgesetzt. Diese Strategien sind essentiell, um eine gegebene Produktionsstraße optimieren zu können. Der Evaluierungsteil der Arbeit zeigt auf, dass der dynamisch generische Ansatz mit einer High-Level Testbeschreibung auskommt, eine erhöhte Flexibilität aufweist und eine festlegbare Testabdeckung ermöglicht.

Abstract

Production automation systems are often complex systems as the behavior of the overall system cannot easily be predicted from the behavior of the subsystems. Thus simulation is used to study the behavior of complex production automation systems. In addition to the accuracy of the simulation the system performance is an important issue, particularly if many parameter variants for system behavior are to be tested. Parameters for assembly lines are, for instance, scheduling strategy, failure handling strategy and the number of products.
Software testing investigates the quality of the product or service under test. In order to fully test all requirements of an application, there must be at least one test case for each requirement. The goal is to generate test cases in a fully automated and systematic way to find suitable scenarios for most of the requirements in a short period of time. In this thesis, the test cases define input data to simulate complex distributed assembly line systems by means of a simulation tool. The thesis presents an effective method for testing the performance of test case generator approaches for a production automation simulator. For this purpose, the test coverage combined with the costs to achieve this test coverage is used as performance metric. In our context the test coverage is the ratio between the number of generated test case scenarios and the number of all possible test case scenarios by a given set of parameters. To make the interaction between the user and the system more user-friendly a graphical user interface (GUI) is offered which allows the user to choose the provided test case parameters.
The thesis investigates two different approaches for providing test cases. One approach is the use of a static specific generator script where it is difficult to add new parameters. In addition, the users need programming skills for both setting and modifying parameters. The second approach uses a dynamic generic generator script together with an ontology as data model. Test cases are generated with respect to the chosen parameters by the user. Important advantages of using an ontology are efficient tool support for modifying ontologies and the fact that the generator script is not affected by the modification. Thus users do not need programming skills to add new parameters. The focus of the practical part of the thesis lies on enhancing the existing ontology of the simulation tool to include the test case generator domain. A dynamic generic generator script is worked out to generate test cases from the ontology and export them as XML file. The implemented generator script and the ontology are coupled loosely. Therefore changes to the ontology do not necessarily lead to changes of the dynamic generic script. This fact enables a flexible and high-level test description. Furthermore, the results of executed simulations can be integrated into the ontology as feedback. As a result, an optimal set of parameters can be achieved. In addition to the GUI, the ontology and the dynamic generic generator script, scheduling strategies and failure handling strategies will be implemented. These strategies are essential to optimize a given assembly line reasonably and are therefore one of the most important test case parameters. The evaluation part explores how the ontology-based approach reduces costs for test description, enables a definable test coverage, and increases expandability due to lower efforts to implement new test case parameters.

Contents

List of Figures

List of Tables

List of Abbreviations

ACC	Agent Communication Channel
ACL	Agent Communication Language
API	Application Programming Interface
COTS	Commercial Off-The-Shelf
EER	Extended Entity Relationship
FAS	Flexible Assembly Systems
FIPA	Foundation for Intelligent Physical Agents
HMS	Holonic Manufacturing Systems
JADE	Java Agent Development Framework
JVM	Java Virtual Machine
KBS	Knowledge Based System
KQML	Knowledge Query and Manipulation Language
MAS	Multi Agent System
MAST	Manufacturing Agent Simulation Tool
MDA	Model-Driven Architecture
MVC	Model-View-Controller
OCL	Object Constraint Language (see page 20)
OMG	Object Management Group
OO	Object-Oriented
OWL	Web Ontology Language (see page 36)
RDF	Resource Description Framework
RDFS	Resource Description Framwork - Schema
ROI	Return on Investment
SAW	Simulation of Assembly Workload
SQL	Structured Query Language
UML	Unified Modeling Language (see page 19)
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XMI	XML Metadat Interchange
XML	Extensible Markup Language (see page 40)

Chapter 1

Introduction

The thesis is set within the production automation domain. In this work the term production and manufacturing is used synonymously. A production system performs productive activities to manufacture products with respect to the available resources and the necessary knowledge of the production process. In general, a production process transforms the raw materials into tangible goods by using an assembly line ([23]). In 1983, CIRP (International Conference on Production Research) defined the term *manufacturing* as *"a series of interrelated activities and operations involving the design, materials selection, planning, manufacturing production, quality assurance, management and marketing of the products of the manufacturing industries"* (found in [23]). Thus, manufacturing is a multidimensional process to manufacture products. Production automation allows to execute most of these activities using machines and therefore results in cost-savings. The major technological driving force in the evolution of advanced manufacturing systems is the application of information technologies ([27]). Complex distributed production automation systems can be seen as the results of this technical evolution process. The advantages of these systems are their flexibility and the transparency of data-processing activities. The challenges faced by complex production automation systems are the growing variety of products as result of customization and the short time to market to achieve an advantage in competition. These challenges can be met with the help of production automation simulation tools. The basic idea of simulation is to get appropriate information about the real system in a short period of time. An example for such a Manufacturing Agent Simulation Tool (MAST) in the production automation domain is the SAW project at the Vienna University of Technology ([44], [31]). As a consequence, manufacturers do not have to optimize their assembly lines by experimenting on the real system which would lead to a long setting-up time for new assembly lines.

Two fundamental facts have to be met to obtain significant results from the simulation. Firstly, the model represented by the simulation tool has to be an adequate abstraction of the real world domain. Secondly, the test cases as input data for the simulation have to be validated and many in number to assure a high test coverage. This thesis deals with the second purpose.
The test coverage is the ratio between generated test cases and all possible test cases for a given set of parameters. It is nearly impossible to produce suitable test cases manually. One reason is that the problem for finding all possible test case scenarios is a combinatorial one. Another reason is the effort required to create machine-readable XML files without tool support, especially in case of many test case parameters, a high number of orders and long shift durations. Apart from the high risk to make mistakes without tool support it is a time-consuming process if all the work is performed manually. For this reasons, it is essential to use a generator script which fulfills both tasks in a fully automated and systematic way.

This section identifies criteria which have to be fulfilled by a generator script to make it a good solution. In general, generator scripts generate test cases and write them into an XML file. This XML file is used as input data for the simulation. Beside the already mentioned test coverage a high-level test description is most important for a good solution. A high-level test description reduces the risk of making mistakes during the configuration phase of the parameter setting. The test case generation process is based on this parameter setting. Therefore, it is reasonable to validate the user input data. Furthermore, the test description should be flexible to meet the requirement of expandability, e.g. by introducing new test case parameters. A consistency check ensures that the test cases are consistent and therefore executable. A test case without a production strategy, for instance, is not consistent but valid. The consistency criteria are different to the validation criteria. A test case is valid if everything is typed well. In other words, the validation ensures for each parameter that the user input data is within the allowed value range. Additionally, the consistency check ensures that structural dependencies are met. Thus a consistent test case contains all parameters which are mandatory to simulate the test case. Most important of all is the usability which makes the test case generation process manageable for the different actors of the target audience.

An overview of the solution approach of the dynamic generic script is given in the following section. An ontology is used as underlying data model for the test case generation process. The ontology defines the vocabulary of the domain, the entities within the domain, and the relations between them. Formal language such as OWL and RDF can be used to describe an ontology. This structured data model enables the realization of

a GUI which contains all offered test case parameters of the data model. This circumstance offers many advantages. Firstly, the user can only choose parameters which are supported by the generator script. Secondly, it can be assured that the configuration of the test case generation process is both valid and consistent. The chosen parameter setting is the input data for generating corresponding test cases by the generator script. The process to generate a dynamic GUI corresponding to an ontology is used as a way of model transformation (see appendix C.1). A further topic from the model engineering field is the Model-Driven Architecture (MDA). Parts of the architecture of the dynamic generic script force the model-driven approach to deal with the expandability of new test case parameters. As a consequence, new parameters only need to be added to the data model to make them available in the GUI. The tool support for enhancing an ontology together with the dynamic generic approach make it possible to add new test case parameters without programming skills. In short, the dynamic generic approach performs all mentioned criteria to be a good solution.

As mentioned above, the domain of the thesis is the production automation domain, which is why figure 1.1 resembles - with some imagination - a product tree (compare figure B.2 in the appendix).

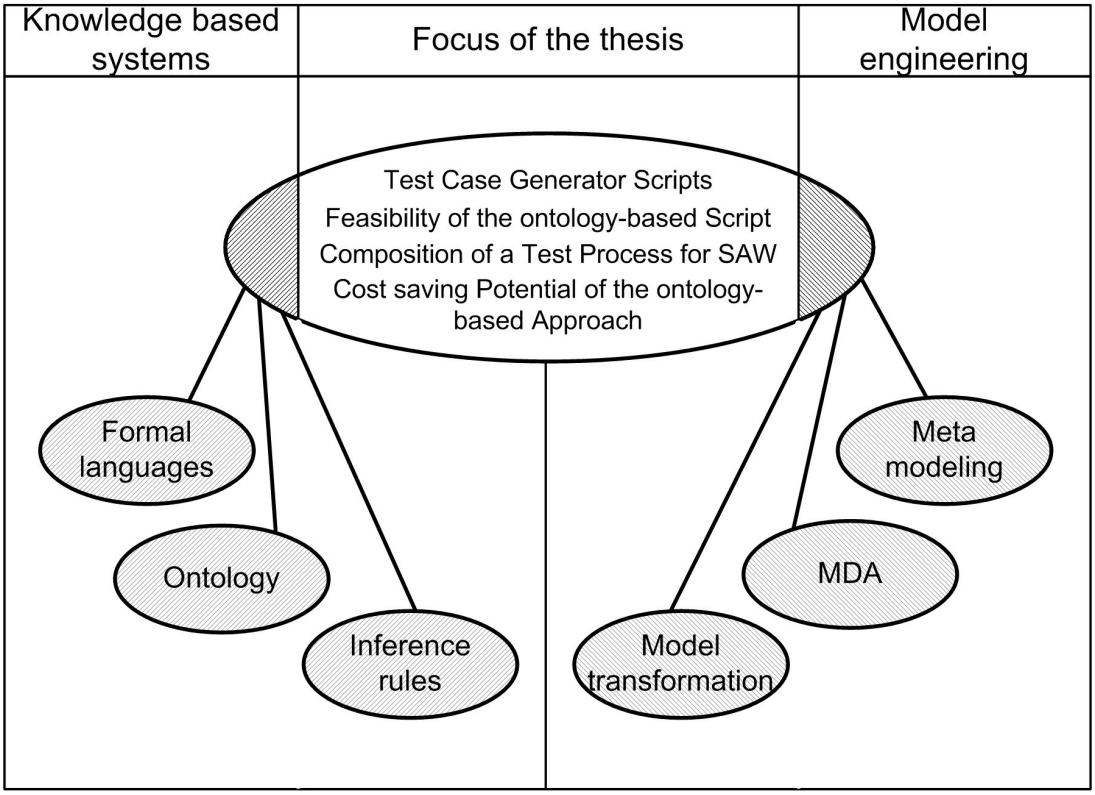


Figure 1.1: Focus of the master thesis

In the following, the limitations of the existing static specific approach will be stated. Firstly, the user needs programming skills to define a parameter setting in the static specific approach. Secondly, the user also needs programming skills to add new test case parameters. Additionally, neither a high-level test description nor a low failure potential can be achieved by means of the static specific approach. These weaknesses of the static specific approach should be met by a dynamic generic approach. Two of the three research issues deal with these circumstances (see chapter 3 on page 45). The goal of the thesis is to offer an easily operable test case generation process for the target audience. The target audience of this work are Testers, persons who are entrusted with the automation process as well as system designers are.

Figure 1.2 gives an overview of the whole process to fulfill the task to generate and simulate a test suite for a given set of parameters. The cyclic process consists of the shown nine steps which are explained in detail in section 4.2 on page 56. The process is an iterative one as the results of one process cycle are only valid for a specific parameter setting. A test suite consists of test cases and should be able to achieve the user-defined test coverage. In our context the test coverage is the ratio between the number of generated test case scenarios and the number of all possible test case scenarios for a given set of parameters. The goal is to present an effective method for testing the performance of test case generator approaches for a production automation simulator. For this purpose, the test coverage combined with the costs to achieve it is used as performance metric. Furthermore, it can be excluded that there is a trade-off between increasing the test coverage and reducing the generating time. Nevertheless, a definable test coverage can be achieved for the ontology-based approach (see table 5.1 on page 92).

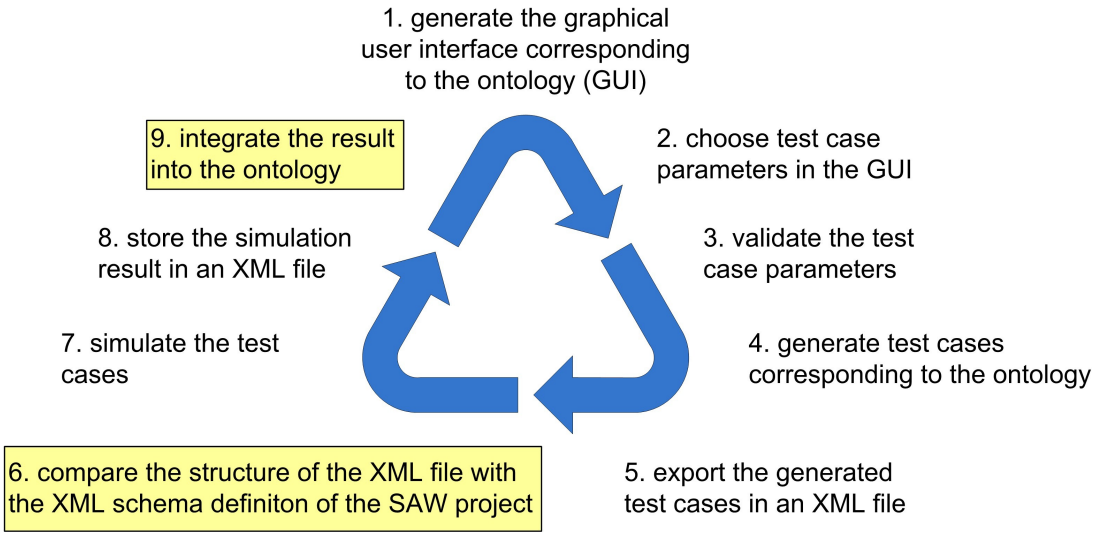


Figure 1.2: Process cycle of the test case generation

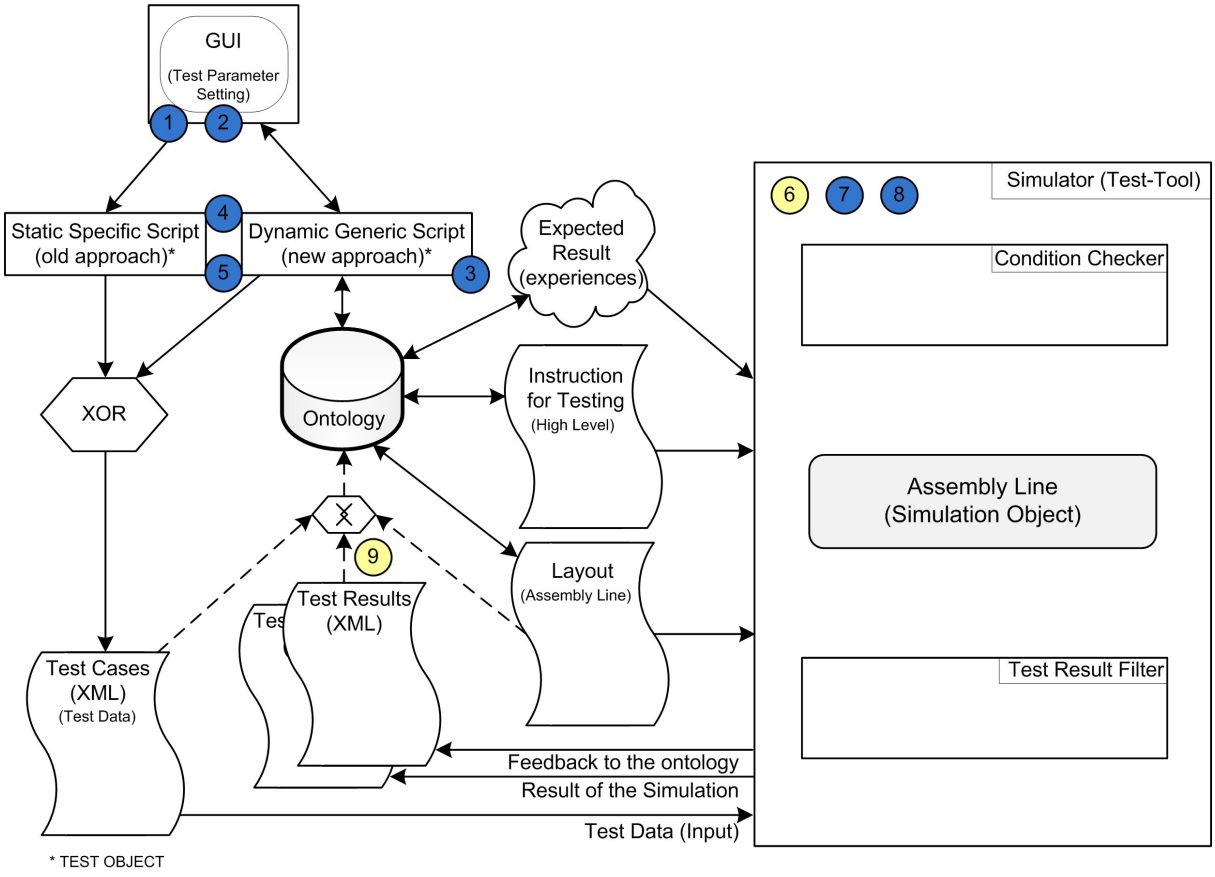


Figure 1.3: Overview of the simulation process (for details about the simulator refer to figure 4.8 on page 62)

Coming back to figure 1.2 mentioned above, all steps shown are necessary to complete a test case generation cycle in a specific simulation environment. Step 9 ensures that the criterion of knowledge increase is met, which means that experience can be taken into account for following test case generation cycles. Step 6 makes sure that the structure of the file which includes the test cases from step 5 are suitable for the assumed structure of the simulation environment. However, the evaluation of the performance of test case generator scripts is not affected by step 6 and step 9 as only one process cycle is taken into consideration at a time. In addition, the whole evaluation took place in the same simulation environment without any structure inconsistencies. Therefore, the two yellow marked steps 6 and 9 are not part of the implemented prototype, but both steps are explained conceptually in the in elaboration chapter. Nevertheless these two set screws should be taken into account to increase the effectiveness and acceptability of the ontology-based approach.

Figure 1.3 shows the simplified simulation process linked with the nine steps of the cyclic

test case generation process already described (see figure 1.2). The color of the differently numbered steps drawn as cycles indicates whether the steps were implemented during the work or not. The color blue indicates that it is part of the elaboration. Step 6 and step 9 are not implemented and are therefore marked yellow. Nevertheless, a feasibility study shows that step 6 and step 9 are realizable (see section 4.3.4 on page 77).

The following simulation scenario sums up the whole simulation process. The user wants to choose from all provided test case parameters to automatically generate input data for the simulation. For this purpose, a GUI is offered. Furthermore, the user can decide which generator script should generate the test cases for the chosen parameter setting. Depending on the selected generator script either the static specific script or the dynamic generic script delivers the XML-structured test cases file as result. The test cases are used as simulation input data together with the layout of the assembly line and the directory where the simulation results should be stored. This information is necessary to start the simulation. After this the simulator executes each test case one by one. Afterwards the results of the simulation can be found in the directory which has been selected. All events of interest are captured in the result file. Which events are of interest can be chosen as parameter at the beginning of the simulation scenario. Finally the most important facts are integrated into the ontology corresponding to the generated test case and the assembly line layout. This experience is used to expect the result drawn as cloud in figure 1.3 as well as the preconditions and postconditions of each test case before the simulation starts. This input data, the instruction for testing, the layout of the assembly line, and the mentioned test data specify a test case for the simulation ([41]). That is to say, the defined test case where the generated test cases are the test input data to simulate the assembly line is **not** the same as the test case of the test case generation process shown in figure 4.9 on page 64. For that reason, the assembly line is called simulation object whereas the test object is the static specific script and the dynamic generic script. In the final analysis the thesis aims to identify cost-saving potential of these generator scripts.

All currently implemented test case parameters are explained in chapter 4.4 on page 82. Most important is that the simulation process (see figure 1.3) and the test case generation process (see figure 4.9 on page 64) as part of the simulation process are coupled loosely. For this reason the test case parameters in the generated test case file do not necessarily have to be implemented in the simulation. As a consequence, the user can generate up-to-date test cases on the basis of the latest version of the ontology even if the version of the simulation is out of date.
Besides the description of all parameters an evaluation of the scheduling strategy and the failure handling strategy is available in section 4.4.1 on page 82 and the following.

The thesis includes, but is not limited to, the discussion of the following research questions:

- Does an ontology address the needs of a Multi Agent System (MAS) in the production automation domain?
- Can the ontology help to increase the test coverage and reduce the costs for testing compared to the hard-coded manner?
- Is it possible to develop a high-level design to add new parameters by using an ontology as data model?
- How much saving potential does the ontology-based approach have compared to the hard-coded manner?

The research questions above are to be understood as a brief outline of the main topics addressed in this thesis. The detailed research issues can be found in chapter 3 on page 45 and the following pages.

The performance evaluation of the static specific approach and the dynamic generic approach pointed out that the dynamic generic approach is more efficient than the static specific approach on most of the objectives. The objectives of the performance evaluation are the test description, the implementation effort, and the test coverage. The test description is used in the context of how to manage the test case generation process. The criteria for a good test description is achieved if the generator approach supports a high-level test description. The metric to compare the effort for modifying existing parameters and especially for adding new test case parameters is a combination of the implementation time and the needed skills to do implementation. For the third objective it is most important to mind the definition of the term test coverage. In the context of the thesis the test coverage is the ratio between generated test cases and all possible test cases for a given set of parameters. The performance evaluation of the test coverage is based on the criteria of a definable test coverage during the configuration phase of the generation process.

The benefits of the dynamic generic approach are represented through the major results of the thesis. The high-level test description of the dynamic generic approach increases the usability and reduces the possibility of making mistakes during the configuration phase of the test case generation process. In case of the dynamic generic approach it is possible to add new test case parameters without any changes to both the test description and the generator script. This result allows to make changes without programming skills by modifying the data model with tool support. A further important benefit of the dynamic generic approach is that the test cases as output of the generation process are both valid

and consistent. As a result, it can be ensured that no inconsistent states which are caused by the test cases occur during the simulation.
Furthermore, it could be shown that the risk of making mistakes during the configuration phase of the test case generation process is significantly higher for the static specific approach.

This work is structured in six chapters. The introduction part gives an overview of the work, the research issues and the method to achieve these aims. Furthermore, the domain and the most important terms are defined. The layered structure makes it possible to decide the depth of getting into the topic individually. A short summary of this work and its results are given in the introduction and conclusion. The research issues specify the challenges and the research methods used to face those challenges. The discussion part includes a detailed explanation of the results of the elaboration part. There, you can also find information about the effectiveness with respect to the performance of the ontology-based approach compared to the hard-coded one. The research issues together with the discussion part which refers to the related work part of the work form the next in-depth level. The core of the thesis is the elaboration part. There, the process to obtain the technical expertise is worked out scientifically. The related work part can be seen as a frame for the whole work because all parts of the thesis refer to the related work part. The conclusion will summarize the results and insights gained and outline some topics which might be interesting in future work.

Chapter 2

Related work

The explanations of the following sections are limited to the issues which the reader should know when reading the elaboration part of the thesis. At first, some elementary terms which are often used during the thesis are defined. Script and approach are probably two of the terms most used in the thesis. In general, a script is an abstract form of a sequence of instructions ([22]). In the thesis the term script or, more precisely, generator script is always related to the instructions which are necessary to perform the test case generation process described in section 4.2.2 on page 63. The static specific and dynamic generic scripts are concrete implementations of two different approaches to generate and provide test cases written in the programming language Java[1]. The terms hard-coded manner and static specific approach are used synonymously during the thesis. In addition, the term ontology-based approach is a synonym for the dynamic generic approach in the thesis.
Firstly, the thesis aims to define the generation process. Secondly, the limitations of the existing static specific script should be met by a dynamic generic script. Finally, the evaluation of these two approaches should inform about the performance with respect to the test description, the implementation and the test coverage.

[1] http://www.sun.com/java/

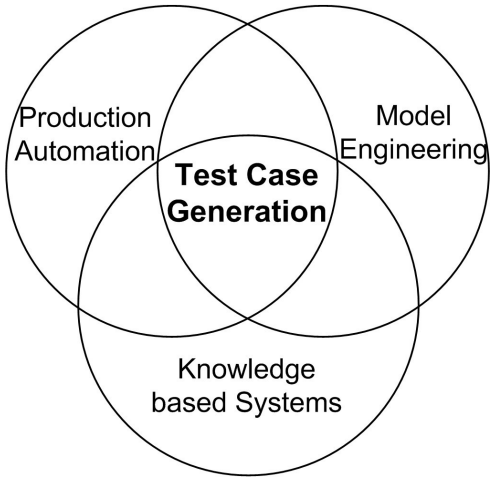


Figure 2.1: Involved fields of the master thesis

Figure 2.1 shows the core field of the thesis written in boldface. On the basis of the surrounding fields the work tries to find an effective method for testing the performance of test case generator approaches. The field "production automation" is the domain of the work and therefore responsible for the work flow of the process. As mentioned in the introduction part, the process is realized using an ontology. An ontology is a "knowledge based system" therefore the work also addresses this field. The field "model engineering" helps to deal with meta models and model transformation. These topics are essential to perform the test process with the dynamic generic approach (see figure 4.9 on page 64).

2.1 Software engineering concepts

This section gives a brief overview of concepts which help by making architectural decisions of a software project in the pre-development phase. The focus lies on software testing which ensures the quality of the software system throughout all phases of the software life cycle.
The term software architecture is often misunderstood with respect to the terms architectural patterns, reference models, and reference architectures. Figure 2.2 shows how these terms are related. The following term definitions refer to [3]:

Architectural Pattern: *"An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used."* [3]
In general, a pattern consists of a set of constraints on an architecture and the element types. The Client-server is a common architectural pattern. The client and the server are two element types which communicate by using a suitable protocol.

Reference Model: *"A reference model is a division of functionality together with data flow between the pieces."* [3]
The reference model uses a standard decomposition to divide a known problem into parts. Afterwards, the problem is solved by solving each parts individually.

Reference Architecture: *"A reference architecture is a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them."* [3]
The reference architecture is the mapping of the provided functionality to software elements by means of the reference model.

A software system describes the structure of the system components and the communication between those components on a relatively high level of granularity ([22]). Examples of system components are data bases, security subsystems or the communication infrastructure.

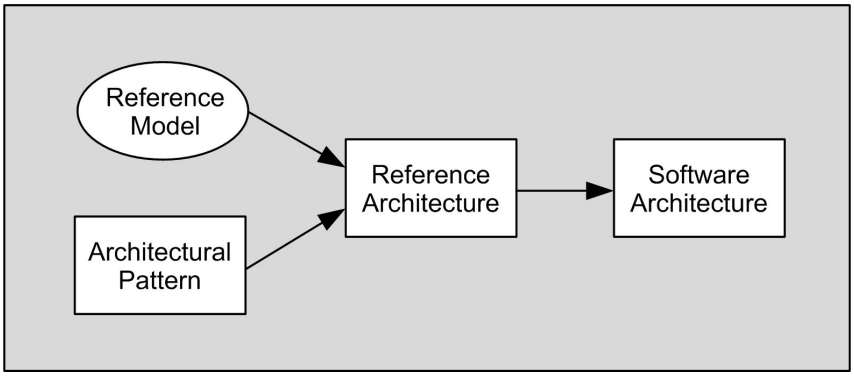


Figure 2.2: The relationships of reference models, architectural patterns, reference architecture, and software architectures (according to [3])

2.1.1 Software testing

The testing of Software is part of every phase of the software life cycle (see figure 2.3). Software testing is getting more and more important and therefore the testing field increases continuously. The National Institute of Standards and Technology stated in a 2002 study that software bugs cost the U.S. economy about 59.5 billion dollar annually ([39]). The same study shows that high costs could be reduced by more than a third by improving testing. The marked process groups and their processes in figure 2.3 show which parts we look at in detail. In the *IEEE Standard for Software Life Cycle Process* a process is defined as a set of activities and the process groups represent a higher level of abstraction ([8]).

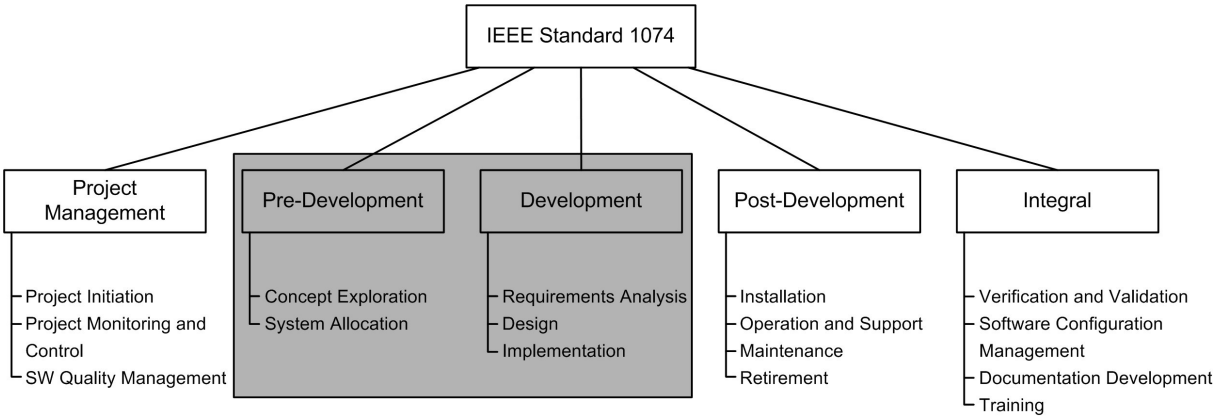


Figure 2.3: Software life cycle referring to the IEEE 1074 standard (according to [8])

A test process can be seen as a systematic execution of a software program. Test management and running the test object with specific data are the important parts of the test process. The goal of the test management is to plan, execute and analyze the test run. A test run consists of one or more test cases. [41]

As mentioned in the introduction part, the test object in our case is the generator script. That is to say, neither the assembly line represented by the XML layout nor the implementation of the simulation is the test object in this work. The main elaboration aims of this work (see chapter 4 on page 49) are finding a way to generate appropriate test cases for the test run and showing the cost-saving potential of the static specific and dynamic generic generator script. The generated test cases are the input data for testing the performance of the two generator scripts by a data-driven approach. Testers, persons who are entrusted with the automation process as well as the system designers are the target audience for proceeding a simulation (see simulation process 1.3 on page 5). The goals of the simulation depend on the parameters chosen by the actors. Mostly, the interests lies on performing a high overall throughput. The test results are written in an output file and can be analyzed for exymple to answer the following questions:

- Which scheduling strategy provides the highest number of finished products in a given period of time?
- How great is the impact of a broken machine during the simulation?
- Which are the bottlenecks of the system?

The questions above are specific ones for the simulation process but the focus of the thesis lies on the subprocess of the simulation process called test case generation process (see figure 4.9 on page 64). This work aims to find answers to the questions laid down in the introduction chapter on page 7. The optimization of this subprocess will lead to an optimization of the overall simulation process. In addition, the usability of the MAST

simulation tool will be increased as a result by means of the dynamic generic script. The reason for this is that for the dynamic generic generator script which is more efficient than the static specific one programming skills are neither required to generate test cases nor adding new test case parameters. The needed skills by the user instead of programming skills and where the complexity of the generation process is moved to will be explained in section 4.2.2 on page 63.

2.1.1.1 Basic principles

Figure 2.4 shows the life cycle of a test case which is subdivided into the creative process and the automatic process. For the creative process, tool support does exit. As a consequence of the tool support, the tester can increase the quality of testing. In the thesis the tasks for planning the test cases are located in the test case generation process (see section 4.2.2 on page 63). The costs for searching a suitable tool, acquisition costs, and maintenance costs build up the cost triangle. The focus of the thesis is to investigate the maintenance costs or rather the expandability costs of the generator scripts.

The execution of the test case can be automated. The SAW project[2] provides a simulator that can execute the test cases for the production automation domain. In addition, the logged events during the simulation allow to analyze the simulation object under test. Thus, the results of the simulation enable us to check the postcondition of the test case. The explained life cycle of a test case is adopted for the simulation process of the SAW project during the elaboration of the thesis (see section 4.2.1 on page 59).

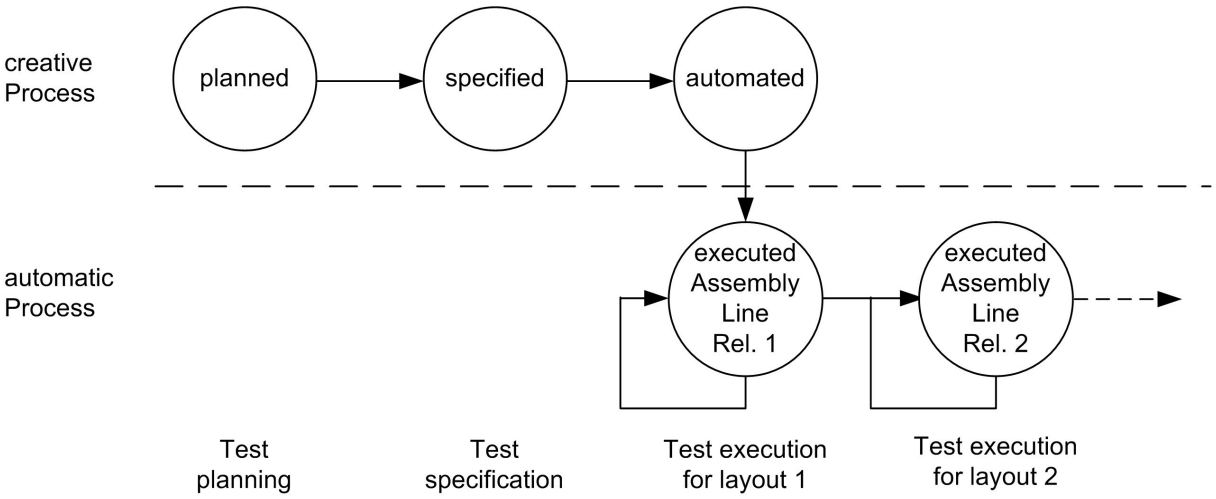


Figure 2.4: Life cycle of a test case (according to [41])

[2]http://www.ifs.tuwien.ac.at/csde/saw

2.1.1.2 Term definitions

This section summarizes a set of term definitions which are essential for testing a system. Most important of all is to distinguish between the terms failure, fault and error. A failure is a deviation of a component or system which may lead to an incorrect result or unsatisfying service with respect to performance requirements. A fault is an incorrect step, process, or data definition in a component or system that could lead to fail to fulfill the system requirements. A fault can cause a failure of the component or system during runtime. An error is an incorrect result caused by a human action. [43]

Failure: 1. Is the condition when the user notices an internal fault. 2. Difference between expected behavior and current behavior.

Failure tolerance: 1. The system is able to keep on working after wrong input data has been entered (robustness). 2. The ability to resume working after an abnormal system behavior (reliability).

Testing: 1. the whole systematic process to provide evidence that the agreed requirements of the system are met by the test object. Testing is also used to detect the effect of failures. 2. each test run on the test object verifies if the test result with respect to the specific constraints is the expected result. 3. stands for all activities and steps in the test process.

Performance: Is a metric to identify how well the offered services of a system works according to constraints like reaction time and throughput.

Performance Testing: Is a process to prove the performance of a system for specific use cases. Mostly the throughput depending on an increased workload is measured.

Costs for Testing: Testing cannot provide evidence that the tested software is entirely free of failures. To test a system with a test coverage of 100 percent would mean that all possible value sets for the input parameters combined with the different constraints are met by the test runs. This is why a completely tested system is not possible in praxis ([41]).

Failure costs: There is a trade-off between costs for testing and costs which result from a failure case. In general, it is much cheaper to find and fix a failure in an early project phase.

Test Automation 1. A software tool generates test cases which can be automatically executed by a computer system. The major advantage is the re-usability of the test cases. 2. Software tools provide support throughout the whole test process

Test Case: The following data is necessary to specify a test case: the test data as set of the input parameters for the test object, the preconditions, instruction for testing,

the expected results as well as the postconditions. Test cases usually also contain a priority.

Test Coverage: Criteria depending on the test method to finish the test.

Test Run: Is the execution of one or more test cases on the test object with a specific version.

Test Method: 1. Process which operates on a set of rules to get test cases. 2. The instruction to execute tests.

Test Metric: Is a measurable property of a test case, test run or test cycle together with the corresponding instruction for measuring.

Test object: Component, part of the system or system which has to be tested.

Test Process: Contains all activities to plan and control, analyze and design, realize and execute, evaluate and report as well as to finish the test activity for a given project (see figure 2.5 on page 18).

Test Strategy: The test strategy defines the methods to achieve the test goals. Furthermore the test strategy defines the costs for testing the test object.

Test Step: A test step is a group of test activities which shall be executed together.

Test Goal: 1. to detect failures is a general goal of testing. 2. to detect the effect of a specific failure by suitable test cases. 3. to provide evidence that the requirements are met by the test object.

Test cycle: 1. process of the whole fundamental test process based on a specific version of the test object. The typical outputs are requests for failure adjustment or change requests.

Failure Classification gives information about how serious the occurred failure is. In other words, to which extent does the failure interfere with the applicability of the system for user purposes. Of course there is a difference between an error in the data base which can affect the stability of the system or a flaw in the GUI layout.

Class 1: crash of the system which may lead to data loss; the test object is not usable.

Class 2: abnormal system behavior as a result of faulty rudimentary features. Requirements are not taken into account or they are implemented wrongly. The test object can be used with many constraints.

Class 3: functional deviation or constraints ("usual" failure). The reason can be insufficient requirements or partly realized requirements. The test object is usable with some constraints.

Class 4: insignificant deviation. The system has no constraints relating to the requirements (rectification of the failure can wait until the next release).

Class 5: flaw which can be a spelling mistake or a mistake in the layout of the GUI for instance. The system can be used without any constraints (the failure should be fixed in the next release).

2.1.1.3 Methods of testing

A test strategy is usually a combination of different test methods. The following test methods can be found amongst others in [9] and [41].

Unit Test: The unit tests have the goal to detect defects with respect to the functionality and structure of the units. Sometimes stubs and mocks are necessary to enable individual unit tests since units are often related to other units. *"Mocks test the behavior and interactions between components, whereas stubs replace heavyweight processes that are not relevant to a particular test with simple implementations"* ([6]).

Integration Test: An integration test has two major goals: 1. to identify defects on the interfaces of units 2. to combine units into working subsystems.

System Test: After the integration test took place all subsystems are put together to the system under test.

Acceptance Test: Developers and clients define in cooperation the acceptance criteria by formulating acceptance scenarios. These acceptance scenarios are the basis for the acceptance test.

Testing after changes: Most parts of the already tested system have to be executed again after a system change took place since the subsystems are related. Therefore it is recommended to run the tests automatically by using a software project management tool like Maven.[3]

Types of Testing: functional test, non-functional test, structural test.

data-driven Test The basic idea of the data-driven tests is to separate the test data and the test script. Usually a spreadsheet captures the different test data records. In the thesis a XML file is used in which all test data sets are strung together to a test suite. Each test data set is represented by a structured test case.

Blackbox Test: The blackbox test is a transparent process (method) which helps to find suitable test cases without knowledge of the internal structure of the system. Two

[3] http://maven.apache.org/

important subcategories are the syntax test and the random test. The syntax test uses information about the formal specification of the test script to specify the test cases. This process checks the syntax of the input data for correctness. The random test randomly generates valid values for input parameters within a given range.

Whitebox: All processes where information about the internal structure of the test object is used to find appropriate test cases.

Figure 2.5 shows the fundamental test process developed by the International Software Testing Qualifications Board (ISTQB). The fundamental test process consists of the following five activities ([25]):

Planning and Control: *"Test planning is the activity of verifying the mission of testing, defining the objectives of testing and the specification of test activities in order to meet the objectives and mission"* ([25]).
The control activity affects all other activities of the test process by comparing the actual progress against the plan. In addition, the test control activity reports the status including the deviations from the plan.

Analysis and Design: The analysis and design activity transforms the general testing objectives to test cases that meet tangible test conditions. After evaluating the testability of the test basis and test objects the test conditions are identified. The major designing activities are designing the test cases and designing the test environment.

Implementation and Execution: *"Test implementation and execution is the activity where test procedures or scripts are specified by combining the test cases in a particular order and including any other information needed for test execution, the environment is set up and the tests are run"* ([25]).
The generator scripts developed during the elaboration of this thesis generate test suites for the test process of the simulation. The test process for the simulation is discussed in section 4.2.1. Detailed information about the generator scripts can be found in section 4.3.

Evaluation and Report: The evaluation activity faces the following major tasks. On the one hand test logs are analyzed to decide if the specified test exit criteria are met. This exit criteria were defined during the test planning activity. On the other hand the exit criteria are changed in case they are not suitable. Finally, the reporting activity provides a test summary report for the stakeholders.

Completion: The completion activities close the test process and collect data from completed test activities. This data is used to analyze lessons learned for future test processes.

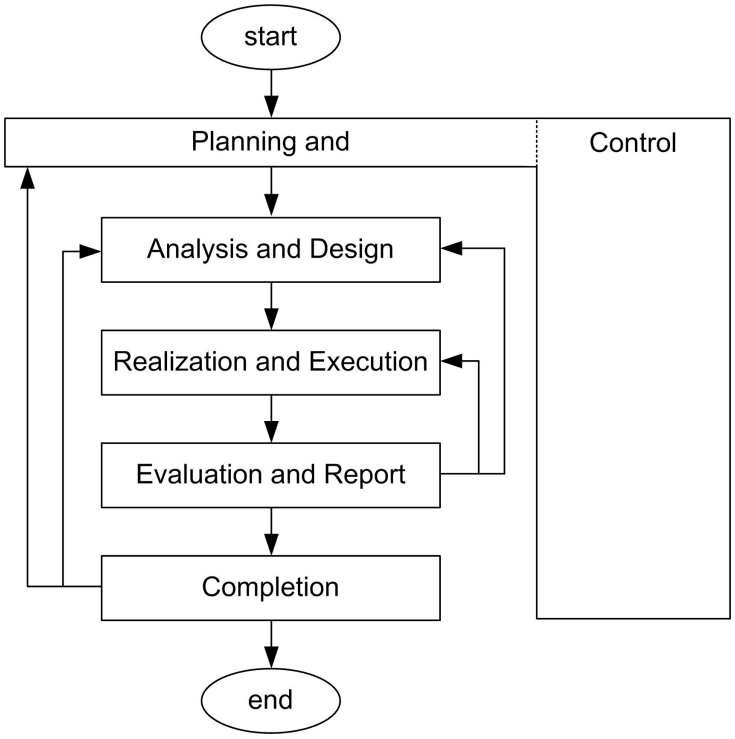


Figure 2.5: Fundamental test process (according to [41])

2.1.2 Model engineering

Modeling is essential for all software projects independent of their size. The structure, also called architecture, is a way of dealing with complexity, which is why modeling is so important for large applications. In the online introduction to UML ([37]) the term modeling is defined as follows:

> *"**Modeling** is the designing of software applications before coding. ... A model plays the analogous role in software development that blueprints and other plans (site maps, elevations, physical models) play in the building of a skyscraper. Using a model, those responsible for a software development project's success can assure themselves that business functionality is complete and correct, end-user needs are met, and program design supports requirements for scalability, robustness, security, extendibility, and other characteristics, before implementation in code renders changes difficult and expensive to make. ... modeling is the only way to visualize your design and check it against requirements before your crew starts to code."* [37]

2.1.2.1 Unified Modeling Language (UML)

The history of UML starts in the nineties when the *method wars* took place. At this time more than 50 development methods arose. Each of them came up with their own modeling language. As a result it was almost impossible for people who were used to different modeling languages to work together. [21]
The problem with the high number of incompatible modeling languages was the motivation for the Object Management Group (OMG) to establish UML based on the winner concepts of the battle zone. According to the simplified list in [21] these winners were:

- Object-Oriented Software Engineering (OOSE) created by Ivar Jacobson
- Object Modeling Technique (OMT) created by James Rumbaugh
- Object-Oriented Design (OOD) created by Grady Booch

The OMG is a non-profit computer industry specifications consortium and UML is OMG's most-used specification[4]. The following statement gives a short overview what the UML specification is defined for:

> *"**The OMG's Unified Modeling Language (UML)** helps you specify, visualize, and document models of software systems, including their structure and design, in a way that meets all of these requirements. ... The process of gathering and analyzing an application's requirements, and incorporating them into a program design, is a complex one and the industry currently supports many methodologies that define formal procedures specifying how to go about it. One characteristic of UML - in fact, the one that enables the widespread industry support that the language enjoys - is that it is methodology-independent. Regardless of the methodology that you use to perform your analysis and design, you can use UML to express the results. And, using XMI (XML Metadata Interchange, another OMG standard), you can transfer your UML model from one tool into a repository, or into another tool for refinement or the next step in your chosen development process. These are the benefits of standardization!"* [37]

In the year 1996 the first version of UML was released as co-production of *Booch*, *Rumbaugh* and *Jacobson*. UML1.1 was released with the Object Constraint Language as feature one year later (OCL, see section 2.1.2.2). UML1.3 includes the XML Metadata Interchange (XMI) specification. XMI makes it possible to exchange object models such as for instance UML models. The latest version is UML2.2 where, among other things, essential restructuring took place (released February 2009). Figure 2.6 shows how the different features and components of UML are related.

[4] http://www.omg.org/

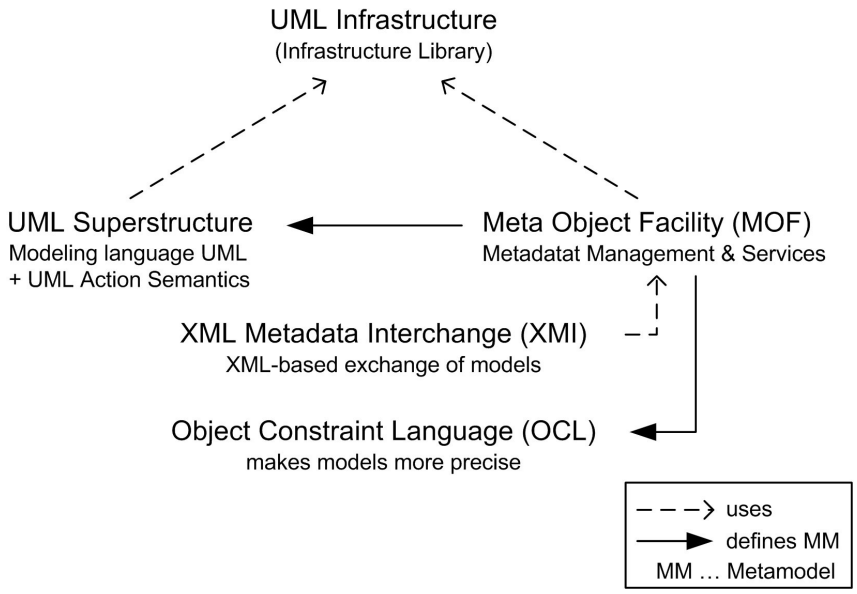


Figure 2.6: Structure of the UML 2.x stack (according to [21])

2.1.2.2 Object Constraint Language (OCL)

The object constraint language is an extension mechanism to UML. OCL was founded by IBM. The formal language allows to describe rules and constraints which has to be fulfilled by the corresponding UML model. However, OCL rules cannot change the underlying UML model. [22]

The OCL constraints can be added to the element in the UML model and enable therefore the idea of *design by contract*. The OCL is part of the UML since the version 1.3 which was released in 1999. [21]

The enhancement of OCL aims to develop a metamodel for OCL which is adjusted to the metamodel of UML. In addition the capability of expression will be increased so that OCL can be used not only for specifying constraints, but also as a general query language for UML models. [24]

2.1.3 Design patterns

The term pattern has its origin during the late 1970s. The architect Christopher Alexander used the term pattern to define best practice concepts in the area of the building industry. The pattern term was adapted in the software field in 1987 ([33]). In the year

1995 *Gamma* et al. published the book "Design Patterns: Elements of Reusable Object-Oriented Software"which presented the first well-described and documented catalog of design patterns ([46]). Later many books and papers, some specific to programming languages were published.

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."* [1]
>
> *"Design patterns are descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context. One person´s pattern can be another person´s building block."* [18]

Design patterns are proven solution approaches that are described in a standardized textual form. A design pattern consists of at least the context for which it is applicable, the problem description, and the solution to the given problem ([22]).
The following design patterns can be categorized into three different types ([18]):

Creational: Encapsulates creational knowledge for an object in a method, a class or another object (e.g. Factory, Singleton).

Structural: Concerned with how classes and objects are composed to form larger structures. Furthermore structural class patterns use inheritance to compose interfaces or implementations. Examples are Composite, Proxy and Facade.

Behavioral: Describes not just patterns of objects or classes but also the patterns of the communication between them (e.g. Observer, State, Strategy).

The following design patterns are a selection of common design patterns of the gang of four (GOF) ([18]). The singleton pattern is used for the implementation of the dynamic generic approach (see section 4.3.1 on page 66).

2.1.3.1 Singleton

This design pattern makes the class itself responsible for keeping track of its sole instance. After the class is instantiated it can ensure that no other instance can be created. The code fragment shows how the class can provide access to this instance.

```
1   public class Singleton {
2         protected Singleton()
3         public static Singleton getInstance()
4         {
5               if (instance == null) {
6                    instance = new Singleton();
7               }
8               return instance;
9         }
10         private static Singleton instance = null;
11   }
```

Code Fragment 2.1.3.1: Singleton Pattern Code

The singleton pattern applies to the situation in which a single instance of a class is required (e.g. handling log objects, factory). The pattern should not be misused doing "OO global variables".

2.1.3.2 Abstract factory

The abstract factory pattern is applicable when a family of related classes can have different implementation details. Under these circumstances the client should not know anything about the used variant.
For instance, a widget factory should offer at least a method for creating a window and a method for creating a scroll bar. However, the implementation should not be affected in case that the look and feel of the GUI might change.

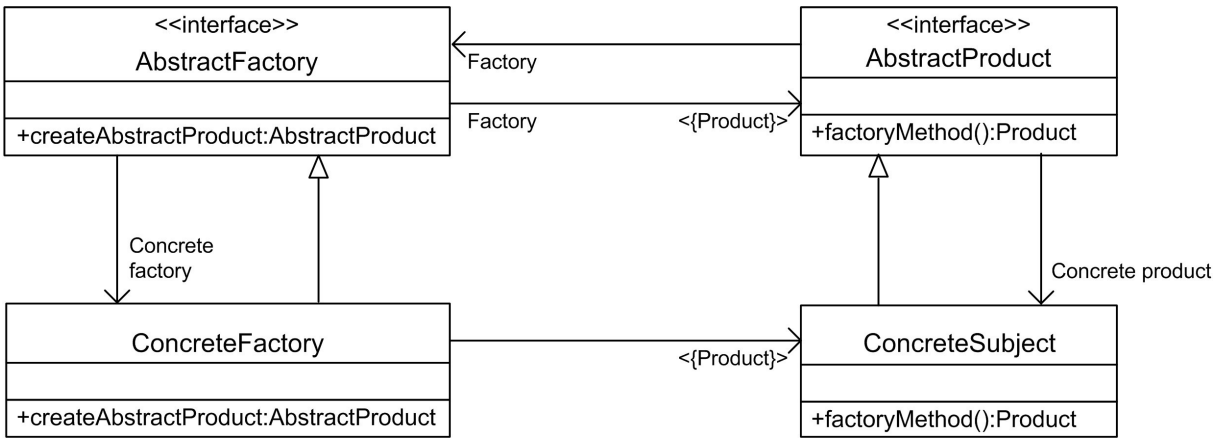


Figure 2.7: Structure of the abstract factory pattern (according to [18])

2.1.3.3 Factory method

The factory method pattern deals with the question how to create an object without knowing its concrete class. The solution to this problem is to provide a method for creation at interface level. As shown in figure 2.8 this can be achieved by deferring the actual creation responsibility to subclasses.

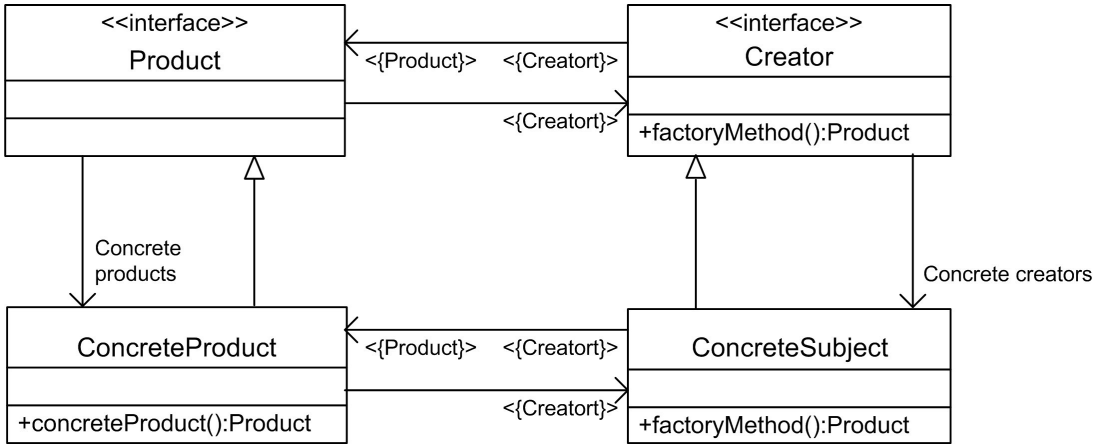


Figure 2.8: Structure of the factory method pattern (according to [18])

2.1.3.4 Composite

The composite pattern faces the challenge to present part-whole hierarchies of objects or manipulating target objects either individually or grouped together. Mostly, there are differences between composition objects and individual objects. The solution is that the composite object implements the same interface as a primitive object. In case the composite object refers to other composite objects the operation will be forwarded (see figure 2.9).

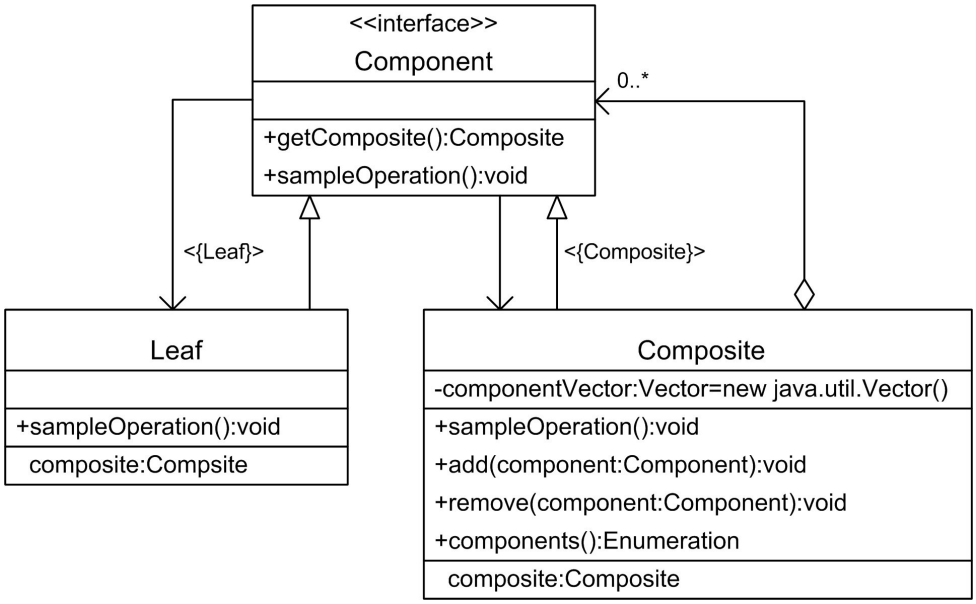


Figure 2.9: Structure of the composite pattern (according to [18])

2.1.3.5 Observer

A change in one object often influences one or more other objects. The number and type of objects to be notified is not always known. The Observer subscribes the observable Subject it is interested in. The observed Subject notifies all its Observers about changes so that they can run their update method (see figure 2.10).

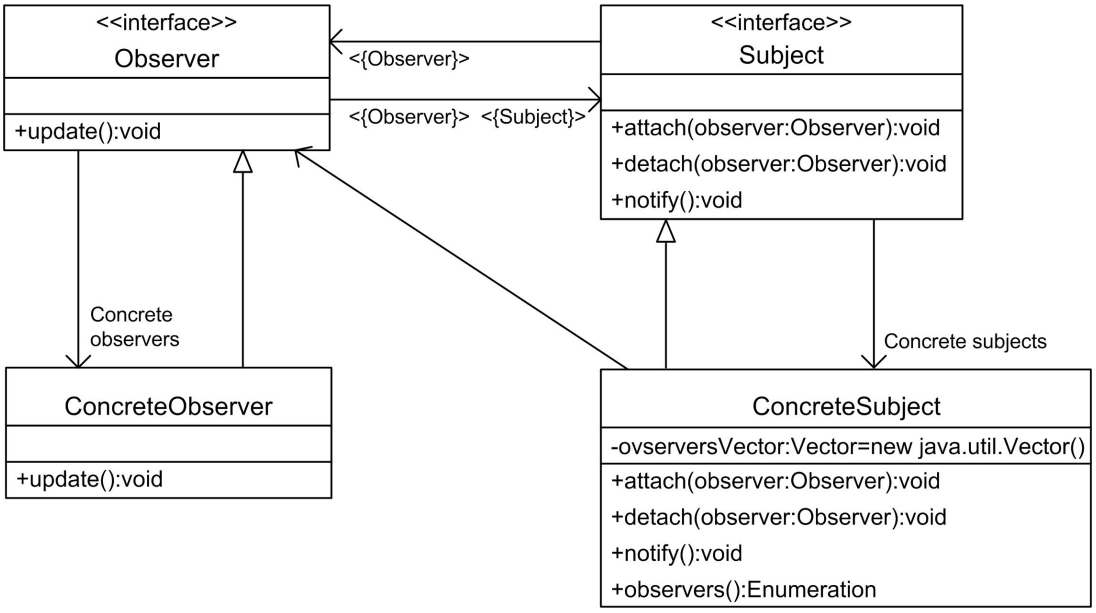


Figure 2.10: Structure of the observer pattern (according to [18])

2.1.3.6 Template method

The Template Method can help to reduce the effort of implementing and maintaining several almost identical pieces of code. Therefore an abstract class offers the core algorithm structured as template. The concrete class can define the variant of the algorithm. Figure 2.11 shows the idea behind this pattern.

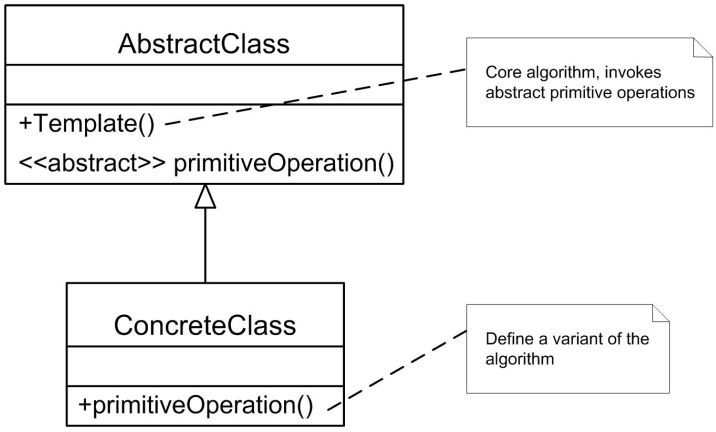


Figure 2.11: Structure of the template method pattern (according to [18])

2.1.4 Model-View-Controller (MVC)

The architectural pattern *Model-View-Controller* (MVC) divides an application with user interaction into three components. The first component is the model which captures the main functionality and data. First, the view component shows the users the information. Second, the view component allows the users to interact with the application. Usually, the interaction with the application takes place via a form. The controller component processes the users' input data and is therefore responsible for controlling the data flow. The view component together with the controller component define the user interface. A special communication mechanism ensures the consistence between the user interface and the model. [17]

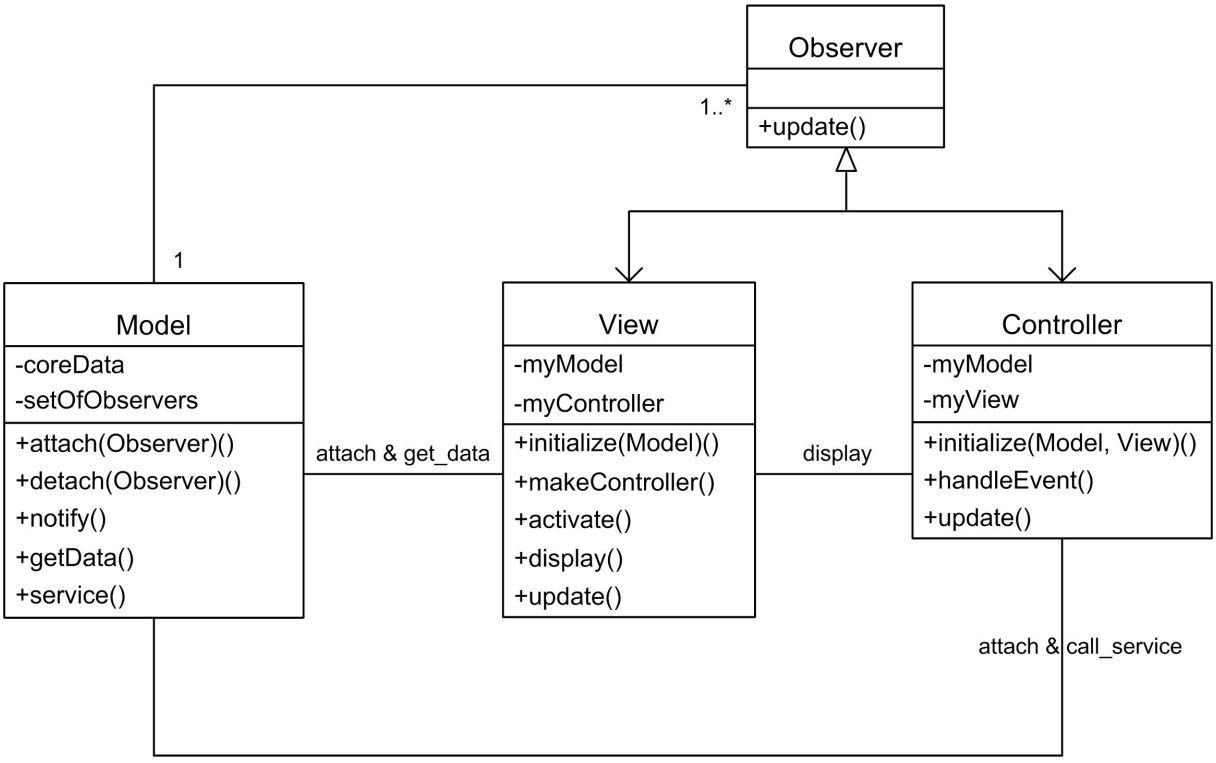


Figure 2.12: Model-View-Controller pattern (according to [17])

Figure 2.12 shows the conceptual design of the model-view-controller pattern. MVC is recommended for interactive applications where the focus lies on a flexible and easy expandability. The basic idea of MVC pattern is that the model and the user interface is coupled loosely, which means that the model encapsulates the data and the functionality of the application. Both the data and the functionality can be accessed via offered methods. The view uses the data management methods to display the data. Thus the MVC pattern enables different views to display the same data in different ways such as text or diagram. The mentioned communication mechanism is based on events. Each time an event occurs the registered controllers are informed about the event. Afterwards, the controllers handle the occurred event. In case data is modified the appropriate notify method is called to inform all affected views. As a result each involved views' update method is called by the controllers.

2.2 Knowledge-based systems

The field of the artificial intelligence originated during a workshop in Dartmouth in the year 1956. Until the late 1960's it was common to develop systems based on a general problem-solving approach by chaining up deduction processes with only little domain

knowledge. This approach was enhanced to achieve better results by using more information about the problem domain. As a result, the field knowledge-based systems appeared. One first systems was DENDRAL which used a high number of rules to define knowledge about a specific domain. DENDRAL is located in the chemistry domain and was developed to interpret mass spectrograms to determine the structure of molecules. Another well-known example for a knowledge-based system is the medical expert system MYCIN. MYCIN diagnoses bacteriological infectious diseases and it makes suggestions if the use of antibiotics is advisable. [4]

2.2.1 Architecture

Most important of all is the separation of the knowledge representation of a given domain which is also known as knowledge base and the knowledge processing in knowledge-based systems. The knowledge base can represent different kinds of knowledge depending on the knowledge characteristics ([4]):

- case-specific knowledge: It is the most specific knowledge as it only refers to a specific case of a problem. This could be facts which are obtained from making observations or doing research.
- rule-based knowledge: It can be seen as core of the knowledge base. The rule-based knowledge can be sub-classified into two classes:
  - domain-specific knowledge: This kind of knowledge is already generic knowledge about the whole domain. It can be both theoretical expert knowledge and know-how.
  - general knowledge: Examples of general knowledge are general problem solving heuristics or rules for optimization as well as general knowledge of objects and their relations in the real world.

The granularity of the different kinds of knowledge depends on the domain of the knowledge-based system and what it is build for.

Figure 2.13 shows the general structure of an expert system.

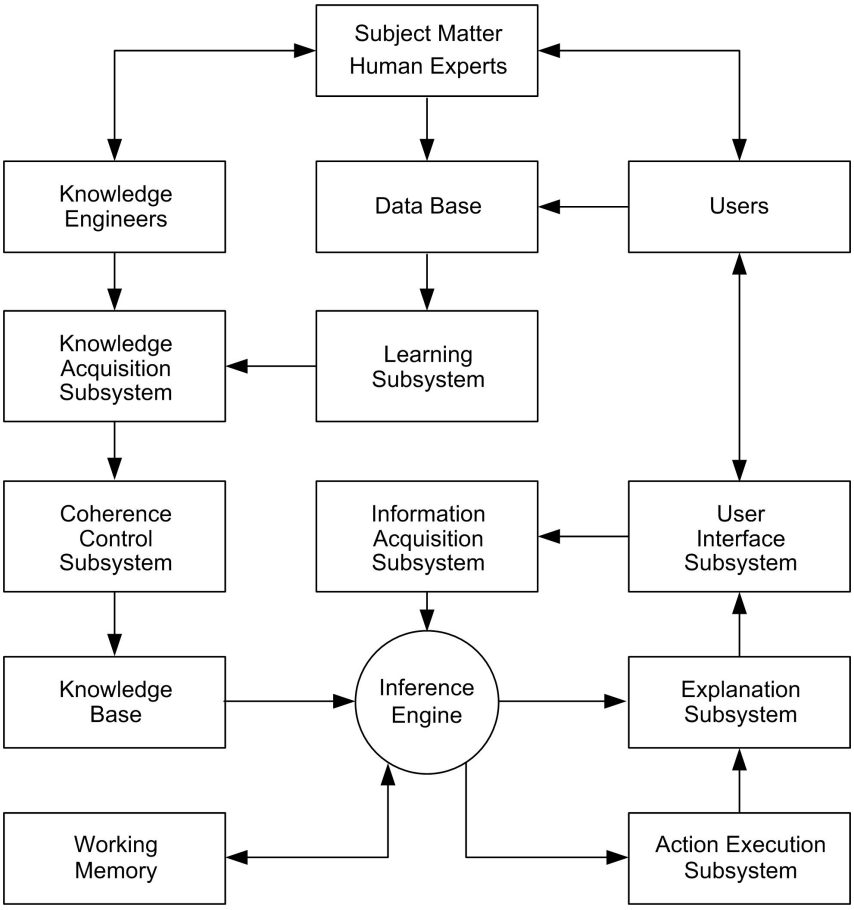


Figure 2.13: Typical components of an expert system. The arrows represent the flow of information (according to [12])

Human Component: Usually, an expert system is created with the help of different participants. First, the human experts provide the knowledge base as different experiences. Second, the knowledge engineers translate this knowledge into a language which can be processed by the expert system. This time-consuming process is most important to meet the users' requirements.

Knowledge Base: The subject-matter specialists ensure that the knowledge base is structured and consists of well-defined and well-explained relationships only.
"Knowledge can be either abstract or concrete. Abstract knowledge refers to statements of general validity such as rules, probability distributions, etc. Concrete knowledge refers to information related to a particular application. For example, in medical diagnosis, the symptoms and diseases and relationships among them form the abstract knowledge, whereas particular symptoms of a given patient form the concrete knowledge. [..] The abstract knowledge is stored in the knowledge base, and the concrete knowledge is stored in the working memory." [12]

Knowledge Acquisition Subsystem: Each time the human experts want to add new

knowledge to the knowledge base the knowledge acquisition subsystem controls if the knowledge is necessary or new at all.

Coherence Control: This component is not realized in all expert systems even though it is essential to ensure the consistency of the knowledge base. Inconsistent statements can affect the performance of the whole system. Furthermore, inconsistent statements can result in offered probability values larger than one or smaller than zero by the system.

Inference Engine: *"The inference engine is the heart of every expert system. The main purpose of this component is to draw conclusions by applying the abstract knowledge to the concrete knowledge. For example, in medical diagnosis the symptoms of a given patient (concrete knowledge) are analyzed in the light of the symptoms of all diseases (abstract knowledge).*
The conclusions drawn by the inference engine can be based on either deterministic knowledge or probabilistic knowledge." [12]

Information Acquisition Subsystem: In case that the initial knowledge is too little for the inference engine to finish the inference process further knowledge has to be obtained. One possibility is to use the information acquisition component to provide the required information. Another possibility is that the user provides the required information via a user interface. Therefore the user information has to be checked before the inference engine operates the provided information.

User Interface: The user interface component monitors the information of the conclusions drawn by the inference engine, the reasons for such conclusions, and an explanation for the actions taken by the expert system.

Action Execution Subsystem: With this component the expert system can take actions based on the conclusions drawn by the inference engine.

Explanation Subsystem: The explanation subsystem is an essential component of the expert system that makes the process flow transparent to the user. This is important for users to understand how conclusions were inferred and why the actions based on these conclusions took place.

Learning Subsystem: *"One of the main features of an expert system is the ability to learn. We shall differentiate between structural and parametric learning. By structural learning we refer to some aspects related to the structure of knowledge (rules, probability distributions, etc.). [..] By parametric learning we refer to estimating the parameters needed to construct the knowledge base."* [12]

2.2.2 Design

Developing a knowledge-based system can be seen as a complex software engineering challenge due to some particularities ([4]). The developing process of an expert system can be described in eight steps ([12]). The following eight steps are adapted to describe the developing process of knowledge base systems ([4]).

1. Statement of the problem: The first step in any project is usually the definition of the problem to be solved. This step is most important for finding adequate requirements which have to be met by the system. If the requirements are wrongly defined it is impossible to meet the features expected by the user.
2. Finding knowledge sources: Usually, knowledge bases of systems are related to at least one specific domain. Data bases, books as well as human experts might provide the necessary knowledge.
3. Design of the knowledge system: In this step, the designing of the structure for knowledge storage, the inference engine, the explanation subsystem, and the user interface take place.
4. Choosing a development tool: If a satisfying tool still exists it is recommended to use it to save money and to assure quality. However, in some cases specially designed systems such as a shell, a tool, or a programming language may be necessary.
5. Building a prototype: In the early phase of the development process an executable prototype is essential to check wheter the system fulfills the requirements.
6. Testing the prototype: This step defines and executes the test process to find out if the prototype meets the requirements. In case that the prototype is not suitable the previous steps have to be repeated.
7. Refinement and generalization: Faults detected during the test process can be fixed in this step. Furthermore, established enhancements to the initial design are also located in this step.
8. Maintenance and updating: Appearing bugs, users' complaints, and common modifications during the software evolution process are part of this step.

The explained eight steps are not isolated ones. There are some loops similar to the ones in software engineering models such as, for instance, the spiral model.

2.3 Ontology

The term "ontology"came up in the eighteenth century in the field of general science of being. "Onto"is an ancient Greek word which signifies being. The discipline ontology arose from the philosophy which was previously a subfield of metaphysics. *Ruiz* and *Hilera* write in the book [10] on page 50 that *"Etymologists may define ontology as the knowledge of beings, that is, all that relates to being. Just as we call those who study "students", we use the term "entity"to describe all things which "are"."*
This sound illustration can also be adapted to abstract or mental beings which are also entities. In the last decade of the twentieth century ontology became of interest for the computational sciences and technologies. Some reasons for the growth in research and application are identified in the following subsections.

2.3.1 Principles

2.3.1.1 Term definition

> *"An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary."* [35]

> *"**An ontology** is formal, explicit specification of a shared conceptualization. Conceptualization refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use, are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private of some individual, but accepted by a group."* [10]

2.3.1.2 Link to related fields

In [35] *Neches et al.* present a vision about building knowledge-based systems by assembling reusable components. The basic idea behind this vision is to build bigger systems by knowledge sharing cheaply assuming that the next mechanism of information exchange are knowledge bases. Existing systems use slightly different names and formalisms.
An ontology for lumped element models that defines these concepts with consistent, shareable terminology is under construction. A library of such shared ontologies would facilitate building systems by reducing the effort invested in reconciliation and reinvention. [35]
The internal interaction is the interaction between knowledge bases. However, the external interaction of applications built on various knowledge-based systems is defined as

the interaction between knowledge-based data bases and other knowledge-based systems. The external modules need a language for encoding the communication such as SQL for conventional data base interactions. The pendant to SQL for knowledge-based applications is called KQML in the paper. KQML stands for Knowledge Query and Manipulation Language. With this language it will be possible to specify wrappers that define messages communicated between modules. Thus, it is required that all systems provide appropriate interfaces.
The shared libraries in figure 2.14 contain several ontologies covering both structural knowledge and problem-solving knowledge. The system engineers develop application-specific systems by assembling components from a library into a customized shell. *The components include a framework for local system software in which one or more local knowledge bases are tied to a shared ontology. [..] With libraries of reusable knowledge-based software components, building an application could become much more of a configuration task and, correspondingly, less of a programming activity* ([35]). Therefore, the library must provide enough information about the offered entities and their constraints enabling the application developers to work with it.

2.3.1.3 Demarcation from related fields

In this subsection the boundaries between the ontology field and related fields are defined. Mostly this is done by describing what an ontology is not compared to the neighboring fields. The following in [10] three concepts are identified as the most confusing ones.

Ontology vs. Conceptual Model: Both concepts provide meta information that describes the semantics of the terms or data. This is the only property shared by these two concepts. The languages for defining and representing ontologies (see section 2.3.3 on page 35) are more powerful than commonly used languages for data bases such as SQL.
The captured knowledge of the ontology is less structured compared to the data in the data base (tables, classes of objects, etc.).
On the one hand an ontology has to be well designed to meet the claim of shared and consensual conceptualization (see next section 2.3.1.4). This effort is necessary because ontologies are usually used for information sharing and the exchange of information. On the other hand the data base schema mostly has to be suitable for a concrete system only.
A further difference is that an ontology provides a domain theory and not the structure of a data container.

Ontology vs. Metamodel: *Ruiz* mentions in [10] that the confusion between ontologies and metamodels is motivated by the fact that both are frequently represented by the

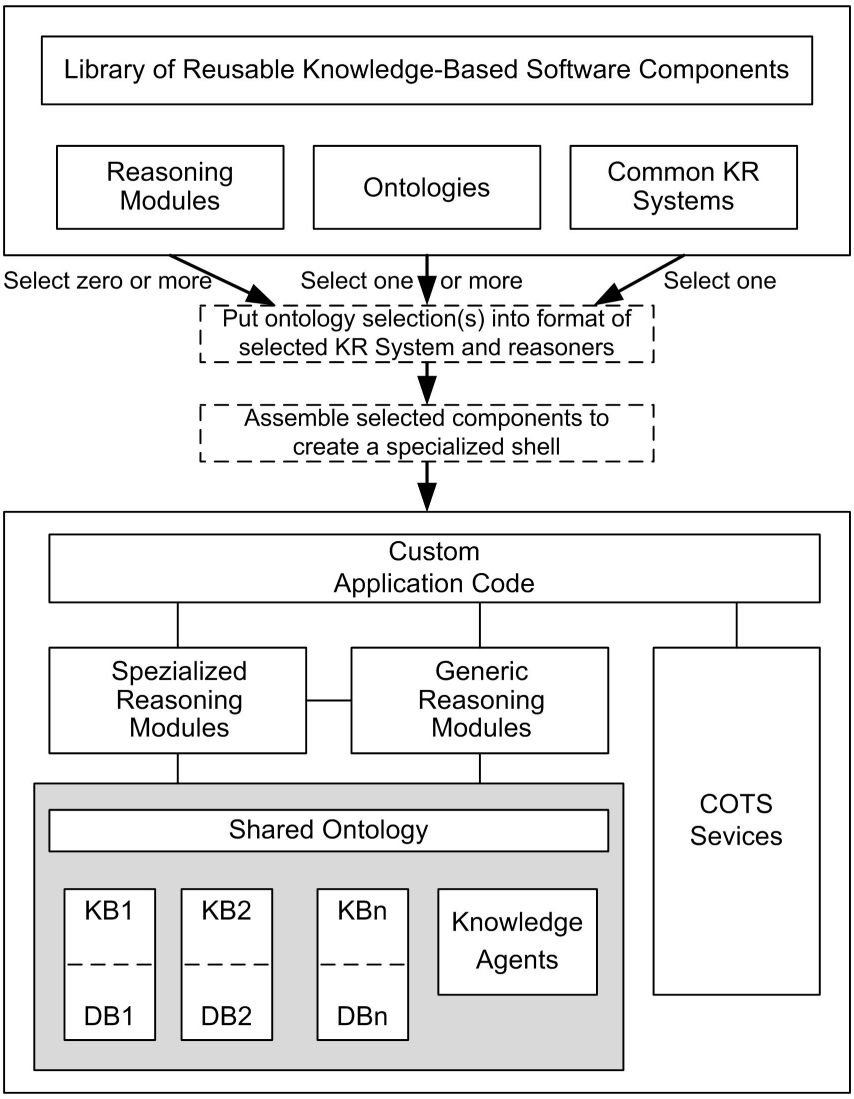


Figure 2.14: Envisioned phases in defining a knowledge-based system (simplified view, according to [35])

same languages. However, the characteristics and scope of application are different. *Bertrand* and *Bezivin* pointed out the relationship between low-level ontologies and metamodels. Their conclusion that metamodels improve the rigor of similar but different models. An ontology does the same but for knowledge models.

Ontology vs. UML: Both an Ontology and UML are data modeling languages. UML defines several types of diagrams that can be used to model the static and the dynamic behavior of a system (see section 2.1.2.1). *Cranefield* and *Purvis* have used an UML class diagram to model an ontology as a static model ([16]). Operations could be modeled as conjunction together with OCL postcondition constraints that specify the result of the operation. Which means that you can define an ontology with UML. UML, however, is no ontology.

2.3.1.4 Design criteria

To represent something in an ontology, design decisions have to be made. For guiding and evaluating designs objective criteria are necessary. *Gruber* proposes a set of design criteria for ontologies ([20]):

1. **Clarity:** *"An ontology should effectively communicate the intended meaning of defined terms. Definitions should be objective."* [20] Mostly, a concept needs to be designed to address social situations or computational requirements. Nevertheless, the definition should be independent of social or computational context. *"It should base on logical axioms. Where possible, a complete definition (a predicate defined by only necessary or sufficient conditions). All definitions should be documented with natural language."* [20]

2. **Coherence:** *"An ontology should be coherent: that is, it should sanction inferences that are consistent with the definitions. Coherence should also apply to the concepts that are defined informally, such as those described in natural language documentation and examples. If a sentence that can be interred from the axioms contradicts a definition or example given informally, then the ontology is incoherent."* [20]

3. **Extendibility:** *"An ontology should be designed to anticipate the uses of the shared vocabulary. [..], one should be able to define new terms for special uses based on the existing vocabulary, in a way that does not require the revision of the existing definitions."* [20]

4. **Minimal encoding bias:** *"The conceptualization should be specified at the knowledge level without depending on a particular symbol-level encoding."* [20]

5. **Minimal ontological commitment:** *"An ontology should make as few claims as*

possible about the world being modeled, allowing the parties committed to the ontology freedom to specialize and instantiate the ontology as needed." [20]

2.3.2 Types of ontology

The diverse classifications of ontologies depend on the different focuses. *Guarino* classifies the ontologies according to the generality level into the following types (found in [10]):

High-level ontologies: Ontologies of this classification are domain independent and describe general concepts such as space, time, material, and object. The aim is to find agreements about criteria on a high generality level.

Domain ontologies: Domain ontologies describe the vocabulary of a domain as specialization of the high-level ontologies concepts.

Task ontologies: Task ontologies describe the vocabulary of a task or an activity as specialization of the high-level ontologies concepts.

Application ontologies: Application ontologies describe concepts related to both a domain and a task. The application ontologies are on a low generality level as specialization of the concepts of the domain ontologies and task ontologies.

2.3.3 Formal languages

A formal language is a well defined artificial language by an formal grammar. This formal grammar specifies the syntax of the language and is therefore responsible for the structure of the artifacts which are written in the formal language. [22]

There are a number of standard languages to describe ontologies. A common formal language for describing ontologies is the web ontology language (OWL) designed by the World Wide Web Consortium (W3C). OWL is based on RDF and is used for applications that need to process the content of the information instead of just presenting information. OWL provides a greater interpretability of web content than XML and RDF. [30]

2.3.3.1 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is recommended by W3C to model metadata about the resources of the web.

"RDF is an application of XML that imposes needed structural constraints to provide unambiguous methods of expressing semantics. RDF additionally provides a means for publishing both human-readable and machine-processable vocabularies designed to encourage the reuse and extension of metadata semantics among disparate information communities. The structural constraints RDF imposes to support the consistent encoding and exchange of standardized metadata provides for the interchangeability of separate packages of metadata defined by different resource description communities." [34]

2.3.3.2 Web Ontology Language (OWL)

OWL consists of three sublanguages ([30]):

OWL Lite: The OWL Lite language is defined for building up a classification hierarchy easily since it has a lower formal complexity than OWL DL. The features are limited compared to the OWL DL and OWL Full. For instance, only cardinality values of 0 and 1 are supported in OWL Lite.

OWL DL: The OWL DL language provides the users the maximum expressiveness retaining computational completeness and decidability. On the one hand the computational completeness means that all conclusions are guaranteed to be computable. On the other hand the decidability means that all computations will finish in finite time. The DL indicates the correspondence with description logics. The OWL DL contains all language constructs of OWL but with several restrictions.

OWL Full: The OWL Full language provides syntactic freedom of RDF beside the maximum expressiveness. However, no computational guarantees can be ensured.

The OWL Full language is more expressive than the OWL DL language, and the OWL DL language again is more expressive than the OWL Lite language.

2.4 Technology and framework description

2.4.1 Protégé

Protégé is an open source ontology editor and knowledge-base framework. Protégé was developed by the Stanford Center for Biomedical Informatics Research at the Stanford University School of Medicine. With the framework it is possible to export the Protégé ontologies into a variety of formats namely RDF(S), OWL, and XML schema.[5] The

[5] http://protege.stanford.edu/

Protégé platform provides two ways of modeling ontologies:

2.4.2 Jena

Jena is an open source java framework for building semantic web applications. Jena evolved during the work with the HP Labs Semantic Web Research. Jena provides APIs for RDF, RDFS, OWL and SPARQL. In addition, a rule-based inference engine is offered by the framework.[6] Jena has the advantage that it is documented well and that there are many helpful examples.

Figure 2.15 gives an overview of the ontology import mechanism of Jena. A separate graph structure holds each imported ontology document. This fact is most important, otherwise it would be impossible to trace where a statement came from. Each arc in an RDF model is called a statement. Each statement asserts a fact about a resource. A statement is a triple consisting of a subject, predicate, and object. The subject is the resource from which the arc leaves. The predicate is the property that labels the arc and the object is the resource or literal the arc points at.

> *"The general Model allows access to the statements in a collection of RDF data. OntModel extends this by adding support for the kinds of objects expected to be in an ontology: classes (in a class hierarchy), properties (in a property hierarchy) and individuals. The properties defined in the ontology language map to accessor methods."* [11]

[6] http://jena.sourceforge.net/

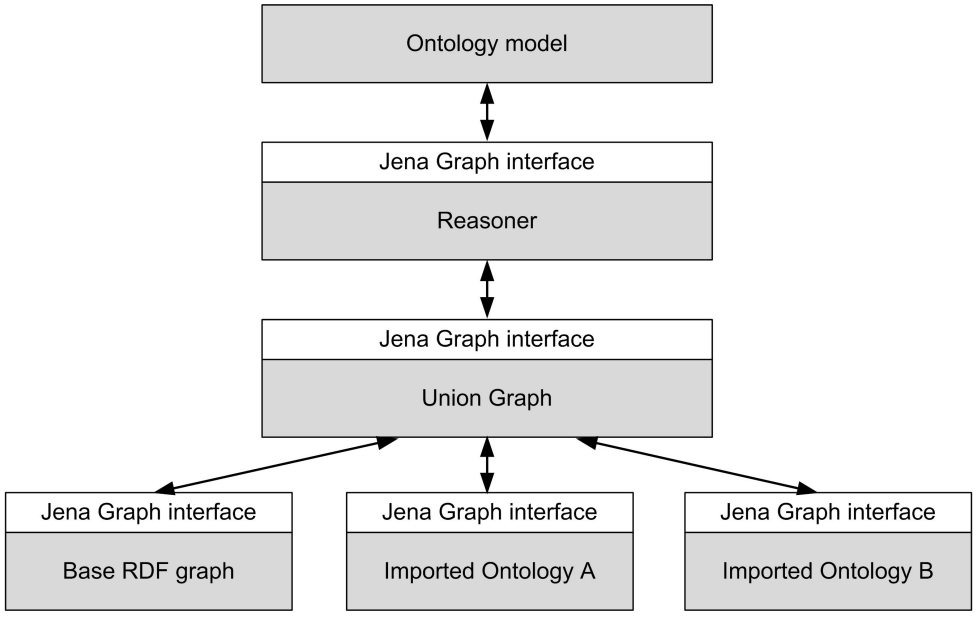


Figure 2.15: Ontology internal structure including imports (according to [11])

2.4.3 Jade

The Java Agent Development Framework (JADE) is a message-based middleware which simplifies the communication in multi-agent systems. The framework is fully implemented in Java language and it complies with the FIPA specifications. It is developed by Telecom Italia Lab (TILAB) under the terms of the Lesser General Public License Version (LGPL). The agent platform provides a Graphical User Interface (GUI) for the remote management, monitoring and controlling of the status of agents. In addition a number of graphical tools are available which support the debugging phase.
The agent platform can be distributed on several hosts with one Java Virtual Machine (JVM) on each host. In general each JVM is a container of agents. The communication architecture of JADE makes it possible to create, manage and queue ACL messages. Since version 2.3 JADE supports user-defined content languages and ontologies that can be implemented, registered with agents, and automatically used by the framework. Furthermore JADE supports format conversion between content exchange formats like XML and RDF ([5]). The current version of Jade is 3.7.[7]

[7] http://jade.tilab.com/

2.4.4 Spring rich client

The Spring Rich Client Project (RCP) is a subproject of the Spring framework. The framework leverage the Spring framework and offers a rich library of UI factories and support classes for building highly-configurable rich-client applications which meet the common GUI standards.[8] In this thesis the Spring-RCP was used to build a dynamic GUI for choosing parameters depending on the underlying data model at runtime. Details about the dynamic GUI are given in the elaboration chapter 4.2.2 on page 63.
The project website states that the project team started a new project some years ago. They maintain the Spring RCP. However, they have no time for documentation. Instead of a documentation they have set up an online forum where everyone can post their needs. The circumstance that almost no documentation exists makes it difficult to use the framework. Nevertheless, the comments in the code of the sample applications together with some helpful blogs make it possible to work with the framework. All the beans of the application has to be listed in the `richclient-application-context.xml` file. The command text in the XML file list typical changes which are necessary to develop an application:

- The startingPageId on the lifecycleAdvisor.
- The eventExceptionHandler on the lifecycleAdvisor (is recommended to use)
- Specify the location of your resource bundle in the messageSource.
- Specify the mapping properties files for images in imageResourcesFactory.
- Specify your rulesSource class, if you are using one.
- Configure your view beans.

The application bean is mandatory since it defines the singleton application instance to be used. The `lifecycleAdvisor` bean arranges the flow of the application. Two key properties, the location of the file containing the command definitions for application windows and the bean id of the page that should be displayed must be configured. The command definitions can be found in the `commands-context.XML` file. The `messageSource` bean specifies the component that is responsible for providing messages to the platform.

Figure 2.16 shows how the application windows, pages and views are related. The application runs in a VM and it contains one or more application windows. Each application window contains exactly one application page which is defined by one or more dockable views. The page manages the view by notifying on view life cycle events like creation, focus gained, focus lost, and destruction. It is possible to open the same view on different

[8] http://spring-rich-c.sourceforge.net/1.0.0/index.html

pages in different windows, but only one view instance per view descriptor is allowed per page.[9]
The implemented test case generation process application consists of one application window, one page and two views (see also section 4.3 on page 66).

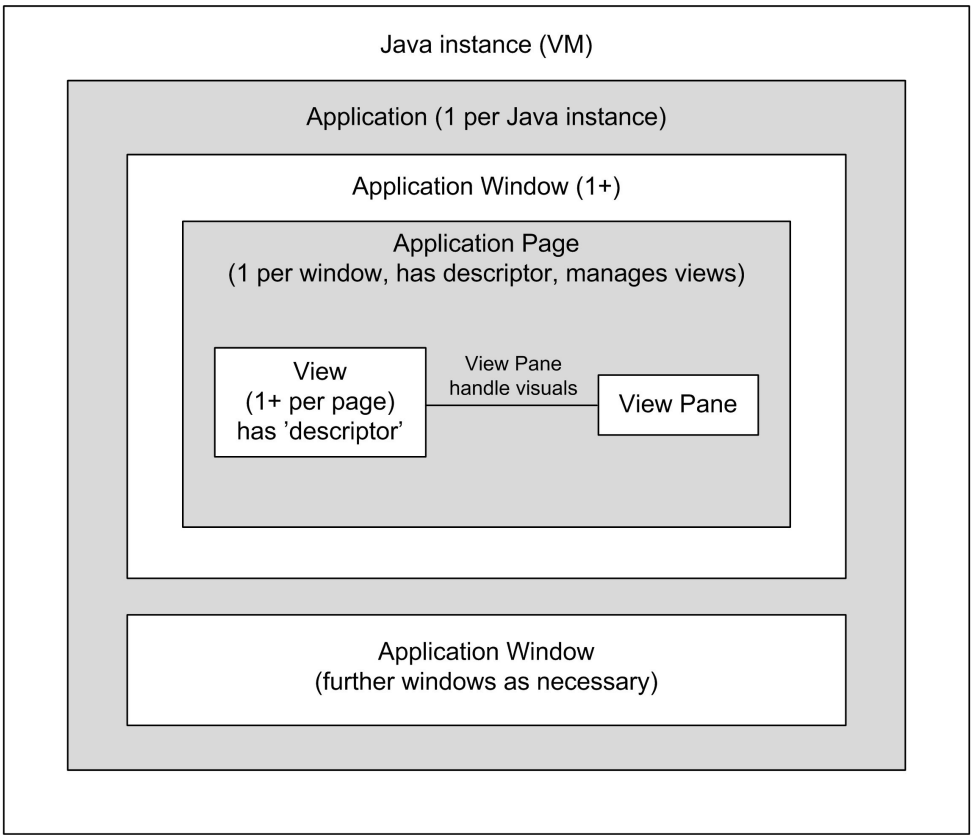


Figure 2.16: Spring Rich Client Platform overview (according to [38])

2.4.5 XML

The Extensible Markup Language (XML) is a data format which can be found in many applications. The reasons for the popularity of XML are characteristics like flexibility, interchangeability and universality. XML is a structured and text-based format for exchanging data and is therefore supported in most programming languages. This section gives a short overview of the structure and possibilities of the standardized Markup Language according to *St. Laurent* and *Fitzgerald* ([42]).

XML is a simplified version of the Generalized Markup Language (SGML) (ISO 8879:1986(E)). The Wide Web Consortium (W3C) makes the key specifications and the name-space and

[9] http://opensource.atlassian.com/confluence/spring/display/RCP/Introduction

XML schema definition for XML 1.0 and XML 1.1[10]. It is most important that the XML document is well-structured to make the XML document processable. Furthermore, XML documents can have but not necessarily must have a related schema. Such Document Type Declarations (DTD) have the major advantage of validating the structure of the XML document.

2.4.5.1 XML structure

The following example gives an overview of the basic structure of XML.

```
1  <?xml version="1.0"encoding="UTF-8"standalone="no"?>
2  <?xml-stylesheet href="mine.css"type="text/css"?>
3  <!--This is a simple document.-->
4  <!DOCTYPE message SYSTEM "myMessage.dtd">
5  <message xmlns="http://simonstl.com/ns/beispiele/nachricht">
6        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7        xsi:schemaLocation="message.xsd"
8        xml:lang="en"datum="2005-10-06">
9    This is a message!
10  </message>
```

Code Fragment 2.4.5.1: Simple XML document

The first line of the code fragment 2.4.5.1 above is optional and gives information about the used XML version and the used character encoding. The standalone attribute informs if an external reference exists. The second line is a processing statement for a locally stored style-sheet which was written in CSS. The third line is a comment line. The next line contains a reference to a Document Type Declaration (DTD) where validation rules are defined. In our example the root element is called message (see line 5). In the lines 5 to 8 different attributes are defined. In the fifth and sixth line the attributes `xmlns` and `xsi:schmeaLocation` declare the name-spaces of the XML document. The attribute `xsi:schemaLocation` in line 7 associate the XML document with an XML-schema-document for the purpose of validation. The `xml:lang` attribute specifies the language and the date format is specified to be ISO 8601 by the `date`-attribute. Line 9 shows the content of the message element. The last line shows the end tag of the XML

[10]http://www.w3.org/

document.

Usually, the XML document consists of text written in the Unicode standard. The text is separated by special characters, mostly <and >, but also &, ", and '.

The most important components of a XML structure are listed in the following.

Element: Elements are the building blocks of the XML document. The elements are surrounded by start tags and end tags. The content is located between the start tags and end tags. A special case is the tag for an empty element.

Attribute: Attributes are tuples with a name and a value. They can be located in start tags or in an empty element.

Text: In general, text consists of a sequence of characters. The XML document consists of markup data and characters which are a type of text.

Comment: Comments represent human readable XML information which does not affect the XML structure.

XML declaration The XML declaration informs about the used version and the used encoding standard. In addition, a reference to external declarations can be made.

2.4.5.2 XML schema

The XML schema is also called XML Schema Definition (XSD) or sometimes W3C XML Schema (WXS). It is an XML vocabulary to describe other XML vocabulary. This makes it possible to validate the structure of the XML document for software programs using the defined rules of the XML schema. With the simple schema definition in the code fragment 2.4.5.2 it will be easier to understand the schema definition of the test cases in chapter 4.3.3 on page 75.

```
1  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema >
2    <xs:element name="autoren">
3      <xs:complexType>
4        <xs:sequence>
5          <xs:element name="person"maxOccurs="unbounded">
6            <xs:complexType>
7              <xs:sequence minOccurs="0">
8                <xs:element name="name"type="xs:string" />
9                <xs:element name="citizenship"type="xs:string" />
10              </xs:sequence>
11              <xs:attribute name="id"type="xs:string"use="required"/>
12            </xs:complexType>
13          </xs:element>
14        </xs:sequence>
15      </xs:complexType>
16    </xs:element>
12  </xs:schema>
```

Code Fragment 2.4.5.2: "Russian doll" XML schema

The following list shows useful structure elements in XML schema with a short description. The list is not complete but should clearly and fully indicate the essential elements for the focus of the thesis. For further reading please refer to the links in the foot note[11] [12] [13].

xs:schema: The `xs:schema` is the container element in which all other schema components has to be located. The different attributes of the `xs:schema` define the name-spaces which can be referenced within the schema.

xs:element: The `xs:element` is used to define elements. In case that the `xs:element` is a child of the root element `xs:schema` then the `xs:element` is a global declaration. This means that it can be used in another declaration. One of the attributes is the `minOccurs` that allows to define an element as optional by setting the value to 0.

xs:annotation: With the `xs:annotation` element it is easy to enhance XML schema declarations. The `xs:annotation` element can be located as first child element to every XML schema element.

[11] http://www.w3.org/TR/xmlschema-0/
[12] http://www.w3.org/TR/xmlschema-1/
[13] http://www.w3.org/TR/xmlschema-2/

xs:documentation: This element is used within the `xs:annotation` element to offer human readable information. Furthermore, the used language in the document can be defined by the `xs:lang` attribute.

xs:simpleType: The `xs:simpleType` can be used within `xs:schema` element or the `xs:redefine` element by using the name-attributes. In addition, the `xs:simpleType` can be used in `xs:attribute`, `xs:element`, `xs:list`, `xs:union` or `union`.

xs:complexType: The `xs:complexType` is a key component for most of the schema. It allows to define types which can be restricted or expanded. It is also possible to define simple elements with attributes.

xs:sequence: The `xs:sequence` sets the order of the contained definitions. Outside of the `xs:group` element the `xs:sequence` allows to define with the `maxOccurs` and `minOccurs` how often it can occur.

xs:attribute: This element can only be used in the `xs:complexType` element or in the `xs:attributeGroup` element. The `xs:attribute` can either define an attribute by using the `name` attribute or refer to an attribute by using the `xs:ref` attribute.

xs:restriction: This element allows to create a new restricted type based on a given type.

xs:minExclusive: With the `xs:minExclusive` element a lower bound can be defined for a given type.

xs:maxExclusive: With the `xs:maxExclusive` element an upper bound can be defined for a given type.

xs:enumeration: The `xs:enumeration` element allows to define a list of valid values for a type.

xs:simpleContent: This element is used within the `xs:complexType` element to define a complex type with a simple content. A simple content consist of a text content and a attribute.

xs:extension: The `xs:extension` can enhance the type of the base. The `xs:extension` can be used in the `xs:simpleContent` and the `xs:complexContent`.

Chapter 3

Research issues and research method

This chapter introduces both the general structuring method of this thesis and the specific research issues covered in this thesis. Furthermore the research methods for each research issue are defined. These methods are essential to get transparent and arguable results. The discussion of the results can be found in chapter 5 on page 88. Detailed information about the different processes to achieve the results are given in the next chapter in which the elaboration of the thesis takes place.

3.1 Feasibility of the dynamic generic approach

The dynamic generic approach aims to face the limitations of the existing static specific approach. Firstly, the new approach should provide a high-level test description to allow the target audience to generate test cases with less effort. Secondly, a validation check and consistency check of the parameter setting is essential to reduce the risk of making mistakes during the configuration phase of the test case generation process. The SAW project is continually being improved by students. Therefore it is most important to meet the requirement of expandability for adding new test case parameters like the production strategy and the failure handling strategy which were implemented during the elaboration of this work. The task for adding test case parameters to the generator script should be possible with tool support.
The feasibility of the dynamic generic approach is proven by implementing a prototype (see section 4.3.1). The implemented prototype meets all mentioned criteria for a good solution. How efficient the criteria are met compared with the static specific approach is part of the research issue 3.2.

3.2 Identification of cost-saving potential for the ontology-based approach

The elaboration part discusses that an ontology is a suitable data model as it is possible to provide the necessary data to generate a suite of test cases with the dynamic generic approach for a given set of parameters. This claim is also met by the static specific approach. In other words, the fact that the dynamic generic approach is newer and works as well as the old static specific one does is not enough motivation for change. However, measurable benefits are essential to accept the new dynamic generic approach and the change costs. The following two subsections define a metric how the cost-saving potential can be measured with respect to their interests. In section 3.2.1 the focus lies on the restriction that both the number of choseable parameters and the number of supported data types are constant during the measurements. In contrary the main focus of section 3.2.2 lies on the costs for adding new parameters with different data types. However, the questions of interest are the same in both sections:

- How much are the costs for the test description?
- How high is the effort to implement the parameters?
- Which test coverage can be achieved?

Two different comparison methods can be found in the literature ([40]). First, you can compare by objects. Second, you can structure the comparison on the basis of the different criteria. In this thesis the objects are the old static specific approach and the new dynamic generic approach to generate test cases as input data for performance testing of assembly lines. The criteria are costs for the test description, the effort to implement parameters, and the test coverage. In our case the criteria can also be called cost units as umbrella term. In summary the comparison takes place between the objects based on the measuring results of the criteria. Let's come back to the open question which comparison method should be used. For the following two research issues the comparison by criteria seem to be more suitable. The main reason for this decision was that the reader might be interested in one specific cost-saving potential criterion. In that case the reader just has to read the section on that specific criterion to get an overview of both objects. Otherwise he would have to go through most of both elaboration parts concerning their object to get the information. In the discussion chapter on page 88 a comparison between the new ontology-based approach and the old static specific approach takes place. Figure 5.1 on page 91 shows th results in a clear and concise manner. Figure 5.1 shows the results as a cube with the technology (object), the environment (research issue 3.2.1/3.2.2), and the cost unit (criteria) as dimensions.

3.2.1 With respect to a constant number of parameters

The three different cost units and the metric for measuring each cost unit are listed below. A constant amount of parameters can be assumed.

Test Description: The test description is a step-by-step instruction for the user to execute the test case generation process. The metric to categorize the test description of the generator scripts is the level of abstraction. Thus, a high-level test description is better than a low-level test description where it is necessary to configure the test case generation process in the source code of the script. A high-level test description increases the acceptability not only for the generation process, but also for the whole simulation process since the simulation is based on the generated test cases as input data.

Implementation: The measurement of the effort for setting up the generator approaches by implementing the corresponding generator scripts are partly based on an estimation. For this purpose the man-months for implementing the static specific script are estimated based on domain experts' experience. In addition, the effort to realize the dynamic generic approach was part of the thesis and is therefore traceable.

Test Coverage: In our context the test coverage is the ratio between generated test cases and all possible test cases for a given set of parameters. For that purpose the test coverage combined with the costs to achieve this test coverage is used as performance metric. A way to determine the necessary information to calculate the test coverage is presented in section 4.3.4 on page 77. As a result, a measurement concept to measure and calculate the performance metric is given.
For the evaluation it is tested whether the generator approach meets the requirement of the definable test coverage by the user during the parameter setting. The parameter setting configures the test case generator script shown in figure 4.4.

3.2.2 With respect to expandability

This section focuses on the expandability of the generator scripts under test. It partly overlaps with the section above describing the constant number of parameters. This is why the term definitions and some redundant information are not repeated in this section. The three different cost units and the metric for measuring each cost unit are listed below.

Test Description: The changes which are necessary for the step-by-step instruction to execute the generation process after a new test case parameter has been added are estimated to categorize the generator scripts.

Implementation: The effort for adding a new test case parameter to the existing implementation of the generator approaches are identified. For this purpose the skills required for the modification as well as the risk of making a mistake during the modification are taken into account to make the two approaches comparable.

Test Coverage: Please refer to section 3.2.1 to get the information about the definition and the metric to calculate the test coverage.

3.3 Composition of a test process for SAW

For the last few decades, testing became an increasingly important task in the software engineering field (see also the chapter on related work on page 11). In general the goal of testing is to detect failures and prove the requirements which should be met by the test object. But how does a sound testing process look like? It seems to be straightforward to use a guideline to ensure that the structure of the testing process contains all important components. With the basic ideas of the fundamental testing process guideline by ISTQB in mind a suitable test process for the domain of performance testing is developed in the elaboration part of this thesis on page 56. In addition, the test process is linked with the discussed life-cycle of a test case in [41] (see also section 2.1.1.1 on page 13).

Chapter 4

Elaboration

This chapter is divided into several sections. The first section gives an overview of the existing Manufacturing Agent Simulation Tool (MAST) developed at Rockwell Automation Research Center in Prague ([44]). The architecture and the way how to process a simulation in MAST is also part of this section. The Simulation of Assembly Workload (SAW) project[1] is based on MAST. In addition, the different existing ontology layers of the SAW project are described in the first section. Also, the test case layer ontology on which the dynamic generic script is based is defined in this section. In the second section an overview of the simulation process as well as the test case generation process is given. In addition, a test process to improve the quality of the overall simulation process is defined with respect to the standardized test process defined by ISTQB[2]. The third section explains in detail how the generator scripts were implemented during the elaboration of the thesis. Beside the structure of the generated test case file the feasibility study is also part of the third section. The feasibility study describes how to face the marked steps of the cyclic test case generation process which are beyond the focus of the thesis. The description of the actually supported test case parameters by SAW can be found in the fourth section of this chapter. The last section introduces the metric to measure the performance of the generator scripts under test. Furthermore, the section proves if the proposed metric is applicable for performance testing of generator scripts. The evaluation of the generator scripts as well as the results of the evaluation are part of chapter 5. The next chapter also discusses the evaluation results to allow the comparison of the scripts.

[1] http://www.ifs.tuwien.ac.at/csde/saw
[2] http://www.istqb.org/

4.1 Simulation system overview

4.1.1 Manufacturing agent simulation tool

The Manufacturing Agent Simulation Tool (MAST) is an agent-based solution for some typical manufacturing tasks by using multi-agent technologies in the holonic manufacturing control ([44]). Flexible distributed manufacturing systems can be modeled as intelligent, autonomous and cooperative elements called holons ([29]). Holons are essential components of a holonic system and each holon manages the local behavior to meet goals locally without centralized control ([44]). In addition, the holons interact to achieve the system goal cooperatively. Holonic Manufacturing Systems (HMS) are the result of research about applying the agent technology to manufacturing areas ([29]).

HMS can be represented as a Multi Agent System (MAS) which is defined in [7] as follows:

> *"Multi Agent Systems are concerned with coordinating intelligent behavior among a collection of autonomous intelligent agents, how they coordinate their knowledge, goals, skills, and plans joinly to take action or solve problems."*

The Foundation for Intelligent Physical Agents (FIPA) is an IEEE Computer Society standards organization that promotes agent-based technology[3]. MAST meets the FIPA standards for the inter-agent communication by using an agent development tool also called agent platform. In general a agent platform allows to create agents with application-specific attributes and behaviors. Furthermore, the messaging between the agents is provided by the agent platform. At the beginning of the MAST project the FIPA-OS platform was used until performance and memory consumptions were reached. Since then the JADE platform is in use. [44]

MAST addresses three different manufacturing tasks:

- the transportation using conveyor belts
- transportation based on Automated Guided Vehicles (AGV)
- assembly line systems

[3] http://www.fipa.org/

4.1.2 Simulation of assembly workshop

The Simulation of Assembly Workshop (SAW) investigates processes, methods, and tool support for planning, coordination, simulation, and lab tests for work shifts in an assembly workshop. Generally, a workshop aims to effectively and efficiently carry out the work orders in a work shift. SAW helps to understand the impact of tactical decisions in the production automation environment. For instance, the user can optimize his assembly line by analyzing the effect of the different strategies (see section 4.4). Further information about SAW can be found on the project's homepage.[4]

Figure 4.1 shows a screenshot of the simulation. The different components as parts of the simulation are described below.

Workpiece: In SAW a workpiece is a transported item on a pallet. The pallet is an extension to the implementation of the AGV in MAST.

Docking Station: Docking stations, also called production machines, offer functions for manufacturing an output product on the basis of at least two or more input products. Input products can be row materials or intermediate products. Output products can be intermediate products or finished products. The IN-sensor and the OUT-sensor read the ID of the workpiece which is shipped through the assembly line. Afterwards the appropriate agent is informed.

Conveyor Belt: Conveyor belts transport the pallets through the system.

Crossing: Crossings are responsible for the routing of the pallets.

[4] http://www.ifs.tuwien.ac.at/csde/saw

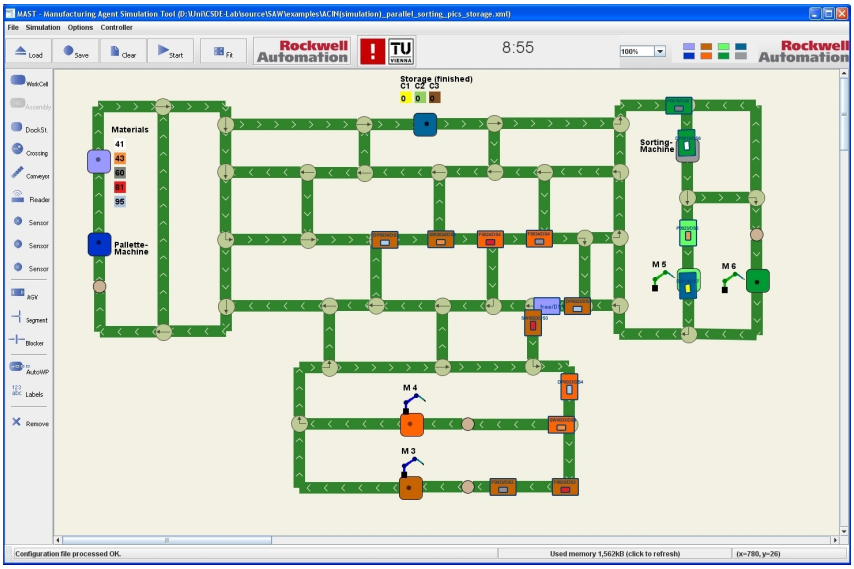


Figure 4.1: Screenshot of the SAW simulator

4.1.3 Advancement by using an Ontology

The five layers of the simulation system and their relations among each other are shown in figure 4.2. At the beginning of this section the five layers get described since they are responsible for the production control (see also the EER diagram of the ontology in appendix C.3). Afterwards the new test case layer gets introduced and explained in detail.

business layer: The business layer prioritizes all incoming orders with respect to the due date.

shift layer: In the shift layer a capacity check of the resources needed for producing the ordered product takes place. In addition, the business orders get transformed to work orders.

job shop layer: In the job shop layer the work orders get broken down to single tasks by the production strategies. Afterwards, the responsible production strategy orders the tasks with respect to their specific criteria (see also section 4.4.1).

operation layer: The load balancer as part of the operation layer is responsible for a well balanced utilization of the docking stations. Furthermore, the shortest path consisting of one or more conveyor belts is calculated to fulfill the different tasks.

master data layer: The master data layer contains all information which are rather constant during the simulation such as the arrangements of the conveyor belts, the

arrangements of the docking stations, and the structure of the product trees. All other layers can reference these information.

The test case layer together with a suitable test case generator script provide high quality test cases in an automated and systematic way. As a consequence a high test coverage can be achieved which is essential for testing the performance of simulation systems. Thus, the user can decide if he wants to create test cases manually or with the help of a generator script. In case of using a generator script the user just has to configure the parameter setting which contains the test case parameters and the corresponding set of valid values. Afterwards, the generator script generates the test cases with respect to the chosen parameter setting.

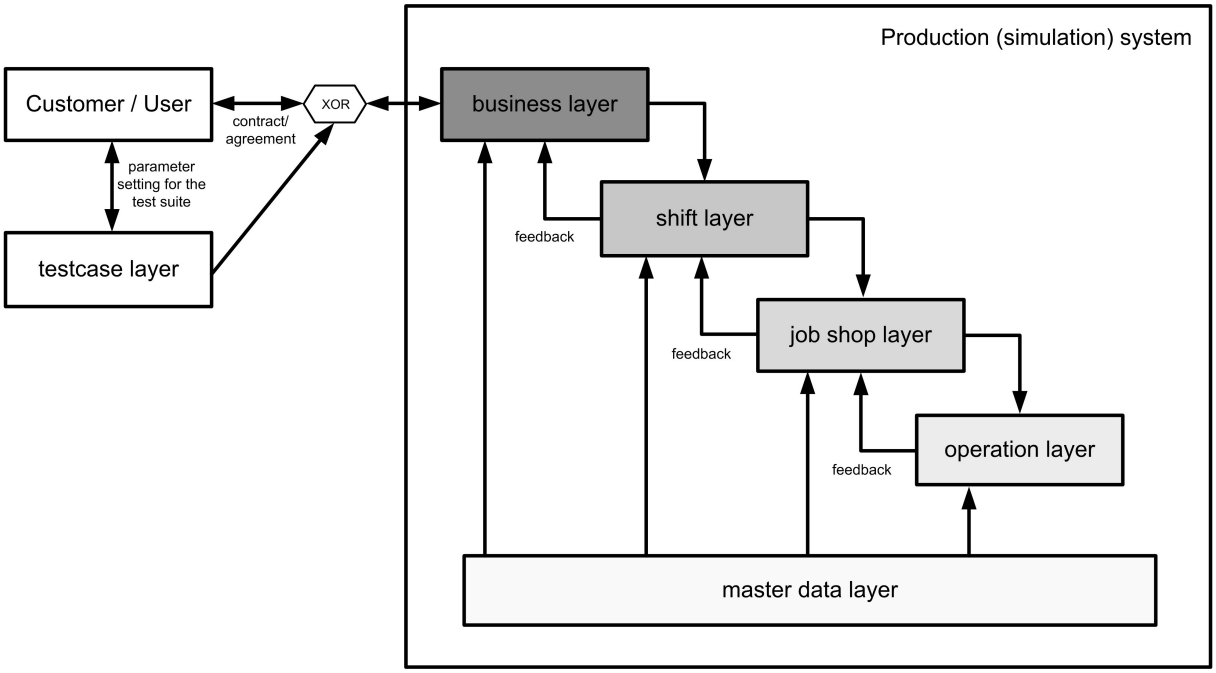


Figure 4.2: Correlation between the different layers of the production system

The EER diagram in figure 4.3 contains all offered test case parameters by the ontology. Each of these test case parameters is described in section 4.4.

The Protégé editor is used to create and modify the test case layer ontology. A short instruction about how test case parameter can be added to an existing ontology is given in appendix C.2.
All existing names for OWL-Classes, Object-Properties, and Datatype-Properties have to be unique within an ontology layer. Thus, some names look a bit cryptic in the EER diagram, especially the contain-relations are affected since most of the test case parameters

directly belong to the root entity.

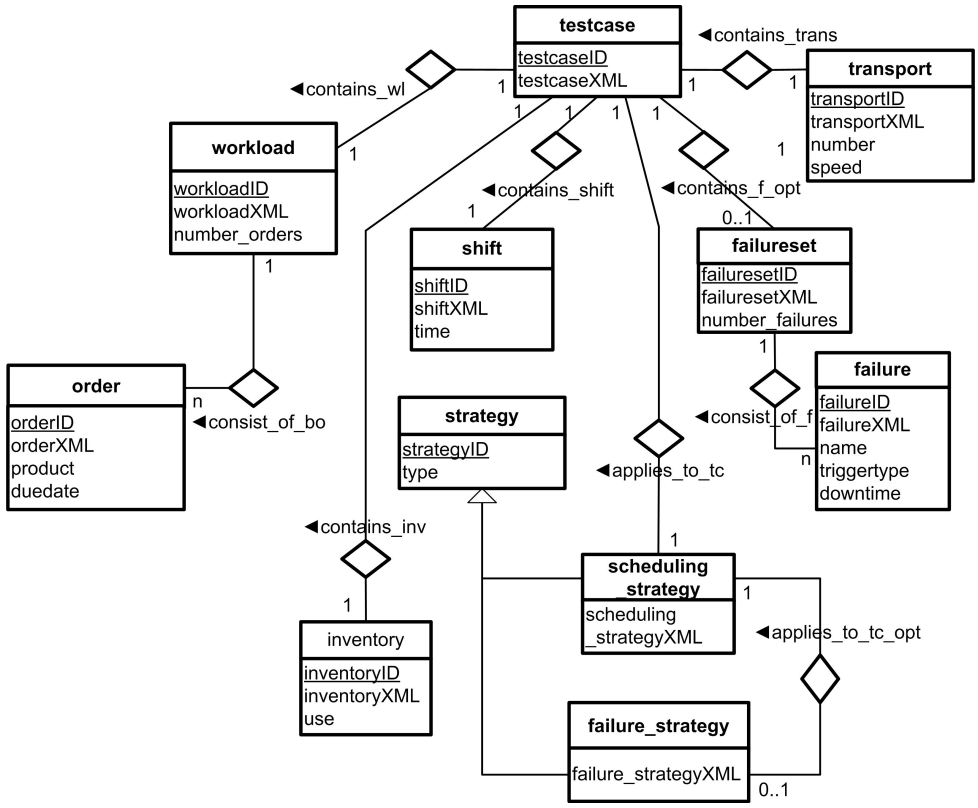


Figure 4.3: Model of the test case sub-ontology as layer of the SAW project ontology (EER diagram)

However, the implementation of the dynamic generic script assumes some naming conventions:

1. Each OWL-Class and each Datatype-Property should have a `rdfs:comment` to display a human readable parameter label in the GUI. Otherwise, the name of the OWL-Class or rather the name of the Datatype-Property is used as label name.

2. Each OWL-Class should have a parameterized XML Datatype-Property (`OWLClassName + "XML"`). Otherwise, the name of the OWL-Class is used as tag name in the XML file.

3. Each OWL-Class which participates in a "1 to n"-relation with the cardinality of 1 has to have a special Datatype-Property. The name convention is a concatenation of `"number_ " + ChildTagName + "s"`. This Datatype-Property is essential to identify the number of nested child tags during the creation of the XML file. The child tag is the OWL-Class on the other end of the "1 to n"-relation such as `order` and `failure` in the EER diagram (see figure 4.3).

Table 4.1 opposes the EER elements and the ontology elements. In addition, the third column gives information about how the GUI is affected by the different elements of the underlying data model. This interrelationship is explained in section 4.3.1. Furthermore a visualization is shown in appendix C.1. A valid and consistent parameter setting can be ensured as a consequence of the explained model transformation.

EER Element	Ontology Element	GUI Representation/Annotation
Entity	Class	Parameter group element
Attribute	DatatypeProperty	Test case parameter (text field)
Relation	ObjectProperty	not displayed in the GUI
Cardinality	if cardinality > 0	Parameter is mandatory
Data type of attribute	DataRange (datatype)	declares the validation rule
-	DataRange (set of literals)	declares valid values
-	comment (datatype)	defines the label name

Table 4.1: Mapping table between EER elements and ontology elements

Table 4.2 informs about how the structure of the ontology can be mapped to a corresponding structure of the XML file. A fragment of an example XML file is shown in section 4.3.3.

EER Component	Ontology Component	XML Representation/Annotation
Entity	Class	Start-tag
Attribute	DatatypeProperty	Attribute of start-tag
Relation	ObjectProperty	Identification of child tag (via Range)
Cardinality	cardinality	is not used
Data type of attribute	DataRange (datatype)	is not used
-	DataRange (set of literals)	is not used
-	comment (datatype)	is used for mapping (origin name)
-	statement contains "XML"	is used for mapping (tag name)

Table 4.2: Mapping table between ontology elements and XML file elements

4.2 Test suite generation

The process to generate test cases for SAW is specified in this section. Figure 4.4 shows the cyclic process on which the static specific approach and the dynamic generic approach approach are based on. Nevertheless, for the static specific approach exists some limitations (see section 4.3).

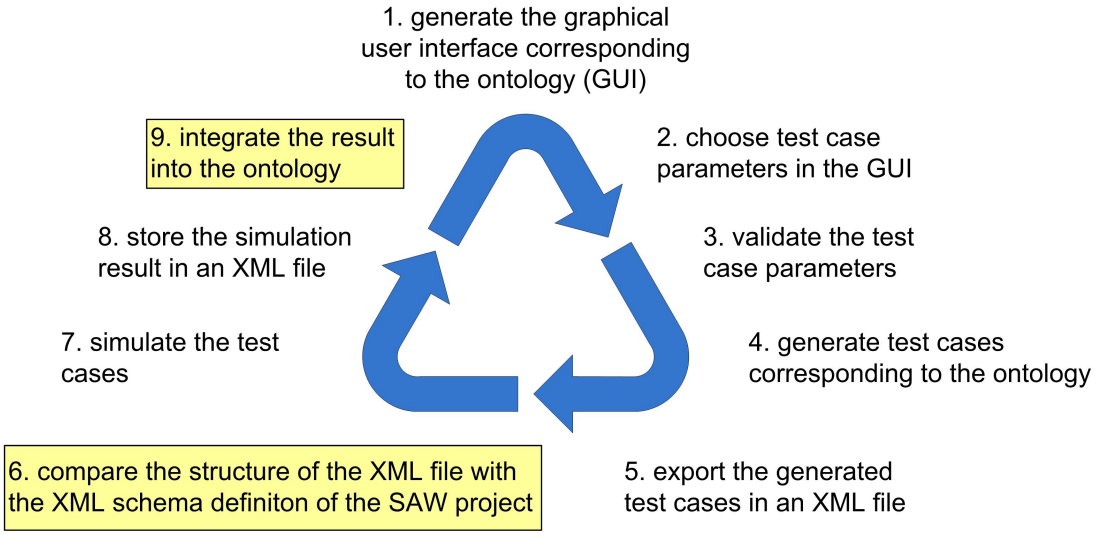


Figure 4.4: Process cycle of the test case generation

The cyclic process consists of the following nine steps:

1. **generate the GUI:** The basic idea in this step is that you can always choose from all parameters the ontology contains. This sounds easy, the realization, however, is complex. The main challenge is to generate a dynamic GUI at runtime. Building an ontology-corresponding GUI at runtime is essential to increase the acceptance by the target audience. Two further non-functional requirements are met with the help of the dynamic GUI, i.e. the usability and the extendability of new test case parameters. The design of the GUI and an instruction how to use it can be found in section 4.2.2.

2. **choose test case parameters in the GUI:** All selectable parameters are shown in the GUI. In addition it is possible to choose a specific value as well as a value range. An example for the scheduling strategy could be *First Come First Served* or a selection like *First Come First Served, Shortest Processing Time, Earliest Due Date*. It is also possible to select all available strategies by clicking on the check box behind the text field (see figure 4.5). In case of a value range or a selection of

more than one element the test case generator will choose a specific value out of the enabled values randomly.

3. **validate the test case parameters:** This feature is only available for the ontology-based approach. The chosen parameter set can be validated before the test cases are generated in step 4. The rules for the validation are listed in section 4.3.1 on page 66. A possible reason for an incomplete parameter setting is that a set of fallible components is chosen without an appropriate failure handling strategy. An incomplete parameter setting message together with a list of the incomplete parameters will be shown on the GUI in such a case. Afterwards, the user can adjust the parameter values before the generation of the test cases starts. This saves both computer power and time because usually the user notices that something went wrong **after** the simulation is finished.

4. **generate test cases out of an ontology:** This step exhausts further advantages of the ontology-based approach. The reasoner takes already simulated test cases and simulation results into account. The knowledge-based approach is a sophisticated feature because the generator does not always have to start from scratch. Furthermore, the ontology makes it possible to work in a semantic manner. Which means that the reasoner takes the results from assembly line A into account for assembly line B even when the machines are named differently but the machine functions are the same. The ontology provides a knowledge repository where an inference engine is acting on this repository. Those features go beyond the scope of this thesis and will be part of future works as outlined again in chapter 6. At the moment only the generation to get appropriate test cases related to the parameter setting is implemented.

5. **export the generated test cases in an XML file:** The snippet of an XML file to state the structure of the XML file is shown in the code fragment 4.3.3. It is important to mention that both the structure and the parameters of the test suite are related to the structure and parameters of the ontology and not necessarily to the simulation. In other words, the test suite might contain parameters which are not implemented in the actual simulation. This circumstance would lead to an error at runtime. The next step faces this aspect. However the generation process of the XML file as part of phase 3 shown in figure 4.10 includes a consistency check to make sure that the structure of the XML file corresponds to the structure of the ontology.

6. **synchronization between XML file and XML schema definition:** The synchronization between the XML structure of the generated test suite and the XML schema definition of the test cases for the simulation is beyond the scope of this thesis. Therefore, this step is marked in figure 4.4 to signalize that this issue will be met

in future works. Comparing the structure of the XML gile and the XML schema definition would make it possible to inform the user that not all of the chosen parameters are implemented in the actual simulation. Without step 6 the user will be informed about the circumstance immediately after the simulation tries to start. The feasibility study in section 4.3.4 shows how to face this problem.

7. **simulate the test cases:** The necessary input data for the simulation is shown in figure 4.6. The test cases (output of step 5), the layout of the assembly line (simulation object) and the path where the test result should be stored are mandatory data to start the simulation. The simulation software is developed by Rockwell and enhanced by students in the course of the SAW project. The detailed information about the SAW project can be found in section 4.1.2 on page 51. Firstly, the simulation tool builds a GUI based on the assembly line layout to visualize the assembly line and monitor the events during the simulation. Secondly, the test suite file will be parsed to instantiate the different test cases. Last the simulation will start to execute the test cases one by one.

8. **store the simulation result in an XML file:** The output file is created with respect to the interested events. The granularity level of the events can be defined by the user in step 1. Naturally, the higher the number of events to be logged the longer the size of the result file will be. The result files as output from each simulation are **not** the basic data for the evaluation which is described in section 5.2.2 on page 92. At first this fact might be confusing because the simulation result is the obvious reason to accept the effort of simulation. This is certainly true for testing the assembly line. The thesis, however, compares the static specific script and the dynamic generic script which are both scripts to generate test suites with respect to a parameter set. For that reason the focus of the thesis lies on the input and output components of the XOR element in figure 4.6 (see also figure 4.9 on page 64). In other words, one of the main goals of the thesis is to identify the strengths and weaknesses of the two generator scripts. For that challenge data about the costs to generate the test cases is essential. The metric to evaluate the scripts is defined in section 3.2. Further explanations will be given in the following sections, particularly in section 3 on page 45.

9. **integrate the result into the ontology:** This last step in the cyclic process is important for following simulations. The achieved simulation results will be integrated as experiences into the ontology. As a result of the feedback the knowledge captured in the knowledge base (ontology) increases. Firstly the goal of the thesis is to define a test process for the test case generation in section 4.2 on page 56. Secondly, a metric has to be found to make the costs of test description, implementation of parameters, and test coverage measurable. In conclusion, the evaluation between the

two approaches together with the following discussion part shows which approach makes the overall process shown in figure 4.4 easier for the target audience. This thesis will concentrate on the above-mentioned research issues while leaving step 9 out of scope. Step 9 would be an interesting topic to be discussed in detail in future works. Nevertheless, a feasibility study shows that step 9 is realizable (see section 4.3.4)

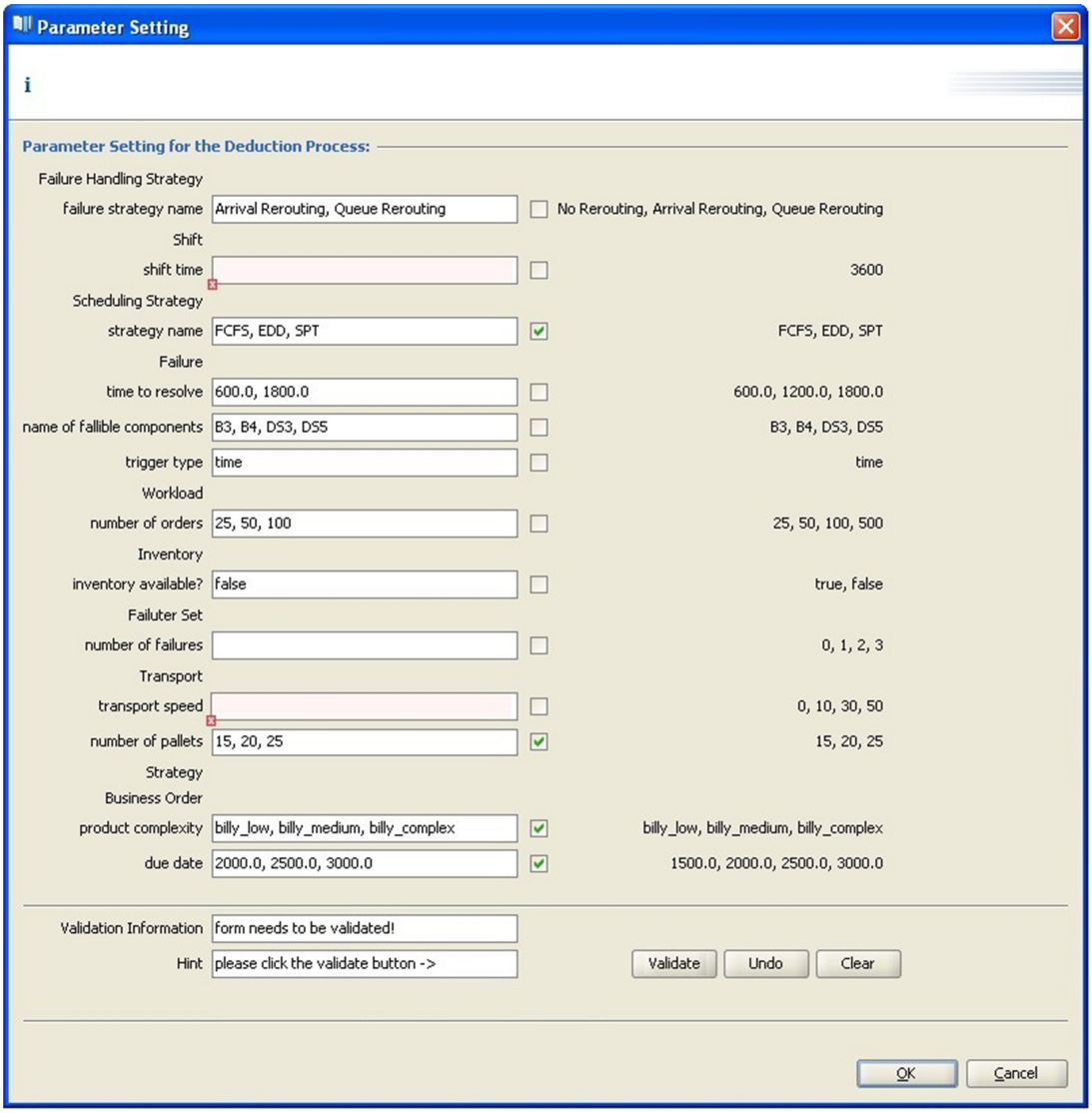


Figure 4.5: Screenshot of the parameter setting GUI

4.2.1 Simulation process overview

The detailed description of the nine steps of the cyclic test case generation process in the previous section will help to understand the simulation process shown in figure 4.6. The

simplified simulation process is linked with the steps indicated by the numbered cycles. The cycles' colors indicate whether steps were implemented during the work. The color blue means that it is part of the elaboration. Step 6 and step 9 are marked yellow and are not implemented during the elaboration. Nevertheless, a feasibility study shows that step 6 and step 9 are realizable (see section 4.3.4). The details about the simulator blackbox together with the related steps 6, 7 and 8 are shown in figure 4.8. The test case generation process as subprocess of the simulation process is explained in detail in section 4.2.2.

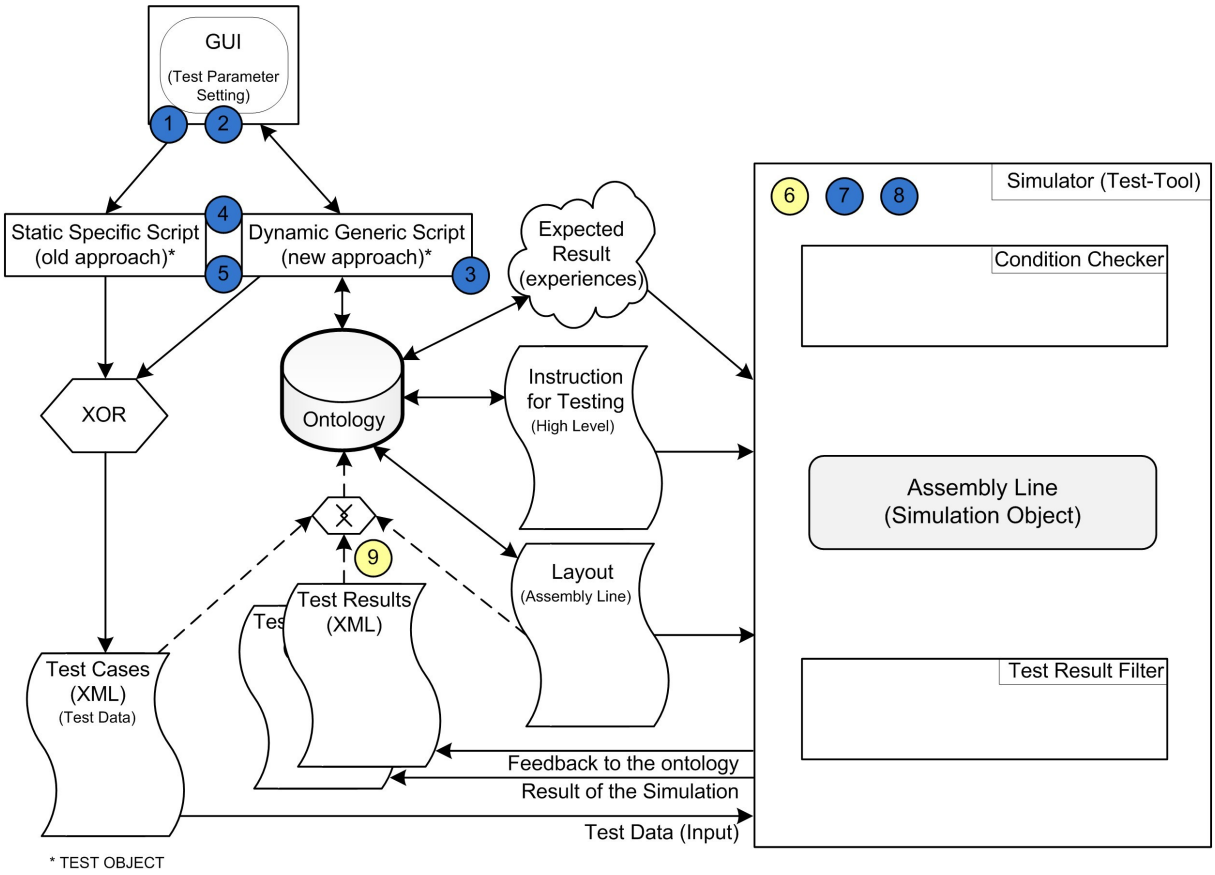


Figure 4.6: Overview of the simulation process (for details about the simulator refer to figure 4.8)

The following simulation scenario sums up the whole simulation process. The user wants to choose from all provided test case parameters to automatically generate input data for the simulation. Therefore, a GUI is offered. Furthermore, the user can decide which generator script should generate the test cases for the chosen parameter setting. Depending on the selected generator script either the static specific script or the dynamic generic script delivers the XML structured test case file as result. These test cases are used as simulation input data together with the layout of the assembly line and the directory

where the simulation results should be stored. This information is necessary to start the simulation. These necessary input data can be filled in a GUI which is offered by the simulator (see figure 4.7).
After this the simulator executes each test case one by one (step 7). Afterwards the results of the simulation can be found in the directory which has been selected. All events of interest are captured in the result file. Which events are considered to be of interest can be chosen as parameter at the beginning of the simulation scenario (step 8). The tasks to fulfill the steps 7 and 8 are performed by the simulator tool. Finally, the most important facts are integrated into the ontology corresponding to the generated test case and the assembly line layout. Step 9 generates the feedback to the ontology based on the test results file and not on the results file of the simulation. These two files are both log files but they capture different events performed during the simulation process. Whereas the results file of the simulation is used to analyze if the simulator works correctly, the test results are used to optimize the assembly line with respect to the overall throughput of the system. The feedback to the ontology allows to meet the criterion of the knowledge-based manner. As a result, experience can be extracted from the ontology which is used to expect the result drawn as cloud in figure 4.6 as well as the preconditions and postconditions of each test case before the simulation starts. This input data, the instruction for testing, the layout of the assembly line, and the mentioned test data specify a test case for the simulation ([41]).
Most important is that the defined test case with the generated test cases being the test input data to simulate the assembly line **does not correspond** to the test case of the test case generation process shown in figure 4.9. For this reason the assembly line is called simulation object whereas the test object is the static specific script and the dynamic generic script. In the next chapter the evaluation of the two scripts takes place and is discussed with respect to cost-saving potential and effectiveness.

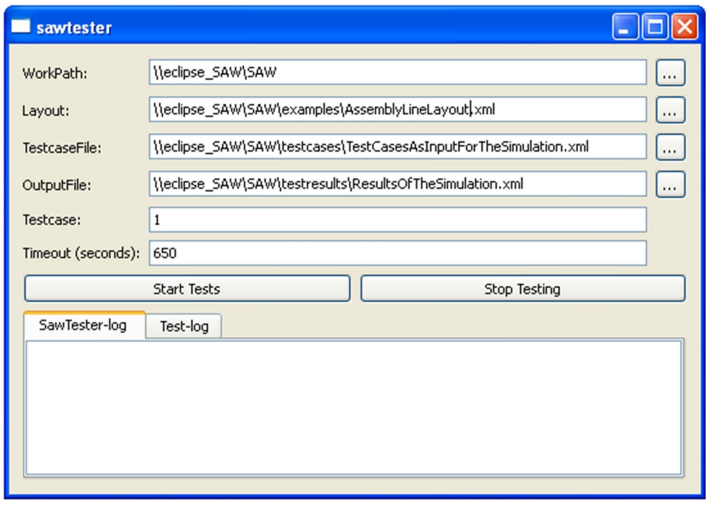


Figure 4.7: Screenshot of the SAW tester GUI

According to [41] the six input components of the AND element, i.e. the test cases to be simulated, the layout of the assembly line, the preconditions, instruction for testing, the expected results as well as the postconditions are essential to specify a test case (see figure 4.8). The instruction for testing is a human-readable manual which describes each step of the test. For running the simulation it is enough to know how to use the GUI shown in figure 4.7. In the following sections of the elaboration part the term test description is used in the context of how to manage the test case generation process. The output of the test case generation process is the test input data for the simulator. The test input data contains the test cases generated by the generator script. The layout of the assembly line is represented by an XML file which can be created with tool support. Therefore, the user can create and modify the model which represents his physical assembly line. The previously not mentioned input components which are expected results, pre-condition generator, and postcondition generator are extracted from the ontology. After the specified test case has passed the pre-condition check the test case can be automatically executed from the simulator.

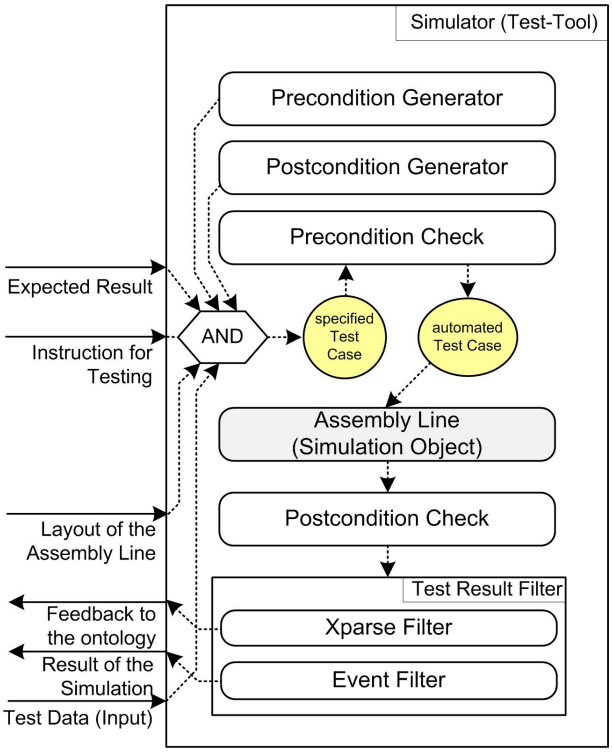


Figure 4.8: Detailed simulation process

Afterwards the postcondition check compares the results of the simulation with the expected results offered by the postcondition generator. Finally, the test result component filters and generates one output file per interest. The feedback file for the ontology usually

contains less data than the results of the simulation file in order to keep the ontology as compact as possible to be able to work on the ontology efficiently. However, the outlined process of the simulator is generated to identify potential to increase the quality of the simulation process. The realization of the process goes beyond the scope of this thesis and will be part of future works.

4.2.2 Generation process overview

The following figure 4.9 shows the generation process enhanced with the module "Cost-Performance Analysis Process". This module is used to evaluate the test objects, namely static specific script and dynamic generic script. The evaluation results can be found in chapter 4.9 on page 64. In general, the test case generation process is a subprocess of the simulation process described in the previous section. The flow of the test case generation starts with the GUI shown in the upper part of figure 4.9. The GUI offers the user the supported test case parameters from which he can choose the parameter setting by allocating values to those parameters. This parameter setting defines the parameters and their possible values for the test cases. Thus the parameter setting as input data configures the generator script. Two directed paths lead to the goal called test cases which are captured in an XML file. Let's start with the left path where the static specific script is the first component after the GUI. The edge which leads to the static specific script is unidirectional. This means that only a static GUI is supported by the static specific approach. This fact leads to the disadvantage that the GUI has to be modified after each modification such as the addition of a new test case parameter to the static specific script. The system changes frequently since the simulation software is continually improved by students. Thus, one of the most important features of the test case generator script is the expandability to support these changes. This is essential because the generated test cases are the input data for the simulation. In other words, most of the simulation changes effect the test case generation script since the test cases are the only input data beside the assembly line where the user can configure the simulation process.
The XOR element symbolizes that either the static specific script or the dynamic generic script can be used to generate test cases during one generation process. The generation process stops after the static specific script has generated the test case XML file.

Then there is the second path to walk through. Initially, the dynamic generic approach requires a dynamic GUI. This feature is indicated by the bidirectional edge between the GUI component and the dynamic generic script component. The dynamic generic approach extracts the offered test case parameters, the allowed values for the parameters, and structural information from the underlying data model. Afterwards the identified

information of the ontology is passed to the GUI. This data flow makes it possible to build a dynamic GUI which corresponds to the ontology at runtime. As a consequence, modifications to the ontology do not necessarily lead to manual changes neither to the GUI nor to the dynamic generic script. This feature allows to add new parameters in the ontology with tool support. Afterwards the modification is represented by the GUI without any line of code. Therefore the user needs no programming skills neither for adding new parameters nor for configuration of the generation process. Most of the complexity is moved from the user to the implementation of the dynamic generic approach. This is why the whole test case generation process could be simplified from the perspective of the user.

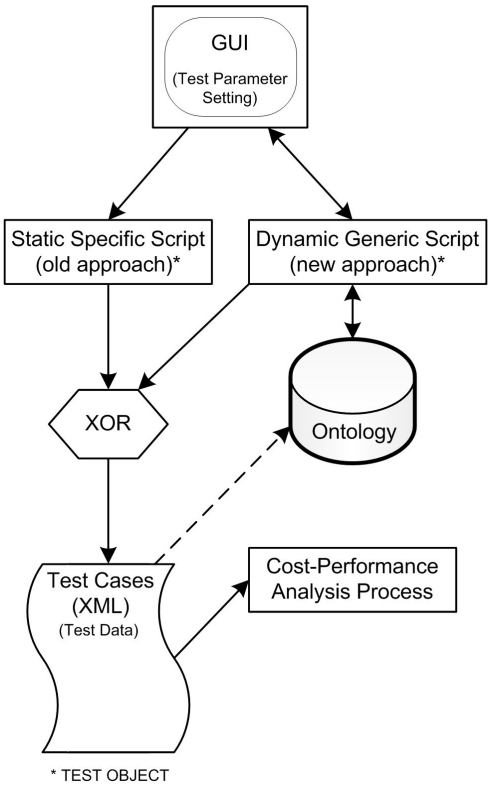


Figure 4.9: Overview of the test process as part of the simulation process

On the one hand the dynamic generic approach uses the underlying ontology's structure and restrictions to ensure the consistency of the test cases. On the other hand the data ranges of the offered parameters are used to ensure the validation of the test cases. Nevertheless, the user has to manage the underlying ontology. Therefore the user needs skills for modifying the ontology with common graphical editor tools like Protégé. Of course, for modifying an ontology users need some experience but it involves less effort than modifying a hard-coded script. Furthermore, if something goes wrong during the modification of the ontology the user will recognize it immediately in the parameter set-

ting configuration process. On the contrary, the user might realize that something went wrong during the modification of the static specific script after the simulation run took place by analyzing the simulation results. This is a frustrating experience as users usually do not know which one went wrong. In that case the user is captured in a trial and error loop. Surely, these circumstances negatively affect the acceptability of the static specific approach. Coming back to the starting point it needs to be said that the user can always choose from all test case parameters supported by the generator script to define the parameter setting. In addition, the dynamic generic approach ensures that only valid and consistent parameter settings can be defined by the user. This circumstance can be achieved with the information of the ontology. Afterwards, the dynamic generic script generates test cases based on the parameter setting. At the end of the path the dynamic generic script generates the XML file.

A useful characteristic of the *"3 Phases Process Model"* shown in figure 4.10 is the fact that the first phase and the second phase are totally independent of the domain. This is ensured as the domain specification is hidden in the ontology. The first phase communicates with the ontology and passes the element names, valid values for element instances, and information about the structure of the ontology to phase two. After this the second phase uses the received information about the ontology to build a dynamic GUI at runtime. The user can configure the third phase based on the data model easily. The implementation of the application's aim is located in the third phase only. In the thesis the third phase uses the chosen parameter setting to generate test cases. Afterwards the generated test cases are written into an XML file. This structured file can be used as input data for the simulation tool.

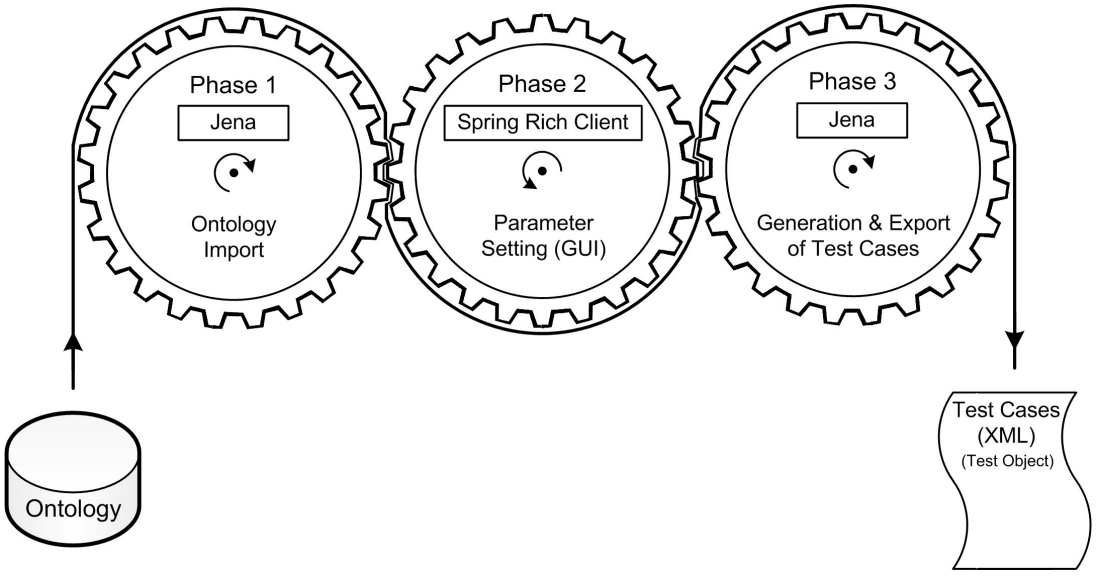


Figure 4.10: 3 phases process model

4.3 Realization of the generator approaches

The next section aims to look into the blackboxes shown in figure 4.9 to find out how they can be realized. Figure 4.10 shows the *"3 Phases Process Model"* and gives an overview about the different technologies used during the implementation of the dynamic generic approach. In the following section some low-level information together with code snippets are given to show how to deal with the implementation of the test case generation process. The numbered steps as subset of the cyclic nine steps process shown in figure 4.4 should make it easier to reference to the figures 4.14 and 4.11.

4.3.1 Implementation of the dynamic generic generator script

The paragraph at the end of the previous section described the flow of data throughout the whole test case generation process. This section gives information about the used technology in the different phases. The first phase reads the ontology with the help of the offered APIs from the Jena framework. For generating the text file which configures the dynamic GUI in the second phase some preliminary steps such as parsing, casting, caching, mapping, and filtering of data are necessary. The test case parameters are represented as data type properties within the different classes in the ontology. The comment property of the data type properties is used to display a human-readable label for the test case parameter in the GUI. In the ontology each data type property contains a data range element which holds one set of literals per allowed test case parameter value. The information about the allowed values together with the defined data types of the literals compose the validation information. The cardinalities of the relationships between the classes represented as object properties in the ontology inform about whether a parameter is mandatory. An important fact is that for each line in the configuration file a getter and setter method is assumed in the second phase. This is why the allowed values for each ontology class is set to "isClass" to save a line for extra layout information in the output text file. If the set of allowed values of a test case parameter contains an "isClass" then only the label of the parameter will be shown during building the dynamic GUI in the second phase. This is due to the fact that ontology classes represent groups of parameters. As mentioned above the test case parameters are represented by data type properties within the ontology classes (see also the mapping table 4.1 in section 4.1.3).

The screenshot in figure 4.5 shows the result of the first step. An interesting fact about the GUI is that it is generated dynamically at runtime. This is necessary in order to keep the selectable test case parameters up to date with test case parameters offered by the data model. Each line of the GUI represents one test case parameter by showing

its name, a text field, and a set of valid values for the parameter. The user can choose specific values out of the set of valid values for each parameter by writing them into the text field. In order to take all valid values of the parameter into account for the given parameter setting the user can enable the check box which is located between the text field and the valid values. The red symbol with the white x at the beginning of the text field denotes that the parameter is mandatory.

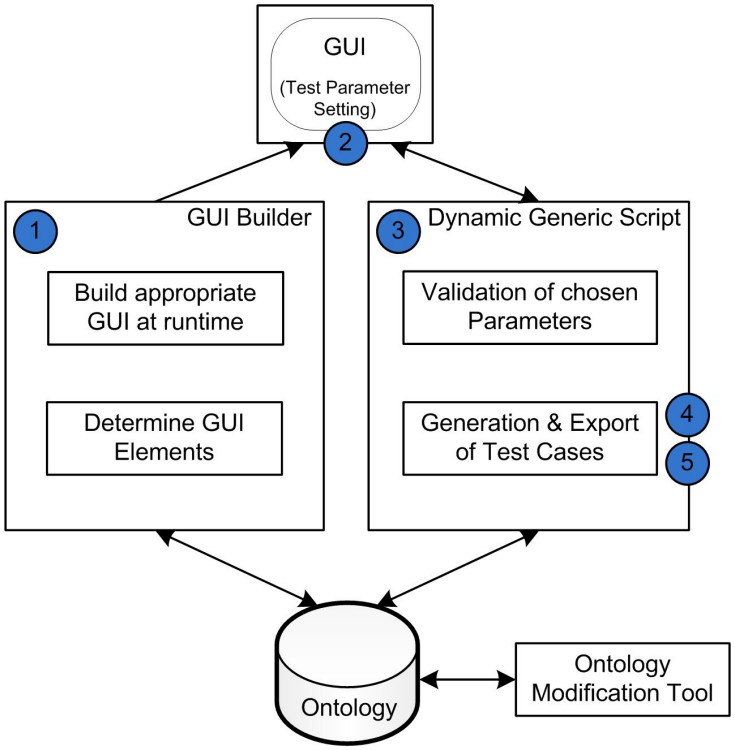


Figure 4.11: Structure of the realization of the dynamic generic approach

For getting an overview of the implemented dynamic generic approach it is helpful to link the structure of the realization shown in figure 4.11 with the already explained data flow shown in figure 4.10. The numbered cycles of the realization structure correspond to the direction of the data flow of the *"3 Phases Process Model"*. In short, the process to be implemented is defined as follows. The ontology defines the test case parameters, permissible values for each of those parameters, as well as the relations between them. These test case parameters have to be shown at a dynamic GUI at runtime. After the user has chosen a valid and consistent parameter setting the generation process takes place. Finally, the generated test cases will be exported into an XML file.

The following section explains the steps of the generation process enhanced with details about the used technologies. The starting point for the explanation is the ontology. The

section 4.1 gives detailed information about the ontology. Most important for the implementation is the EER diagram shown in figure 4.3. The ontology was created with Protégé which is an open source ontology editor. Of course, Protégé can also be used to modify the ontology defined as an OWL file. The ontology editors are illustrated by the `"Ontology Modification Tool"` component in figure 4.11.
In the first step all test case parameters are read from the ontology with the help of the Jena framework. The parameter groups are represented as classes in the ontology and as entities in the EER diagram. The parameters within the parameter groups are represented as datatype properties in the ontology and as attributes of the entities in the EER diagram. RDFS comments are used for the label names of the parameter since the names of the datatype properties are difficult to read. For building an appropriate dynamic GUI two further pieces of information are necessary. First the permissible values which are represented as data ranges in OWL, second, the cardinalities of the datatype properties (parameters) which are used to decide which of them are mandatory. Two classes namely a set of failures and the failure handling strategy are optional in the used ontology. Now all information for building the GUI is available.
However, the information has to be written in a text file to configure the dynamic GUI which is realized as a spring rich client application. The offered dynamic GUI at runtime leads to many advantages. Firstly, the user can only choose from supported parameters. Secondly, the validation component in the third step informs the user if a value is invalid. The user is also informed about a data type mismatch. This is the case if a character is typed and an integer or float value is assumed (see code fragment 4.3.1). Moreover, the information whether a parameter is mandatory enables a consistency check. The consistency check is also part of the third step. The parameter setting of the GUI is implemented as singleton since only one parameter setting can be chosen for each iteration of the generation process. The GUI passes the valid and consistent parameter setting to the fourth step. Afterwards the chosen parameter setting as well as the information about all combinatorial possible scenarios for the chosen parameter setting is displayed. The user can edit the parameter setting or start the test case generation process by clicking the "Run"button. The user has to define the two stop conditions namely the test coverage and the generation time in a form before the generation process can continue. Therefore the stop criterion which appears first applies.

For generating test cases a combinatorial collection of possible test cases related to the chosen parameter setting is created. Furthermore it can be ensured that the generated test cases are valid and consistent since they are related to the valid and consistent parameter setting. During the fifth step the export component creates the structure of the XML file corresponding to the structure of the ontology. Afterwards the dynamic generic script extracts test cases from the collection of possible ones and writes them iteratively into an

XML file until the number of test cases to fulfill the required test coverage is reached.

```
1  //the following class is adopted from the simple example application
2  //of the spring richclient framework
3  public class SimpleValidationRulesSource extends DefaultRulesSource {
4
5    //AlphaNum validator for mandatory paramters of datatype string
6    private final Constraint MANDATORY_ALPHANUM_CONSTRAINT =
7      all(new Constraint[] {required(), minLength(2),
8      regexp("[-_,a-zA-Z 0-9]*", "mandatoryAlphanumConstraint")});
9
10    //integer validator for mandatory paramters of data type int
11    private final Constraint MANDATORY_INT_CONSTRAINT =
12      all(new Constraint[] {required(), minLength(2),
13      regexp("[[, 0-9]*", "mandatoryIntConstraint")});
14
15          ⋮
16  /*
17  * 1.construct rules with respect to the restrictions of the ontology
18  * 2.bind the appropriate rule to each test case parameter
19  */
20  }
```

Code Fragment 4.3.1: Snippet of the validation rule source

Figure 4.12 gives an overview of the system architecture of the implemented prototype. The arrows show the data flow between the different layers of the test case generation application. In addition, the package names within the Java project are stated at the end of each layer. More information about the different classes within each package can be found in the package diagram shown in figure 4.13. The shown layered architecture is a typical MVC architecture (see section 2.1.4 on page 25). Whereas the `"Graphical User Interface Layer"` represents the view component and the `"Business Logic for GUI Layer"` represents the controller component of the MVC architecture. The model component and the controller component together define the user interface. Thus, they are both located in the `ui` package. The `"Business Model Layer"` represents the model component and therefore the `domain` package contains most of the functionality of the application with respect to the parameter setting phase (see also the second phase in figure 4.10). However, the `"Ontology Access Layer"` is responsible for the ontology import, for the test cases generation as well as for the test case export in an XML file (see also

the first and the second phase in figure 4.10).

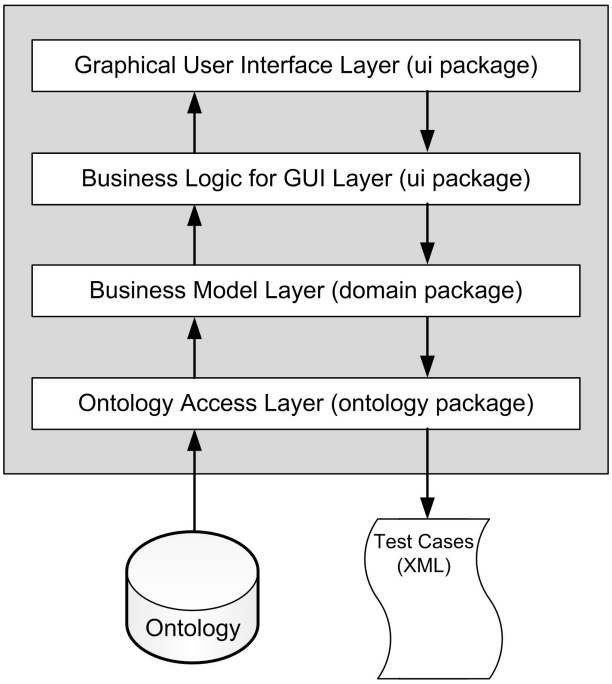


Figure 4.12: Architecture of the dynamic generic approach

UML provides thirteen diagrams to describe the structure and behavior of a system. The package diagram is used to describe the structure of a system by grouping parts of the system as packages. Therefore different abstraction levels can be used to reduce the complexity of the modeled system. [24]

In the package diagram of the test case generation application the grouped parts of the system are classes (see figure 4.13). The package names correspond to the package names of the Java project. The `"import"` relations in the package diagram indicate on which framework the package is based on.

Application (app package): Firstly, the application instantiates the `OntologyImport` class which provides methods to extract both structural information and domain information of the ontology. Secondly, the `MessagesPropertiesExport` object creates a configuration text file for building the dynamic GUI at runtime (see appendix B.2). The figures in appendix C.1 visualizes the idea behind the transformation. Afterwards the `TestCaseGenerationApp` gets instantiated to build the dynamic GUI with respect to the provided configuration file as output of the first phase (see figure 4.10).

Ontology Access (ontology package): The `ontology` package consists of four classes. Two of these classes use the Jena framework to access the underlying ontology. On the one hand, the `OntologyImport` class uses Jena to get the information which are necessary for creating the configuration file by the `MessagePropertiesExport` class during the first phase. On the other hand, the `TestCasesExport` class uses Jena for creating the XML file corresponding to the structure of the ontology during the third phase.

Model (domain package): The `domain` package offers the getter and setter methods for each data field. In the first place, the `ParameterSettingRulesBinding` class set the data type and if the data field is mandatory or not with respect to the configuration file for each data field. Afterwards the `SimpleValidationRulesSource` class links an appropriate validation rule to each data field based on the set data type and mandatory property by the `ParameterSettingRulesBinding` class (see code fragment 4.3.1). The data object for the parameter setting as well as the data object for the condition setting are represented by singleton instances. The parameter setting contains all chosen test case parameters with their corresponding values. The condition setting contains the stop criteria namely the test coverage and the maximal generation time.

User Interface (ui package): The two classes `ParameterSettingPropertiesDialog` and `ConditionSettingPropertiesDialog` are responsible for showing the appropriate form. However, the properties dialog classes are instantiated in the view classes since each form belongs to a view in the spring rich client platform. For more information about the spring rich client framework please refer to the section 2.4.4 on page 39.

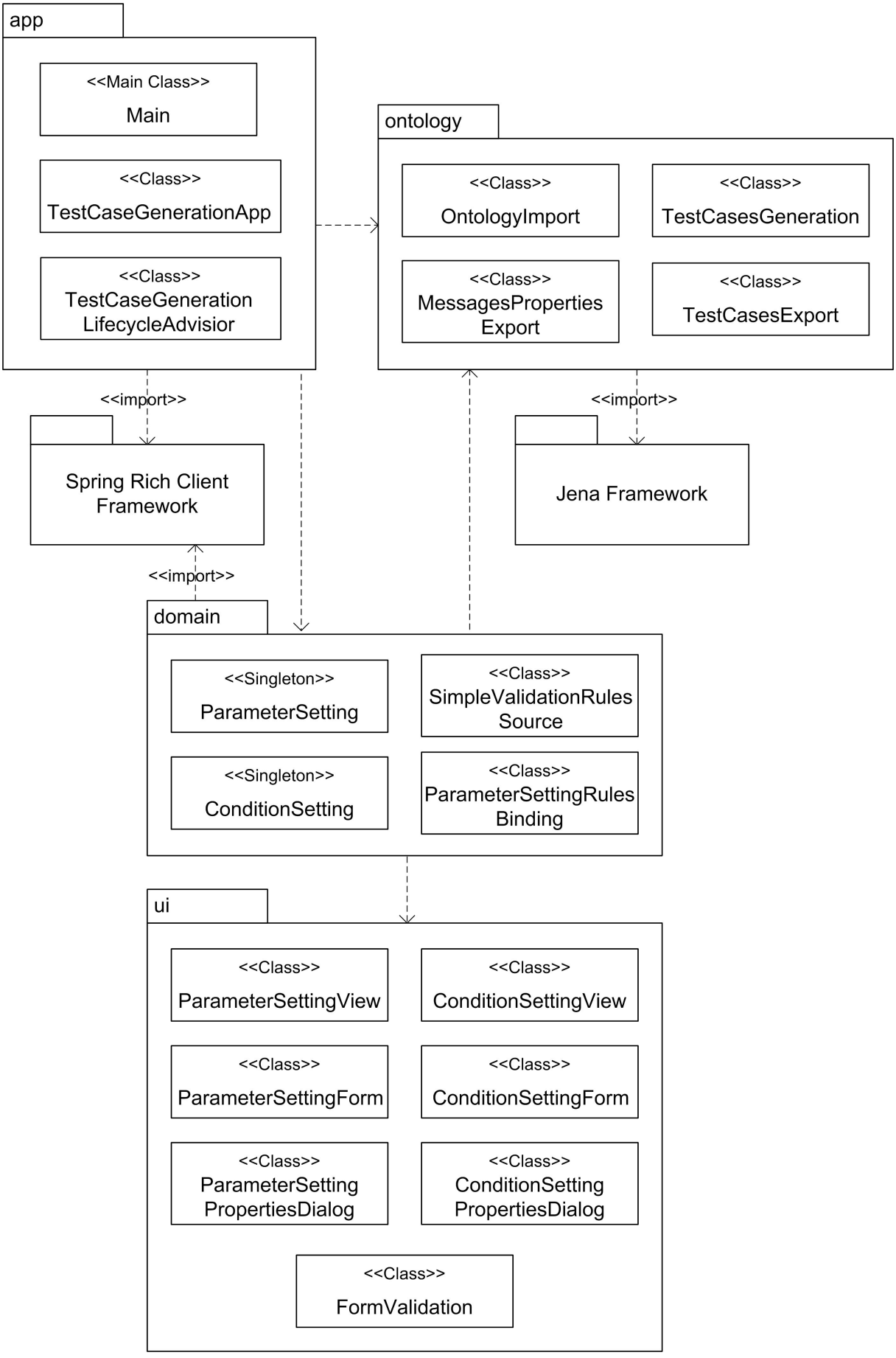


Figure 4.13: Package diagram of the dynamic generic approach

```
1  public class OntologyImport {
2    public OntologyImport() {
3      base.read(SOURCE, "RDF/XML");} //read the ontology
4
5    private String SOURCE = "testcaselayer.owl"; // path to the ontology
6    private String URL="http://qse.tuwien.ac.at/datamodel/testcaselayer";
7    private OntModel base=ModelFactory
8      .createOntologyModel(OntModelSpec.OWL_MEM);
9
10    public void importOntology() {
11      //list the statements in the Model
12      StmtIterator iter = base.listStatements();
13      Map<String, String> mapLabelComment =
14        new HashMap<String, String>();
15      Map<String, String> mapLabelCardinality =
16        new HashMap<String, String>();
17      String dataLabelComment = "", String dataLabelCardinality = "";
18
19      //print out the predicate, subject and object of each statement
20      //build comment map to display comment instead of param names
21      //build cardinality map for identifying mandatory parameters
22      while (iter.hasNext()) {
23        Statement stmt = iter.nextStatement(); // get next statement
24        Resource subject = stmt.getSubject(); // get the subject
25        Property predicate = stmt.getPredicate(); // get the predicate
26        RDFNode object = stmt.getObject(); // get the object
27
28        if (object instanceof Resource);
29        else { // object is a literal
30          if (subject.getLocalName() == null) { //it is a restriction
31            if (predicate.getLocalName().equals("cardinality"){
34              dataLabelCardinality = object.toString().substring(0,
35                object.toString().indexOf("b"));
36              mapLabelCardinality.put(subject.toString(),
37                dataLabelCardinality);
38          if (predicate.getLocalName().equals("comment")) {
39            dataLabelComment = object.toString().substring(0,
40              object.toString().indexOf("b"));
41            mapLabelComment.put(subject.getLocalName(),
42              dataLabelComment);
43    }}} ...} //end of the ontology import
```

Code Fragment 4.3.1: Snippet of the dynamic generic script

4.3.2 Implementation of the static specific generator script

The static specific generator script was provided at the beginning of the work. The aim of the thesis is firstly to locate weaknesses of the given static specific approach and secondly to find a solution to face those weaknesses. After the implementation of the new approach has taken place a evaluation shall make the effectiveness of both approaches comparable. The input parameters for configuring the test case generation process has to be set by modifying constants in the source code of the static specific script. Therefore, the missing step 1 of the static specific approach is not implemented during the elaboration part of the thesis. However, there is much tool support available for building a GUI at design-time. Additionally, the GUI would lack in maintainability and therefore it would relativize the achieved usability of the GUI. In other words, each change of the script would lead to changes in the GUI. Both tasks require programming skills of the user. Thus it is easier to do step 2 directly in the code of the static specific script (see code fragment 4.3.2).

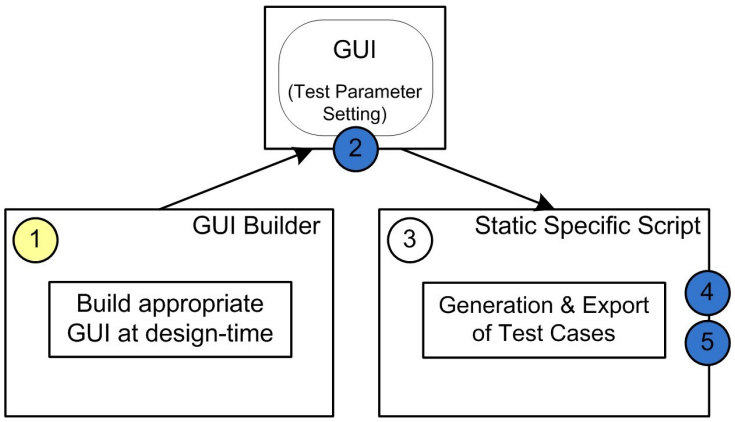


Figure 4.14: Structure of the realization of the static specific approach

A validation of the user input data is not supported by the static specific approach. Therefore, step 3 is colored white. The implementation of the generation step generates a set of all possible values based on the parameter values defined by the user for each test case parameter. Afterwards the static specific generator script takes one value from the set of possible values randomly for each test case parameter. After this the script writes the whole test case into an XML file. The last two tasks are iterative ones. Hence, neither a validation check nor a consistency check are supported by the static specific approach.

```
1  public class TestCaseGenerator {
2
3      // path to XML output file
4      private final static String outputFile = "testcases_autogen00.xml";
5
6      //production strategies (full class name)
7      private final static String[] strategies = {
8           "mast.saw.strategy.ShortestProcessingTime",
9           "mast.saw.strategy.EarliestDueDate",
10           "mast.saw.strategy.FirstComeFirstServe"};
11      //failure handling strategies (full class name)
12      private final static String[] failureStrategies = {
13           "mast.saw.strategy.failurehandling.NoRerouting",
14           "mast.saw.strategy.failurehandling.ArrivalRerouting",
15           "mast.saw.strategy.failurehandling.QueueRerouting",
16           "mast.saw.strategy.failurehandling.AllRerouting"};
17
18      //products (type and number)
19      private final static String[] products = { "billy_low",
20            "billy_medium", billy_complex};
21
22      //Workload
23      private final static int[] numberOfProducts = { 750, 1500 };
24           ⋮
25      /*
26      * test case generation and export in an XML file
27      */
28  }
```

Code Fragment 4.3.2: Snippet of the static specific script

4.3.3 XML structure of the generated test case

The XML file contains the generated test cases as output of the generator scripts. These test cases are the input data for the simulation.
The example of a test case shown in the code fragment 4.3.3 gives an overview of the basic structure. Most important of all is that the structure of the XML file corresponds with the structure of the ontology. The `tags` of the XML file are related to the `classes` of the

ontology and the `attributes` are related to the `Datatype-Properties` of the ontology (see also table 4.2). The skeleton of the XML file is specified by the relations between the classes of the ontology. The different test case parameters are described in section 4.4.

```
1  <?xml version="1.0"encoding="UTF-8"?>
2      <eventTypesToLog>
3        <type value="ProductFinished"/>
4      </eventTypesToLog>
5      <inputparameters id="1">
6        <workload number_orders="500">
7          <order duedate="2500.0"orderID="1"product="billy_complex"/>
8          <order duedate="1500.0"orderID="2"product="billy_medium"/>
9          <order duedate="3000.0"orderID="3"product="billy_complex"/>
10          ⋮
11          <order dueDate="2000.0"orderID="7200"product="billy_low"/>
12       </workload>
13       <scheduling_strategy type="ShortestProcessingTime">
14         <failureHandler type_fs="ArrivalRerouting"/>
15       </scheduling_strategy>
16       <inventory use="false"/>
17       <transport number="20"speed="0"/>
18       <shift time="3600"/>
19       <failures>
20         <failure id="1"name="DS5"triggertype="time"downtime="1200.0"/>
21           <!-- after 20 minutes simulation time -->
22           <!-- DS5 will be down for 20 minutes -->
23       </failures>
24       <simulation/>
25     </inputparameters>
26     <inputparameters id="2">
27       ⋮
28     </inputparameters>
29     ⋮
30 </testcases>
```

Code Fragment 4.3.3: Snippet of the test case XML file

4.3.4 Feasibility study

This section shows ways to deal with topics which are beyond the focus of this thesis. Firstly, the conceptual feasibility of the steps 6 and 9 of the cyclic test case generation process are shown (see figure 4.4). Secondly, a measurement concept to measure the test coverage and calculate the performance metric is identified. The last section gives information to face the structural limitations of the dynamic GUI which is part of the dynamic generic approach.

4.3.4.1 Structural comparison between XML file and XML schema

The XML file as output of the test case generation process contains the generated test cases in a structured form. The question is whether the structure of the file is the structure expected by the simulator. The test case structure supported by the simulator is defined in the XML schema (see appendix B.3). Thus, the structural comparison is a consistency check.

Section 4.3.1 highlights that the dynamic generic approach ensures valid and consistent test cases. This is certainly true, but with respect to the corresponding ontology as data model and not necessarily to the XML schema of the simulator. As a result, the sixth step of the cyclic test case generation is necessary for both generator scripts.

Firstly, the XML schema has to be referenced. Secondly, the XML file has to be validated corresponding to the XML schema until the end of the file is reached or a failure occurs. Lastly, the user is informed about the result of the consistency check. Code fragment 4.3.4.1 shows a solution to prove the consistency.

```
1  public class XmlFileStructureValidation {
2        ⋮
3    public static final String XML_SCHEMA_TESTCASE =
4      "schema/testcases-1.4.xsd";
5    File xmlFile = new File("testcases_output_step5");
6
7    public static void validateXmlFile(File xmlFile)
8      throws XmlException, IOException {
9
10     // instancate the XML Schema
11     SchemaFactory sfac = SchemaFactory
12       .newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
13     Schema xmlSchema = null;
14     try {
15       xmlSchema = sfac.newSchema(new File(XML_SCHEMA_TESTCASE));
16     } catch (SAXException e1)
17       throw new XmlException("Error while parsing the schema", e1);}
18     }
19
20     Validator validator = xmlSchema.newValidator();
21
22     // validate the XML file corresponding to the XML schema
23     try {
24       validator.validate(new StreamSource(xmlFile.toURI().toString()));
25     } catch (SAXException e) {
26       throw new XmlException(
27         "the provided XML file is not a valid testcasefile", e);
28     }
29   }
30       ⋮
31 }
```

Code Fragment 4.3.4.1: XML file and XML schema comparison (source: SAW project)

4.3.4.2 Feedback of the simulation result into the ontology

The basic idea of the feedback into the ontology is that the cyclic test case generation process has not to start from scratch each time. An important fact is that the feedback is not limited to the simulation results since the results depend on the test cases and the assembly line.

A feedback consists of the following artifacts (see figure 4.6):

- Test cases
- Layout of the assembly line
- Simulation results

Each artifact should be stored as feedback into a separate ontology for maintainability reasons. Thus, a unique key has to be created to make the artifacts referable for subsequent iterations of the cyclic test case generation process.
It is also possible to store the feedback without a merged key of the artifact keys since ontologies support reasoning techniques. In this case, a reasoner could infer the information for each iteration. On the one hand, the extraction of experience by reasoning techniques is more flexible. On the other hand, the extraction of feedback stored by using a merged key is probably faster.
However, the aim of the ninth step of the cyclic test case generation process shown in figure 4.6 is to integrate a feedback into the ontology. Therefore, the principle tasks are the creation of ontology individuals and a way to store them to the ontology. Code fragment 4.3.4.2 shows a solution to face these tasks.

```
1  public class OntologyFeedback {
2          ⋮
3    // define URL, NS and read the OWL model (see section 4.3.1)
4
5    public static void createIndividual(String individualName,
6      String individualClass, OntModel base) {
7      // get resource of the target class
8      Resource res = base.getResource(NS + individualClass);
9      // add the individual to the class
10      base.createIndividual(NS + individualName, res);}
11
12    public static void addPropertyToIndividual(String propertyLiteral,
13      String propertyName, String individualName, OntModel base) {
14      // get target individual
15      Individual ind = base.getIndividual(NS + individualName);
16      // get the property
17      Property prop = base.getProperty(NS + propertyName);
18      // add the literal to the property
19      ind.addProperty(prop, propLiteral);}
20  }
```

Code Fragment 4.3.4.2: Integration of feedback into the ontology

4.3.4.3 Measurement concept of the performance metric

The test coverage combined with the costs to achieve this test coverage is used as performance metric. In our context the test coverage is the ratio between generated test cases and all possible test cases for a given set of parameters. The necessary information to calculate the test coverage could be logged as comment at the end of each generated test case suite. The test case suite is the output file of the test case generation process. The meta information of the generation process are represented as comments at the end of each test suite and can be therefore extracted from the XML file to determine the test coverage. For the component `"Cost-Performance Analysis Process"` shown in figure 4.9, for instance, a spreadsheet program could be used to process these meta information.

The following meta information are necessary to calculate the performance metric:

- number of generated test cases
- number of all possible scenarios for a given parameter setting
- generation time for the provided test cases

4.3.4.4 Optimization of the dynamic GUI

The term reflection was first introduced by *Smith* in 1982 as a principle that allows a programmer to access, reason about and alter its own interpretation (found in [19]). Java supports the concept of reflection by allowing access to the Java bytecode as intermediate form ([2]).
The Java reflection API is supported by the classes in the `java.lang.reflect` package. The Java reflection examines the class of an object and determines its structure. It is possible to find out which constructors, methods, and fields a class has, as well as their attributes. Furthermore the programmer can change the values of fields, dynamically invoke methods, and construct new objects. [36]
Therefore Java reflection seems to be a powerful technique to analyze the class at runtime. The reflection technique is used to inspect the getter and setter methods of classes and to invoke them at runtime. This is necessary since the number of methods might vary over the application's life cycle. However, one limitation of the Java reflection is that the standard reflection API does not support to alter program behavior. *Chiba* discusses how this limitation can be addressed with an extension to the reflection API called *Javassist*

([15]).
The application developed during this work has to face the challenge that the number of the needed getter and setter methods depend on the number of test case parameters offered by the ontology. For fulfilling the task of adding getter and setter methods automatically a structural reflection is required. Without the structural reflection technique the number of supported test case parameters is limited to the implemented number of getter and setter methods at design-time. At the moment the solution to face this limitation of expandability is to implement a high number of getter and setter methods. This approach lacks in resource efficiency because in most cases the number of required parameters is lower than the number of parameters supported by the static system structure.
OpenJava is a macro system for Java which provides a data structure called class metaobjects ([13]). The OpenJava framework also allows modification of class definitions. OpenJava and Javassist restrict structural reflection within the time before a class is loaded into the JVM. However, the OpenJava is source-code based and Javassist is byte-code based. The listed characteristics are taken from the work [15]. This work includes several further facts about OpenJava and Javassist as well as a performance evaluation between those two.

The following table 4.3 gives an overview of different reflection technologies.

Area of Operation	Technology	Description	Abilities	Constraints
Source Code	Reflective Java	Preprocessor	Interception of method calls	Source code is necessary
Compile Time	OpenJava	uses the meta-object protocol to modify the source code	Interception of method calls Enhancement of the syntax	Source code is necessary
Byte Code	Javassist Kava Dalang	modifies the byte-code at load-time	no source code is necessary	requires offline preprocessing
Runtime	Metaxa Java Reflection API	Reflective JVMs	Interception of method calls Introspection at runtime	proprietary JVM only introspection

Table 4.3: Reflection technologies (according to [45])

4.4 Test case parameters

The following sections describe test case parameters currently provided in SAW. In the first two sections the production strategy and the failure handling strategy are explained in detail. These strategies are essential to optimize given assembly lines reasonably and are therefore two of the most important test case parameters. Further test case parameters are described in the last section.

4.4.1 Production strategies

A production strategy also called scheduling strategy serializes the tasks to process all orders received by using dispatching rules. Every strategy creates the ordered task list with respect to their specific criteria such as, for instance, earliest due date of the order, shortest processing time of the imminent operation and total processing time of the order. At the beginning of the thesis only the simple strategy called First Come First Served was implemented. In this strategy the jobs are processed depending on the order of their arrival. Advanced strategies aim at increasing the throughput of the whole system. The two scientists *Chiang* and *Fu* benchmark twelve strategies which can be used for job scheduling with the due date in focus ([14]).

Production strategies can be classified into static and dynamic strategies. The static strategies order the tasks before the simulation of the shift starts. Therefore, no shift time is taken into account. Static strategies are suitable if no system disturbances can be assumed. Mostly, the system states are not predictable in a dynamic and uncertain environment. Dynamic scheduling strategies have the ability to react to system state changes. For this purpose the order of the tasks can be changed during shift runtime. Such scheduling decisions as reaction to changes have to be done quickly. The following section describes both static and dynamic dispatching rules according to [14] and [26]. The implementation which is done during the work on the thesis is limited to the static production strategies. On the one hand no inventory is available in SAW at the moment, which would be necessary for most dynamic strategies to store intermediate products that currently cannot processed during the shift. On the other hand aim of the thesis is to test the performance of the generator scripts and not to find an optimized production strategy for a given assembly line with the help of SAW.

First Come First Served (FCFS): The FCFS strategy is one of the simplest production strategies. Each product of the order list is produced sequentially, which means that only one order is processed at a time.

Earliest Due Date (EDD): The production strategy prioritizes products with the earliest due date. The strategy requires enough production capacity in the workshop to achieve a good performance. Furthermore the slack time is not taken into account.

Shortest Processing Time (SPT): The SPT is much more complex than production strategies as the orders are produced one at a time. The SPT results in a processable task list without grouped order tasks. This means that usually intermediate products of different orders are produced before the order can be finished. As consequence a high number of pallets or an inventory is necessary. An example is given in appendix B.2 on page 115. The estimation of the processing time is based on the processing time at the machine without taking the transportation time into account. In some cases the machine load time and the transportation time can affect the processing time significantly ([32]).

Shortest Remaining Processing Time (SRPT): The order with the shortest remaining processing time is sequenced first. Besides the processing time the transportation time is also included in the calculation.

Processing Time Divided by Job Processing Time (PDJT): The PDJT gives the highest priority to the operation with the smallest ratio of the operation processing time to the job processing time.

Shortest Job Processing Time (SJPT): Selects an order with the shortest job processing time. The shortest job processing time is defined as the sum of all processing times which are necessary to finish the order.

Operation Due Date (ODD): The ODD gives the highest priority to the operation with least slack time $s´$.

Slack: This first explained dynamic strategy gives the highest priority to the order with the lowest slack time.

Modified Due Date (MDD): In a first step the strategy determines the tuples of due date and remaining processing time for each order. Afterwards the maximum value of each tuple is determined. Finally, the minimum of the maximum values gets the highest priority (maxd, t+r). In short, an order with the minimum modified due date is selected.

Modified Operation Due Date (MOD): The process to determine the highest priority is similar to the MDD strategy. Instead of the due date of the order the slack of the operation $s´$ is used. Thus, the MOD gives priority to the operation level whereas the MDD gives priority to the order level. In short, an operation with the minimum modified operation due date is selected.

Cost OVER Time (COVERT): The COVERT strategy takes the costs for finishing products after the required due date into account. Therefore the operation with the largest ratio of expected delay penalty to the processing time is given the highest priority.

Apparent Tardiness Cost (ATC): The operation with the largest apparent tardiness costs is prioritized.

Critical Ratio (CR): The critical ratio is defined as r/(d-t). Thus, the orders with a high remaining processing time and a due date in the near future related to the actual shift time are given the highest priority.

p	processing time of the imminent operation
r	remaining processing time of the job (including p)
P	total processing time of the job
q	queuing time of the imminent operation
t	system time, the time at which the dispatching decision is to be made
d	due date of the job
s	slack of the job, s = d-t-r
$s´$	slack of the operation, s´= d-c(r-p), where c is a parameter
l	average processing time of the operations at the machine
$l´$	total processing time of the operations at the next machine

Table 4.4: Notations of job and operation properties (according to [14])

Figure 4.15 gives an overview of the effectiveness of both static scheduling strategies and dynamic scheduling strategies. The CR and CRT are dynamic scheduling strategies. The ordinate of the diagram represents the throughput and the axis of abscissa represents the number of available pallets during the shift. The SPT and CRT are represented by dashed lines in the diagram. This is due to the fact that they are taking the transportation time into consideration which positively affect the throughput.

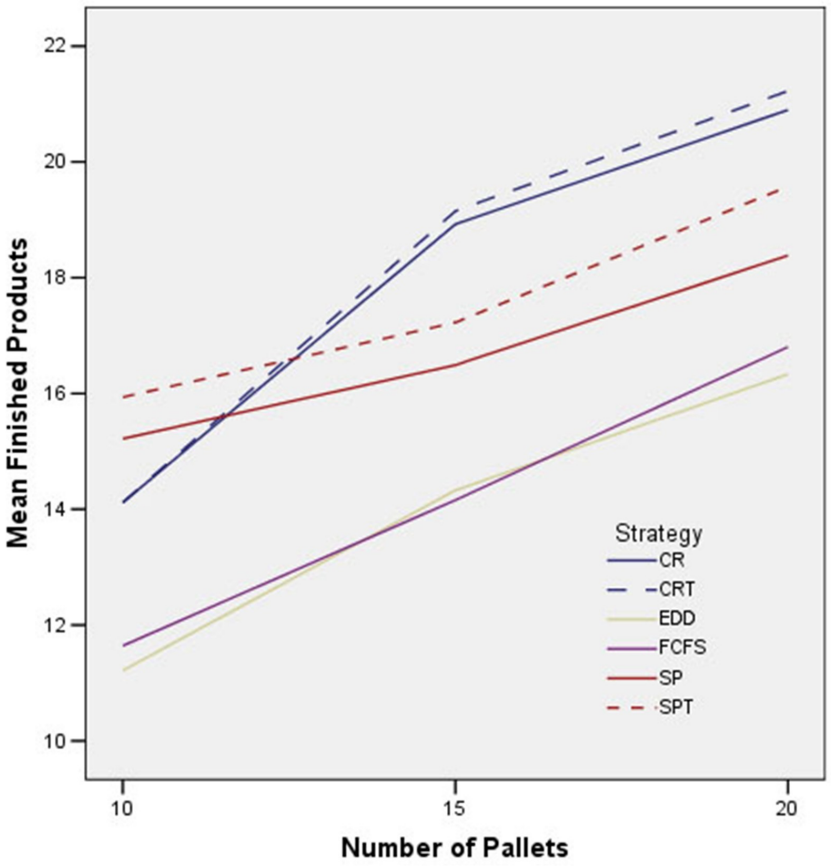


Figure 4.15: Production effectiveness without failures for 6 work scheduling strategies (according to [32])

4.4.2 Failure-handling strategies

In SAW the conveyor belt components and the machine components can be simulated as fallible. The time when a failure of an enabled component occurs is based on a normal distribution. The "number of failures" test case parameter is the upper limit of the number of failures which can occur during the shift. The simulation system reroutes the pallets automatically in case of a conveyor belt failure event.
In case of a machine failure different failure handling strategies can apply. These strategies should help to balance both the jobs in the system and the arriving jobs while the machine is out of service. There are three rerouting policies, as well as the possibility to do nothing and just queue the jobs of the failed machine. The aim of the different failure handling strategies is to reroute the queue of the failed machine, to reroute the new arrivals to working machines or to do both.

The following four alternative reactive scheduling policies were implemented ([28]):

No Rerouting (NR): In the no rerouting policy the jobs are kept in the queue of the failed machine during the repair time. Additionally, the new arriving jobs are also routed to the broken machine by the load balancer. This means that the simulation does nothing to increase the throughput in case of a machine failure by using the no rerouting policy. After the machine's repair is completed, the queue of the machine is processed sequentially. This failure handling strategy is used as default policy in SAW.

Queue Rerouting (QR): The queue rerouting policy reroutes the jobs which are in the machine queue of the broken machine at the time the failure occurs. Thus, the repair time is used for finding a suitable machine with the functions required to process the job instead of just waiting until the broken machine is working again. New arriving jobs are not affected by this policy and therefore are queued in the broken machine.

Arrival Rerouting (AR): The arrival rerouting policy reroutes the affected new arriving jobs by the machine failure to alternative machines while the original machine is down. The jobs in the queue of the failed machine are kept in the queue. The rerouting of the arrivals takes place immediately after the failure occurs. Therefore a short path to the nearest alternative machine can be ensured without making a detour to the broken machine first.

All Rerouting (AAR): This policy is a combination of the queue rerouting and the arrival rerouting policy. It reroutes all affected jobs by the machine failure. Thus, the queued jobs of the broken machine as well as the new arriving jobs are rerouted to alternative machines. The rerouting is done only during repair time as for all introduced failure handling strategies.

4.4.3 Further test case parameters

number of orders (workload): The workload consists of a list of orders. Each order is represented by a type of product and the related due date.

strategy name (scheduling strategy): The different scheduling strategies order the tasks items which are necessary to manufacture the product with respect to their specific scheduling characteristics (see also section 4.4.1).

product complexity: The complexity of the production tree defines the type of a product. At the moment it is possible to choose from three different types, which are namely billy_ low, billy_ medium and billy_ complex. The product trees of these products can be found in appendix on figure B.2 on page 115. However, product

trees are represented by XML files and can therefore be defined easily.

due date: The lower and upper limits of the due date define the value range. Each due date has to be within this range. The prototype of the dynamic generic script explained in section 4.3.1 supports only a fixed set of due date values.

number of pallets: The "number of pallets"parameter defines the number of available pallets during the shift simulation.

shift time: This parameter defines the duration of the shift in ms.

transport speed: The transport speed allows to define the simulation speed. This is a major advantage in SAW since it enables to simulate more shifts in the same period of time. The value 0 for the transport speed means that the system simulates the test cases as fast as possible.

failure distribution: The time when failures occur are based on a normal distribution. At the moment the trigger type when a failure event occurs is set to time.

name of fallible components: Fallible components are those components in the assembly line which can have a failure during the shift.

number of failures: Is the upper limit of the number of failures which can occur during the shift.

time to resolve: This is the downtime of the component after the failure has occured.

events of interest: The listed events of interest narrow down the output file. Thus, this parameter affects the size of the result file since only events of interest are logged. A common example for an event of interest is the finished product event.

Chapter 5

Discussion

This chapter gives answers to the research issues and has the same structure as the research issue chapter consequently. In this section the benefits of the dynamic generic approach are summarized. Afterwards the feasibility and the cost-saving potential of the dynamic generic approach are discussed. Lastly, the developed test process for SAW is discussed in detail.

The following lessons could be learned in the dynamic generic approach throughout this work:

1. High-level test description improves usability.
   No programming skills are necessary to configure and run the test case generation process since the implemented prototype offers GUI assistance throughout the whole generation process.

2. Low implementation effort supports expandability of parameters.
   No programming skills are necessary to add new test case parameters. Nevertheless, the target audience needs some experience to modify the ontology but it involves less effort than modifying a hard-coded script (see appendix C.2).

3. Users can define the test coverage.
   After the parameter setting took place the dynamic generic approach calculates the number of all possible test case scenarios. Afterwards the user can set the test coverage to be achieved by the test case generator.

4. The dynamic generic approach reduces costs.
   In the dynamic generic approach the generation process is configured faster than in the static specific approach. Moreover, the Return on Investment (ROI) of the

higher effort for the first establishment is achieved after 38 implemented parameters (see section 5.2).

5. Consistent and valid test cases are generated.
   The dynamic generic approach ensures consistent and valid test cases as output of the generation process. Firstly, the structure of the ontology is used to determine whether a parameter is mandatory to ensure the consistency. Secondly, valid values for a test case parameter are defined by the `DataRange` element of the `DatatypeProperty` element in the ontology (see appendix C.1). Thus, the validation can be proven for each parameter chosen by the user.

6. The configuration phase of the dynamic generic approach is domain independently.
   The first and second phase of the dynamic generic approach are totally independent of the domain as the domain is hidden in the underlying ontology (see figure 4.10).

7. The architecture meets the requirement of expandability.
   The dynamic generic approach allows to add new test case parameters without any line of code (see appendix C.2).

8. The architecture meets the requirement of maintainability.
   The test case parameters and the structure of the data model can be maintained by the target audience with tool support. Most important is that no changes to the dynamic generic script are necessary.

9. The dynamic generic approach decreases the risk of making mistakes.
   The parameter setting form displays all test case parameters together with their valid values supported by the ontology. In addition, the parameters which are mandatory are marked explicitly by symbols (see figure 4.5).

10. Mistakes during the modification of the ontology are detected in an early stage.
    The underlying ontology as data model has to be modified to add a new test case parameter or to amend an existing one. Tool support exists to modify an ontology. The user will notice in the configuration phase if something went wrong during the modification since the parameter setting form in the GUI represents the data model.

5.1 Feasibility of the dynamic generic approach

The implemented prototype of the dynamic generic approach shows the feasibility of the high-level test description, the generation of consistent and valid test cases, and the expandability with tool support. All these requirements are essential for a good solution.

The developed *"3 Phases Process Model"* shown in figure 4.10 gives an overview of the different phases to fulfill the test case generation process. An important fact is that the first and second phase of the dynamic generic approach are totally independent of the domain as the domain is hidden in the underlying ontology. Firstly, the structural information of the ontology is used to ensure that the parameter setting is consistent. Secondly, the literals in the ontology are used to define valid values for each test case parameter. This information is necessary to configure the GUI in the second phase (see appendix B.2).
Based on the configuration file as output of the first phase a dynamic GUI for the parameter setting is created at runtime (see figure 4.5). As a result, the consistency and validation of the parameter setting chosen by the user can be ensured. A high-level test description can be achieved since the implemented prototype offers GUI assistance throughout the whole generation process. In other words, no programming skills are necessary to configure and run the test case generation process. Furthermore, the user can set the test coverage to be achieved by the test case generator. The dynamic generic approach allows to add new test case parameters with help of tool support (see appendix C.2).
The third phase generates test cases based on the parameter setting and writes them into an XML file. The structure of the generated test cases corresponds with the structure of the ontology to ensure consistency.

5.2 Identification of cost-saving potential for the ontology-based approach

This section is structured into two subsections. The first subsection gives an overview of the evaluation results including a brief discussion. In the second subsection, detailed information about how these results were achieved can be found.

5.2.1 Overview of the results

The results of the evaluation are listed in table 5.1. The result cube shown in figure 5.1 illustrates these results. The criteria of the evaluation are explained in detail in section 5.2.2.
The green-colored fields of the result cube visualize that the performance of the corresponding approach is significantly higher than the performance of the other approach. The result cube has three dimensions, namely the used technology, the application domain, and the cost unit. The technology is the script used to fulfill the task of generating test cases. In other words, the technology is the object under test. The application domain informs if the focus of the evaluation was laid on a constant number of parameters or on

the expandability of parameters. The objectives, namely the costs for the test description, the effort for the implementation of parameters, and the question whether the test coverage is definable can be seen on the cost unit dimension. This multidimensional result cube consists of twelve cuboid parts, six for each technology. If one cuboid is colored green then the equivalent cuboid of the other technology has to be red-colored because only one technology can perform better with respect to the same objective and the same application domain.

As can be seen, the dynamic generic approach performs better on most objectives. The dynamic generic approach demands a higher effort for the first time implementation. The evaluation in section 5.2.2 shows that the Return on Investment (ROI) is achieved after 38 implemented parameters with experience of using the generator scripts assumed. In addition, the user needs to obtain the necessary skills for the used generator script. For using the static specific script the user needs programming skills for both generating test cases and adding new parameters. The dynamic generic script assumes skills in using an ontology editor for adding new test case parameters to the ontology (see tutorial in appendix C.2). The dynamic generic approach requires no specific skills for generating test cases.

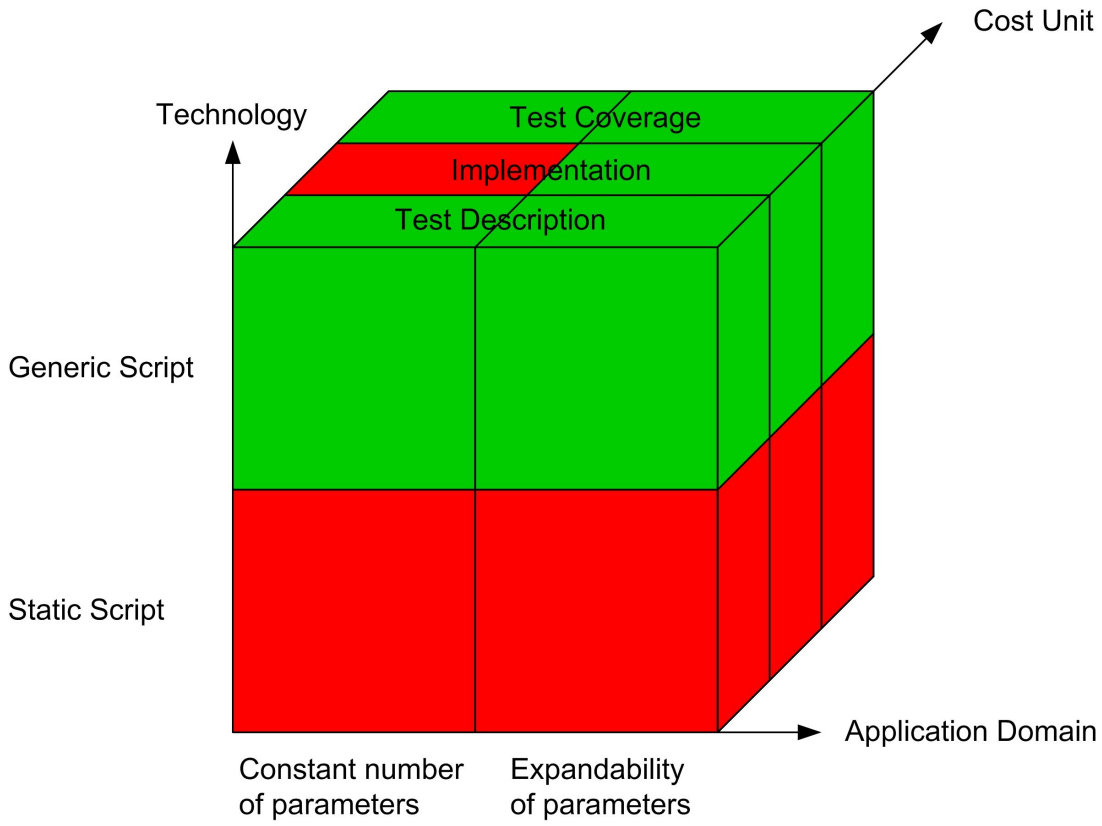


Figure 5.1: Results of the evaluation

The following table 5.1 shows the information of the result cube in a more structured way with the specific numerical values as result of the evaluation. However, important requirements such as usability and the risk of making mistakes during the configuration phase of the test case generation process are not included in the performance metric since it is difficult to measure these requirements. Nevertheless, the dynamic generic approach ensures that the parameter setting is valid and consistent by using a dynamic GUI at runtime. In addition, a mistake during the modification of the ontology would be noticed immediately after the parameter setting form is displayed in the GUI. Therefore a failure caused by wrong modification of the data model will be detected in an early stage before the generation of the test cases takes place. In contrary, for the static specific approach mistakes during the addition of new parameters as well as the configuration of the test case generation process will usually be detected after the simulation is finished. Furthermore, the user has to work on code level without a validation check and a consistency check of the parameter setting. In order to add a new test case parameter the user also has to modify the XML structure of the XML file to be generated on code level. This circumstance increases the testing efforts significantly. As a result, the risk of making mistakes in the static specific script is higher than the risk of making mistakes in the dynamic generic script.

	Dynamic Generic Script		Static Specific Script	
	Constant Parameters	Expandability of Parameters	Constant Parameters	Expandability of Parameters
Test Description	high-level	high-level	low-level	low-level
Implementation	260%	100%	100%	250%
Test Coverage	is definable	is definable	is not definable	is not definable

Table 5.1: Results of the evaluation

5.2.2 Evaluation of the static specific and the dynamic generic approach

This section focuses on making the evaluation concept transparent. Therefore, the criteria for the different objectives are explained in detail. The objectives are the costs for the test description, the effort for implementing test case parameters, and if the test coverage is definable by the user. The evaluation compares the static specific approach and the dynamic generic approach. Both approaches are test case generator scripts which aim at providing test cases as input data for a simulation in an automated and structured way.

Firstly, the evaluation concept determines the costs for the test description as configuration time of the parameter setting. Secondly, the effort for implementing test case parameters is determined with respect to the experience level. Lastly, the Return on Investment (ROI) is calculated assuming experience in the use of the different technologies. All results are based on the opinions of domain experts.

Figure 5.2 shows the costs for the test description for both generator scripts with respect to the implemented number of parameters. Therefore, the time to configure the test case generation process is measured.

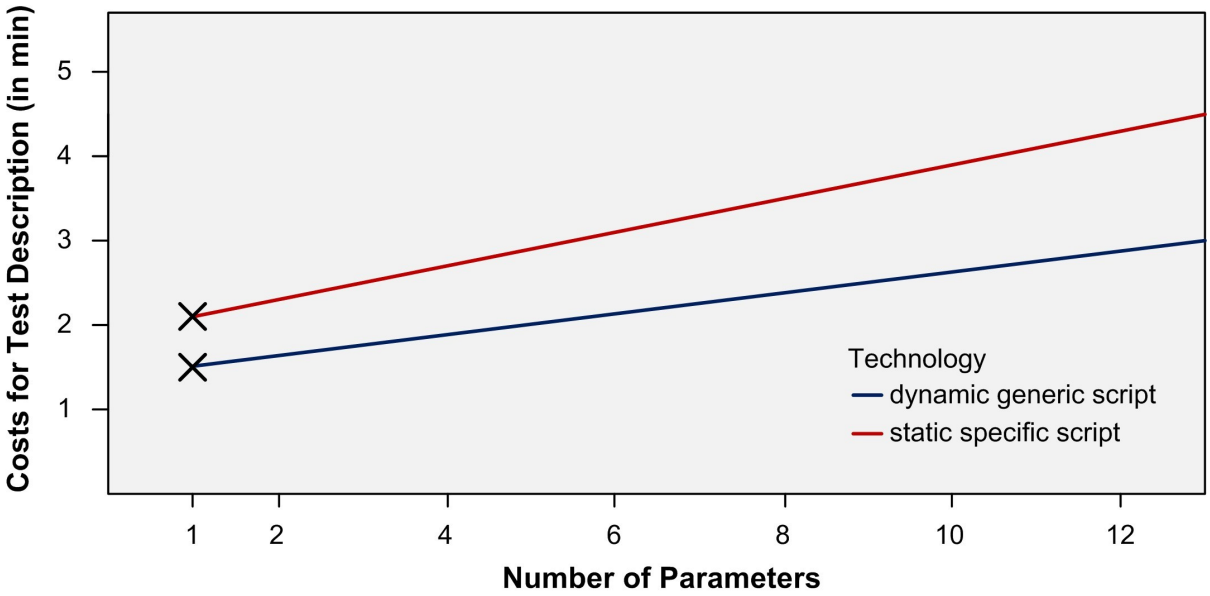


Figure 5.2: Costs for the test description

The evaluation concept for the implementation effort measures the time for implementing new test case parameters. In addition, the evaluation concept takes the learning curve of using new technologies into account (see figure 5.6). Therefore, two clusters were identified during the evaluation. On the one hand, figure 5.3 compares the effort in man-months to implement the static specific script and the dynamic generic script without assuming any experience. On the other hand, figure 5.4 shows the implementation effort of the generator scripts after the user has gained experience in the corresponding technology.

The graphs shown in the figures 5.3 and 5.4 display the effort for setting up the generator approach for the first test case parameter as well as the effort for adding further test case parameters. The Return on Investment (ROI) is reached after three implemented test case parameters with respect to the requirement of expandability. In this case the effort

for setting up the generator script is not taken into account. However, the ROI with respect to the overall effort is shown in figure 5.5.

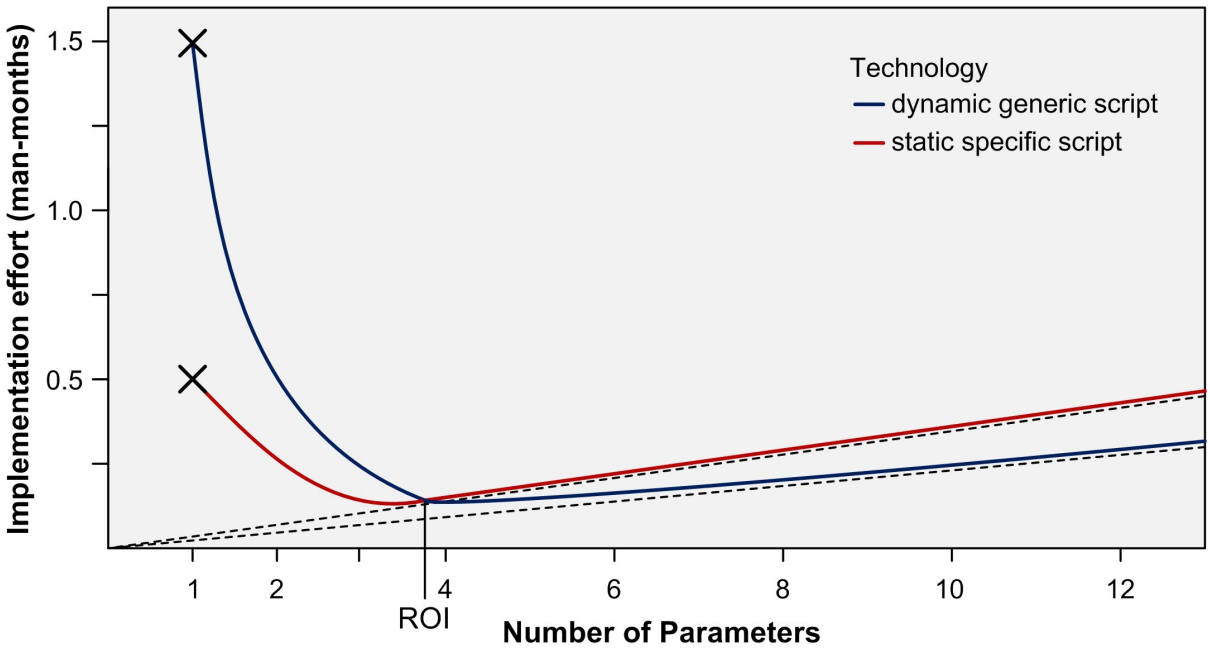


Figure 5.3: Costs for adding a new parameter (without domain experience)

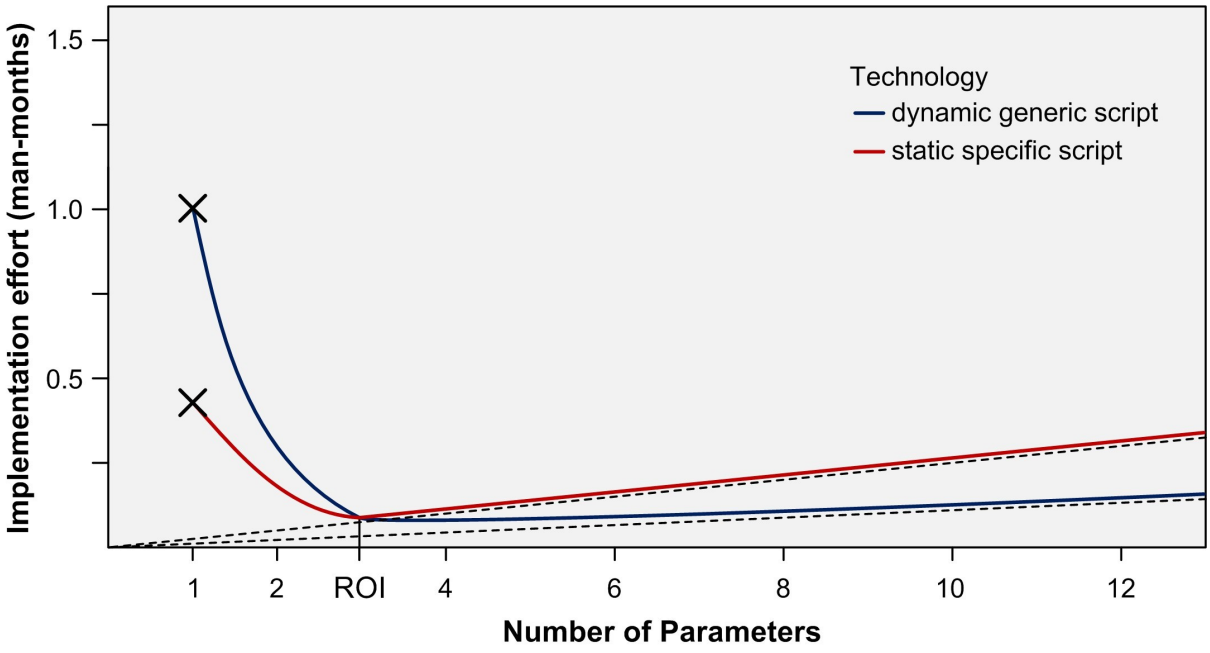


Figure 5.4: Costs for adding a new parameter (with domain experience)

Figure 5.5 outlines the effort for implementing up to 195 test case parameters. The calculated ROI has a value of 37.59 and is visualized in the figure. As a result, the higher

effort for the first time implementation of the dynamic generic script pays off after the implementation of the 38th test case parameter. The calculation is done by determining the number of implemented parameters for the static specific script to reach the effort for implementing the dynamic generic script for the first time. Therefore, the linear equation `y = k*x` of the static specific script has to be solved for `k = 1.6` and `y = 1MM`.

Nevertheless, the mentioned criteria in section 5.2.1 such as a lower risk for mistakes during the configuration phase of the generation process and ensuring valid and consistent test cases as output of the generation process make the use of the dynamic generic script preferable even for a small number of test case parameters.

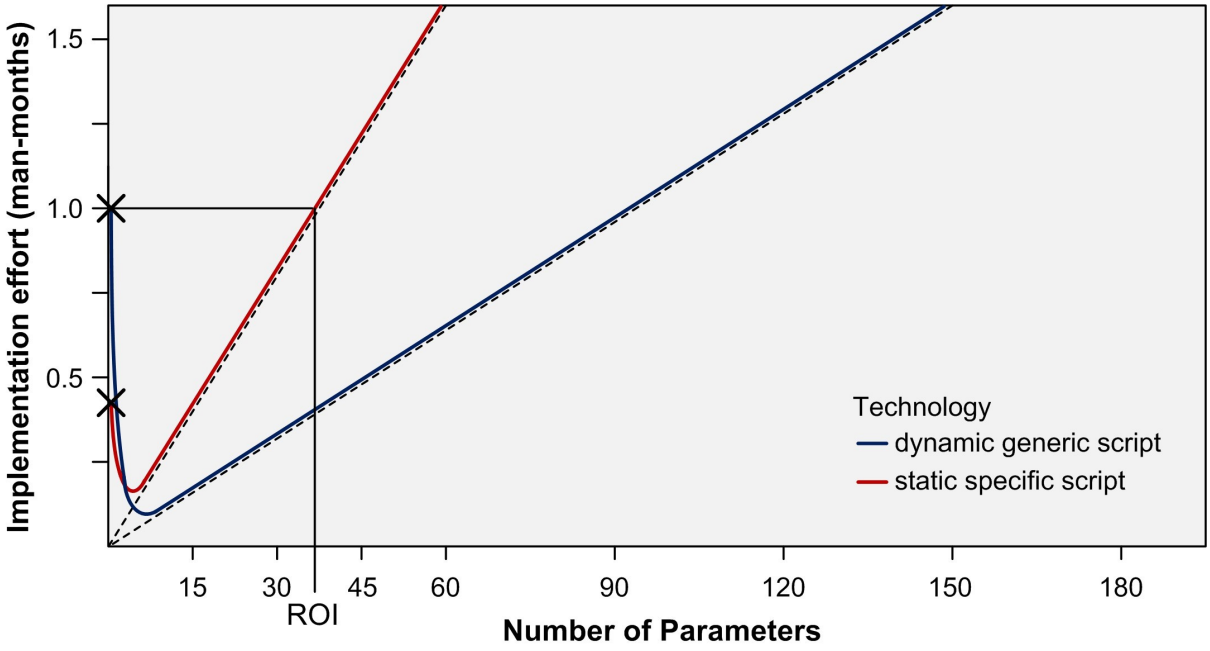


Figure 5.5: Costs for adding a high number of parameters (with domain experience)

Figure 5.6 shows the efficiency's typical schematic course of the new technology over time. New technologies do not only bring the suggested benefits. Missing experience can make the use of new technologies very time consuming especially if little documentation is available. Two new technologies were used for the implementation of the *"3 Phase Process Model"* shown in figure 4.10: the Jena framework for parsing the ontology and the Spring Rich Client Platform for building the dynamic GUI at runtime. However, not only the lower efficiency caused by the fact that the technologies were used for the first time, but also the dependencies of the frameworks presented a challenge which has to be dealt with.

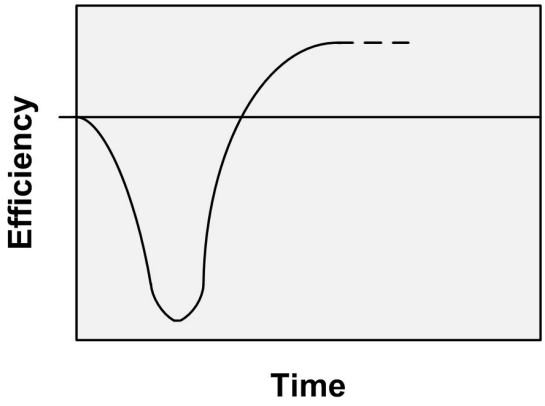


Figure 5.6: Learning curve for using a new technology (according to [21])

5.3 Composition of a test process for SAW

The developed test process for SAW shows a way to improve the quality of the simulation process (see figure 4.8). The test process is based on the "fundamental test process" developed by ISTQB and the "life cycle of a test case" explained in [41] (see section 2.1.1).

All five activities of the fundamental test process shown in figure 2.5 are supported by the developed test process for SAW. Firstly, the `planning and control` activity specifies the events of interest defined by the user as objectives of testing. The control activity analyzes the events logged by the simulator and informs the other activities in case of identified deviations. The control activity affects all other activities and is present throughout the whole testing process. Secondly, the `analysis and design` activity defines the simulation tool as testing tool for the assembly line under test. In addition, the input data for testing such as the test cases generated by the test case generator are specified and verified on the basis of identified conditions. Typical conditions are, for instance, that the test cases are related to the assembly line and that the test cases' parameters are implemented in the simulation tool. Afterwards, the `implementation and execution` activity verifies the information needed by the simulator to run the test cases (see figure 4.7). The `evaluation and report` activity proves if the postconditions as exit criteria are met. The `completion` activity closes the test process and writes the simulation results into an XML file.

Chapter 6

Conclusion

A high test coverage is essential for testing the performance of simulation systems not only in the production automation domain. Test case generators provide such test cases as input data for the simulation in an automatic and structured way. Many solutions for test case generators use a static specific approach which is difficult to expand. Mostly, these solutions offer an unsatisfying usability since only a low-level test description is supported. The thesis discussed the test case generation process and developed a dynamic generic approach based on an available static specific one to meet the criteria of a high-level test description, lower effort for implementing additional test case parameters, and a definable test coverage.
Firstly, the feasibility of the dynamic generic approach is proven by the implementation of a prototype during the elaboration of the thesis. Secondly, the cost-saving potential for the ontology-based approach is determined by the performance evaluation. The third research issue deals with the composition of a test process for the SAW project. For this purpose, a standardized test process is used to design a specific test process for the simulation tool.

The thesis identified weaknesses of the given static specific approach and found a solution to face these weaknesses. After the implementation of the new approach had taken place the effectiveness of both approaches was compared. The two generator scripts were evaluated with respect to three objectives – the test description, implementation, and the test coverage. In addition, two different application domains, i.e. a "constant number of parameters" and the "expandability of parameters" were taken into account. The dynamic generic approach, however, performs very well on most objectives as the result of the evaluation in chapter 5.2.2 shows. It could be shown that the new approach only lacks the implementation objective in the "constant number of parameters" domain. The

reason for this limitation is based on the higher effort for the first establishment of the *"3 Phases Process Model"* as shown in figure 4.10. Once the dynamic generic script is running it does not have to be maintained anymore even if the number of test case parameters varies over time. The next paragraph explains where the complexity for adding test case parameters is moved to and which other skills are required instead of programming skills to fulfill this task. A further advantage of the realization of the *"3 Phases Process Model"* is that the first phase and the second phase are totally independent of the domain as the domain specification is hidden in the ontology.

The simulation software is continually being improved by students. Thus, one of the most important features of the test case generator script is the expandability to support these changes. This is essential because the generated test cases are the input data for the simulation. In other words, beside the assembly line the test cases are the only input data by means of which the user can configure the simulation process. One of the strengths of the dynamic generic script is its flexibility as modifications to the ontology do not necessarily lead to manual changes to the GUI or to the dynamic generic script. This feature allows adding new parameters in the ontology with tool support. Afterwards the modification is represented by the GUI without any line of code. Therefore, the user needs no programming skills neither for adding new parameters nor for using them to configure the generation process. Most of the complexity is moved from the user to the implementation of the dynamic generic approach. This is why the whole test case generation process could be simplified from the user's perspective. Firstly, the dynamic generic approach uses the underlying ontology's structure and restrictions to ensure the consistency of the test cases. Secondly, the data ranges of the offered parameters are used to ensure the validation of the test cases. Nevertheless, the user has to manage the underlying ontology. Therefore, the user needs skills for modifying the ontology with common graphical editor tools such as Protégé. Of course, users need some experience before they can modify an ontology but it is less effort than modifying a hard-coded script. Furthermore, if something goes wrong during the modification of the ontology the user will recognize it immediately in the parameter setting configuration process. On the contrary, the user might notice failures during the modification of the static specific script after the simulation run took place by analyzing the simulation results. This is a frustrating experience as you usually do not know what went wrong. In this case the user is captured in a trial and error loop. Surely, these circumstances negatively affect the acceptability of the static specific approach.

The current dynamic generic approach can be extended in several ways. Two steps of the developed cyclic process for generating test cases shown in figure 4.4 are not implemented yet. The feasibility study in section 4.3.4 presents ways of implementing those steps. The

first of these two steps ensures that only parameters which are actually implemented are in the test cases file by synchronizing the generated XML file and the XML schema definition of the simulator. At the moment the user will be informed about the existence of a test case parameter which has not been implemented immediately after the simulation tries to start. The second step stores the achieved results of the simulation as feedback into the ontology. This feedback can be used as experience for following simulations. This information can be used as experience even if, for instance, the names of the docking stations in an assembly line are different. Therefore, a reasoner can build inferences of the captured knowledge in the ontology. This method enables the identification of concurrent docking stations of different assembly lines by comparing their machine functions instead of machine names. Another way to increase the quality of the simulation process is to implement the process of the simulator outlined in figure 4.8.
The developed dynamic GUI is an essential part of the test case generation process. At the moment the dynamic GUI is a prototype which helps to show the benefits of the ontology-based dynamic generic approach. The main disadvantage of the GUI is that the number of dynamically presentable test case parameters is limited to the static structure of the implementation. More information and a way to face this disadvantage can be found in section 4.3.4.4. The parameter settings could be stored in the ontology and therefore be offered as parameter setting pool to the user in the GUI. Thus, the user could choose from predefined parameter settings and modify if necessary. The programming methodology to calculate the test coverage assumes a fixed number of parameters and does not support data ranges for the test case parameters' values.

In conclusion the following future works can be identified:

- Realization of step 6 and 9 of the cyclic test case generation process:
  The sixth step synchronizes the generated XML file and the XML schema definition of the simulator to ensure that the XML file is structured well. The ninth step writes the results of the simulation to the ontology and makes them available for further iterations of the cyclic test case generation process.

- Realization of the conceptual test process shown in figure 4.8:
  The test process increases the quality of the simulation process since preconditions and postconditions can be generated with respect to the simulation input data. Thus, the conditions can be checked to ensure the quality of the simulation.

- Deduction of test cases by using reasoning techniques:
  At the moment the ontology is used for building a dynamic GUI at runtime to ensure that the parameter setting is valid and consistent. In addition, the structure of the XML file corresponds to the structure of the ontology. However, the deduction of test cases is probably more efficient than generating test cases as combinatorial

possibilities of the parameter setting especially since the ontology is a knowledge-based system and therefore enables to infer from the stored fact base.

- Improvement of the implemented prototype of the dynamic GUI:
  The code reflection technique is used to inspect and invoke the getter and setter methods of the prototype at runtime. The standard reflection API of Java does not support the alteration of program behavior. Thus, a possibility of improvement would be to use the structural reflection which is necessary to create the number of getter and setter methods with respect to the offered number of parameters by the ontology at runtime.

The feasibility section 4.3.4 shows ways to face the listed future work challenges.

Bibliography

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Building, Constructions*. Oxford University Press, August 1978.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, January 2008.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, second edition, April 2003.

[4] C. Beierle and G. Kern-Isberner. *Methoden wissensbasierter Systeme. Grundlagen, Algorithmen, Anwendungen*. Vieweg, third edition, 2006.

[5] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. Jade a white paper. 3, September 2003. `http://jade.tilab.com/papers/2003/WhitePaperJADEEXP.pdf`, accessed on 05.10.2009.

[6] C. Beust and H. Suleiman. *Next Generation Java Testing. TestNG and Advanced Concepts*. Addision Wesley, 2007.

[7] A. Bond and L. Gasser. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, 1988.

[8] B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering. Using UML, Patterns, and Java*. Prentice Hall, July 2009.

[9] I. Burnstein. *Practical Software Testing: A Process-Oriented Approach*. Springer, 2003.

[10] C. Calero, F. Ruiz, and M. Piattini. *Ontologies for Software Engineering and Software Technology*. Springer, 2006.

[11] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the semantic web recommendations. December 2003. `http://www.hpl.hp.com/techreports/2003/HPL-2003-146.pdf`, accessed on 24.08.2009.

[12] E. Castillo, J. M. Gutiérrez, and A. S. Hadi. *Expert systems and probabilistic network models*. Springer, 1997.

[13] W. Cazzola, R. J. Stroud, and F. Tisato. *Reflection and Software Engineering*. Springer, 2000.

[14] T.-C. Chiang and L.-C. Fu. Using dispatching rules for job shop scheduling with due date-based objectives. *IEEE International Conference on Robotics and Automation*, pages 1426 – 1431, May 2006.

[15] S. Chiba. *Load-Time Structural Reflection in Java*, volume 1850/2000. Springer, 2000.

[16] S. Cranefield and M. Purvis. Uml as an ontology modelling language. 1999. `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-23/cranefield-ijcai99-iii.pdf`, accessed on 11.03.2009.

[17] S. Dustdar, H. Gall, and M. Hauswirth. *Software-Architekturen für Verteilte Systeme*. Springer, 2003.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[19] E. Gregori, G. Anastasi, and S. Basaqni. *Advanced Lectures on Networking*. Springer, 2002.

[20] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Formal Ontology in Conceptual Analysis and Knowledge Representation*, August 1993. `http://www.itee.uq.edu.au/~infs3101/_Readings/OntoEng.pdf`, accessed on 25.06.2009.

[21] V. Gruhn, D. Pieper, and C. Röttgers. *MDA: Effektives Softwareengineering mit UML2 und Eclipse*. Springer, 2006.

[22] H. R. Hansen and G. Neumann. *Wirtschaftsinformatik I, Grundlagen betrieblicher Informationsverarbeitung*. UTB, eighth edition, 2001.

[23] K. Hitomi. *Manufacturing Systems Engineering: A Unified Approach to Manufacturing Technology, Production Management, and Industrial Economics*. Taylor & Francis, second edition, 1996.

[24] M. Hitz, G. Kappel, E. Kapsammer, and W. Retschitzegger. *UML @ Work. Objektorientierte Modellierung mit UML 2*. dpunkt.verlag, third edition, 2005.

[25] ISTQB. Foundation level syllabus. April 2007. `http://www.istqb.org/downloads/syllabi/SyllabusFoundation.pdf`, accessed on 12.04.2009.

[26] K. C. Jeong and Y. D. Kim. A real-time scheduling mechanism for a flexible manufacturing system: using simulation and dispatching rules. *INT. J. PROD. RES.*, 36:2609–2626, 1998.

[27] W. Karwowski and G. Salvendy. *Organization and Management of Advanced Manufacturing: A Human Factors Perspective*. John Wiley & Sons, 1994.

[28] E. Kutanoglu and I. Sabuncuoglu. Routing-based reactive scheduling policies for machine failures in dynamic job shops. *International Journal of Production Research*, 39:3141–3158, 2001.

[29] E. H. Van Leeuwen and D. Norrie. Holons and holarchies. *Manufacturing Engineer*, 76:86–88, April 1997.

[30] D. L. McGuinness and F. Harmelen. Owl web ontology language. February 2004. `http://www.w3.org/TR/owl-features/`, accessed on 21.07.2009.

[31] M. Merdan, T. Moser, D. Wahyoudin, S. Biffl, and P. Vrba. Simulation of workflow scheduling strategies using the mast test management system. *Control, Automation, Robotics and Vision, 2008. ICARCV 2008. 10th International Conference*, pages 1172–1177, December 2008.

[32] M. Merdan, T. Moser, D. Wahyudin, and S. Biffl. Performance evaluation of workflow scheduling strategies - considering transportation times and conveyor failures. *Industrial Engineering and Engineering Management, 2008. IEEM 2008. IEEE International Conference*, pages 389–394, December 2008.

[33] N. Meyrowitz. *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, volume 22. Association for Computing Machinery, October 1989. 1987, Orlando, Florida.

[34] E. Miller. An introduction to the resource description framework. *D-Lib Magazine*, May 1998.

[35] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3), 1991.

[36] P. Niemeyer and J. Knudsen. *Learning Java*. O'Reilly, third edition, 2005.

[37] OMG. Introduction to omg's unified modeling language. `http://www.omg.org/gettingstarted/what_is_uml.htm`, accessed on 02.07.2009, 2009.

[38] C. Parsons. Spring richclient: A journey. February 2006. `http://pa.rsons.org/node/6`, accessed on 16.04.2009.

[39] Associated Press. Software disasters are often people problems. 2004. `http://www.msnbc.msn.com/id/6174622`, accessed on 02.06.2009.

[40] W. E. Rossig and J. Prätsch. *Wissenschaftliche Arbeiten*. PRINT-TEC Druck+Verlag, fifth edition, 2005.

[41] A. Spillner and T. Linz. *Basiswissen Softwaretesten*. dpunkt.verlag, third edition, 2005.

[42] S. St.Laurent and M. Fitzgerald. *XML kurz & gut*. O´Reilly, third edition, 2006.

[43] J. Tian. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. Wiley & Sons, 2005.

[44] P. Vrba. Mast: manufacturing agent simulation tool. *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference*, 1:282–287, September 2003.

[45] M. Wichmann. Javassist: Java - api zum Ändern von java bytecode. June 2003. `http://m2w2.de/articles/javassist_presentation.pdf`, accessed on 12.09.2009.

[46] S. M. Yacoub and H. H. Ammar. *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addsion-Wesley, 2004.

Appendix A

Screenshots of the test case generation application

The screenshots of this section present a typical life-cycle of the test case generation application.
Figure A.1 shows the *"3 Phase Process Model"* of the implemented prototype. The configuration file as output of the first phase is the basis artifact for building the dynamic GUI at runtime (see section B.2). After the application is started the initial page presents a brief instruction of how to use the application for generating test cases.

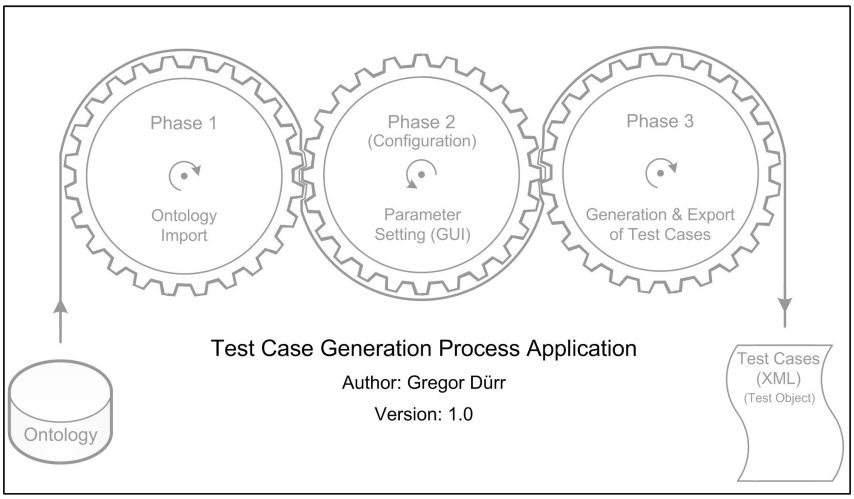


Figure A.1: Start-up screen of the application

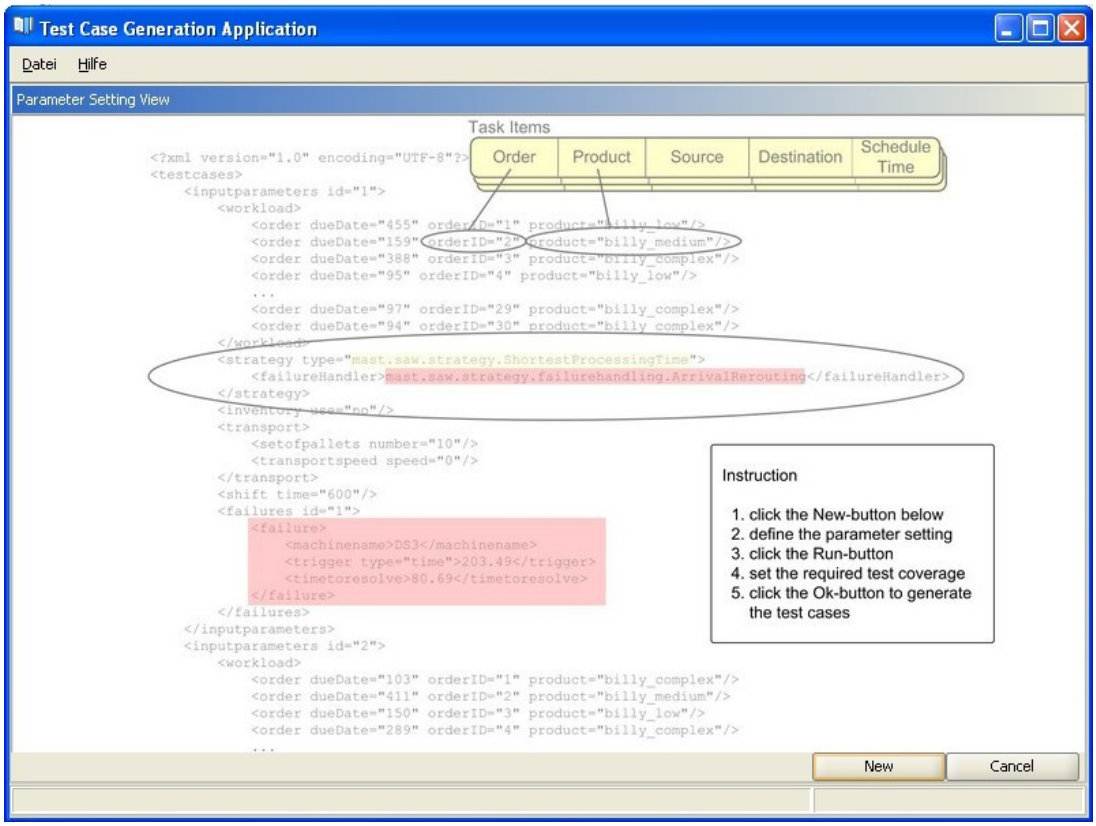


Figure A.2: Screenshot of the initial view

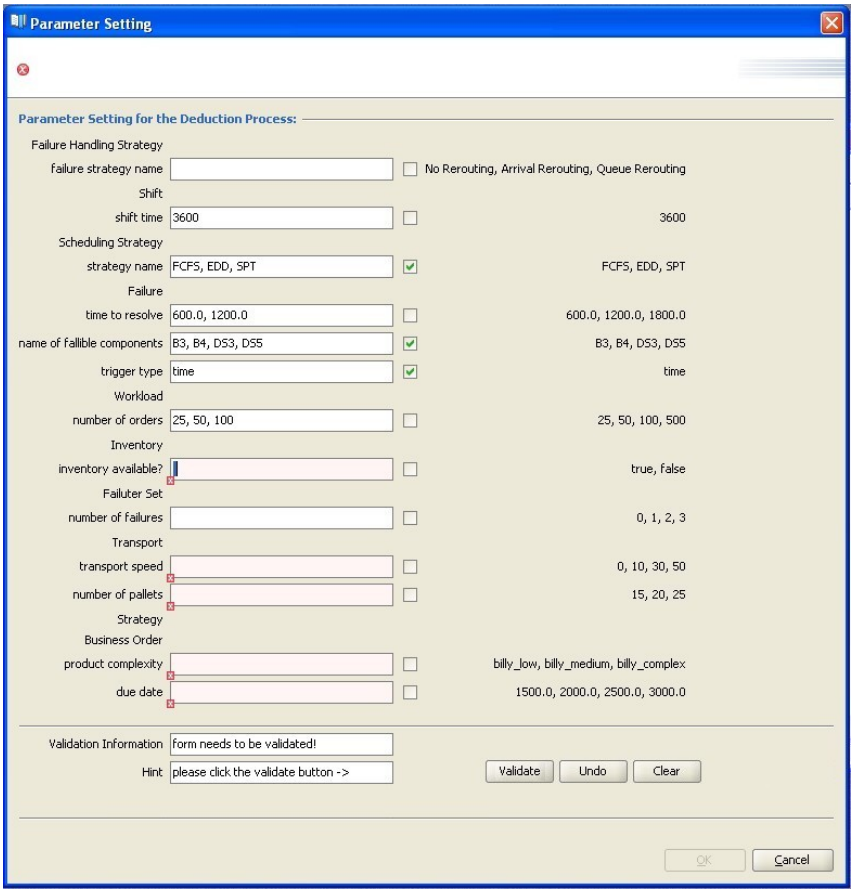


Figure A.3: Screenshot of the parameter setting form (incomplete)

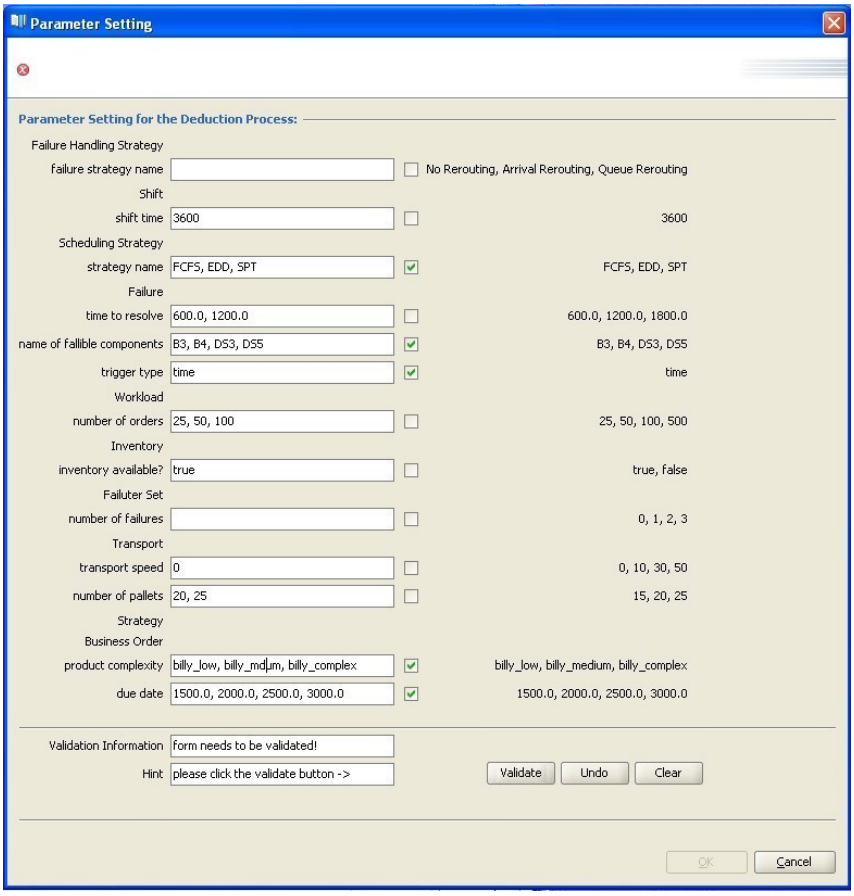


Figure A.4: Screenshot of the parameter setting form (completed with failure)

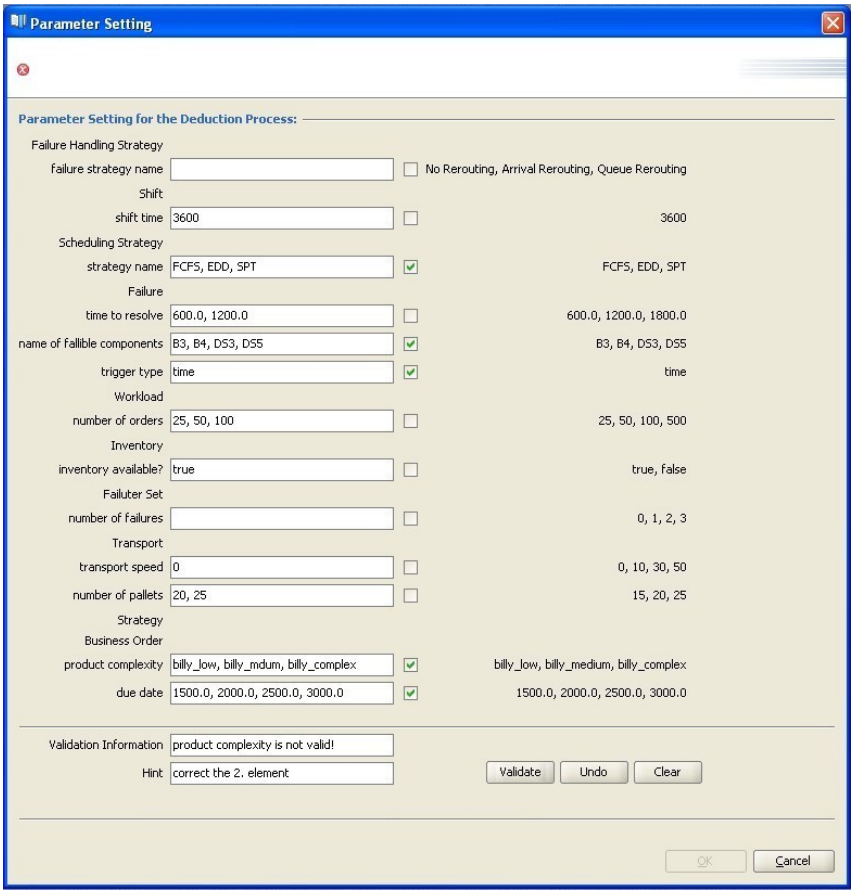


Figure A.5: Screenshot of the parameter setting form (validation info of the failure)

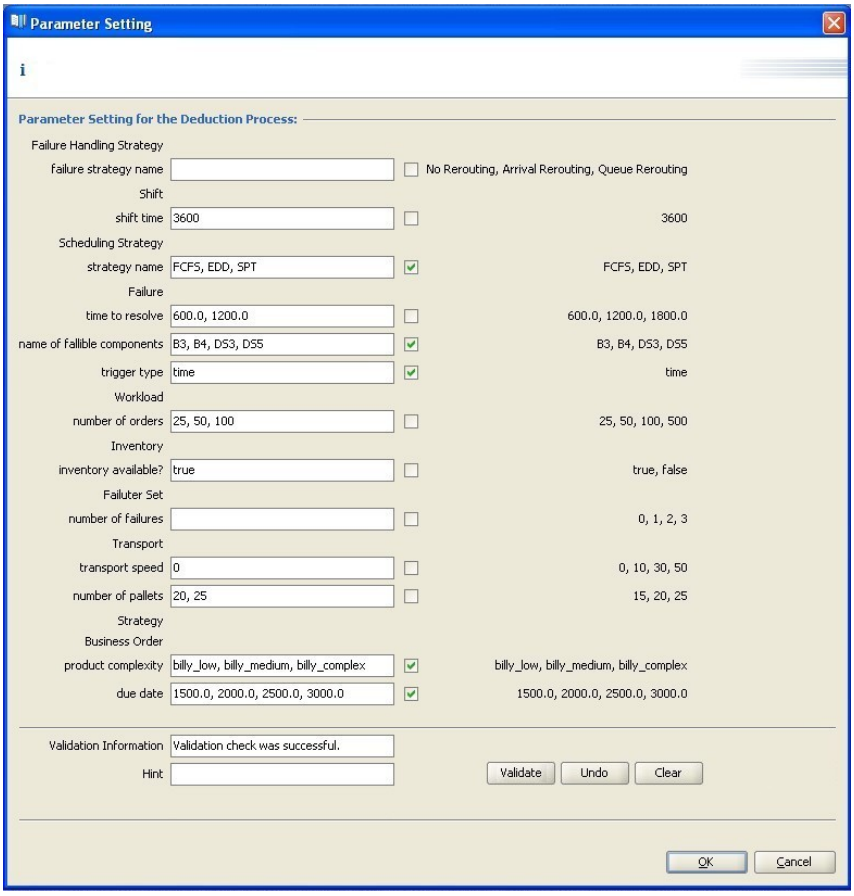


Figure A.6: Screenshot of the parameter setting form (valid and consistent)

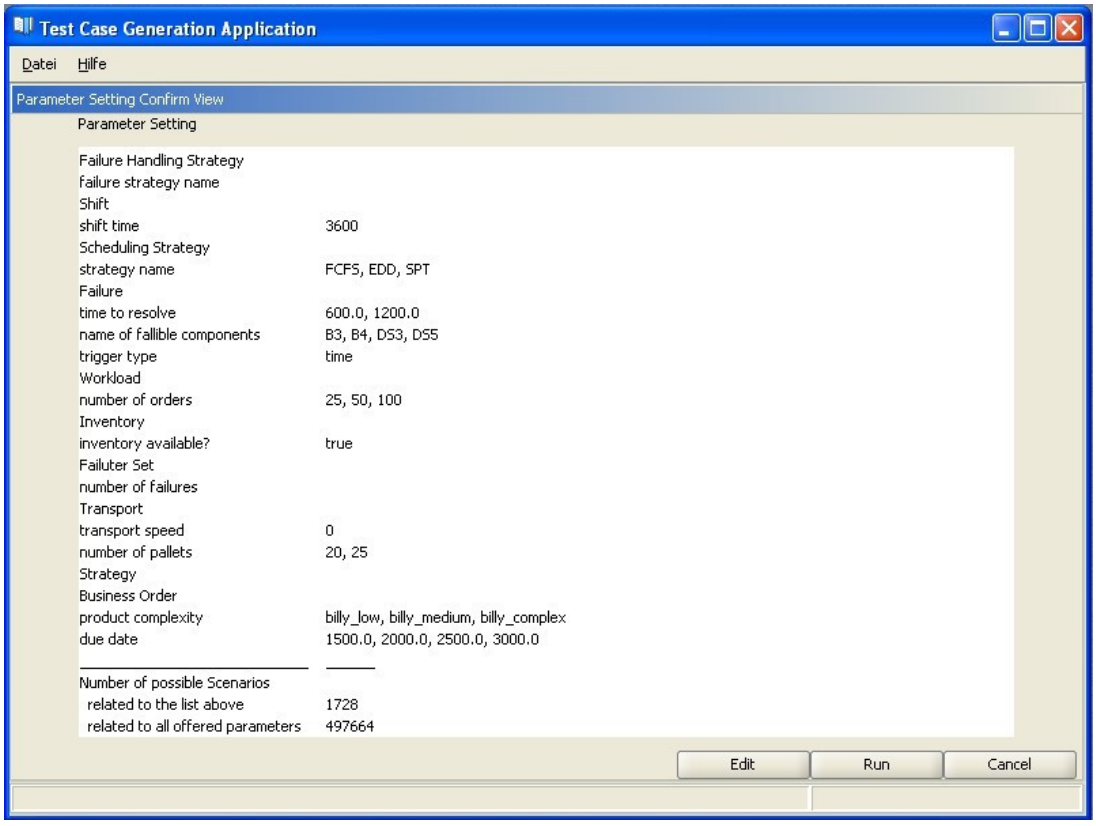


Figure A.7: Screenshot of the parameter setting confirmation view

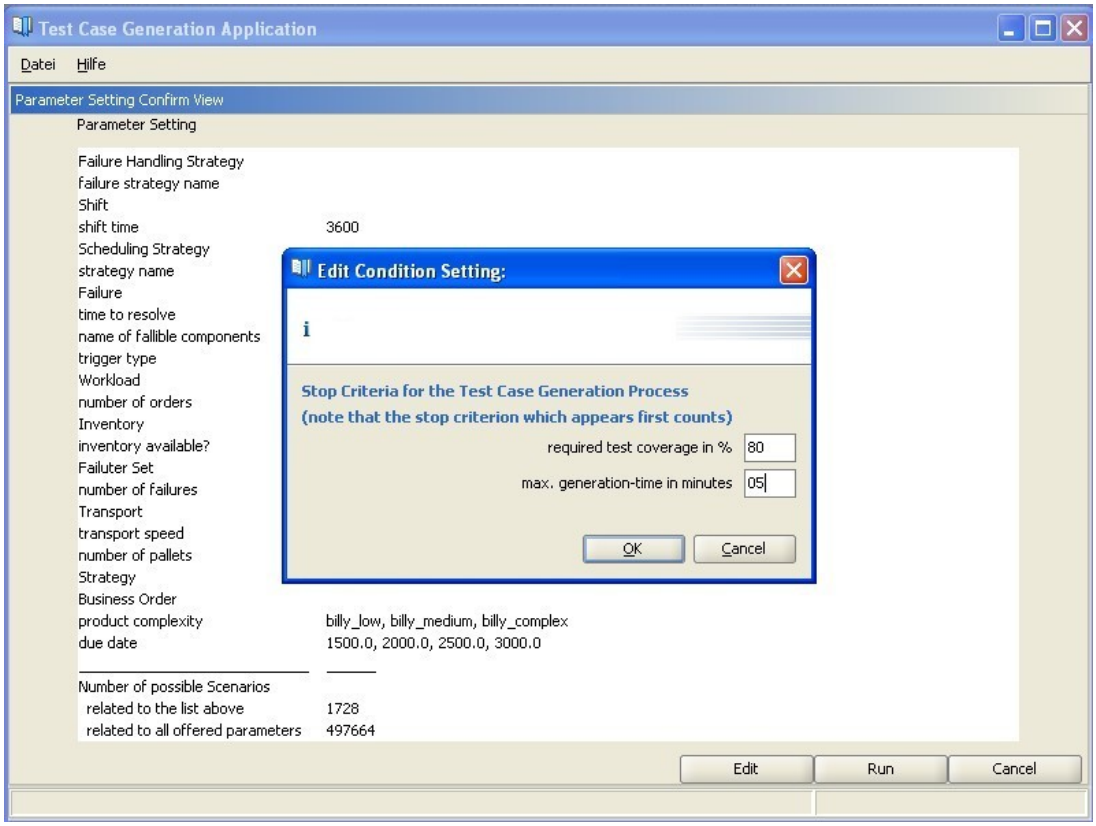


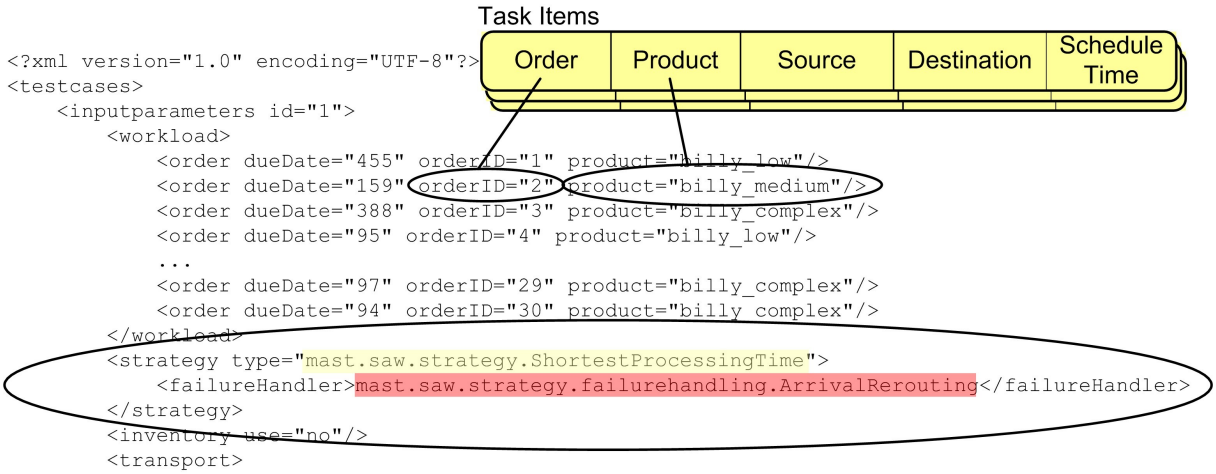
Figure A.8: Screenshot of the condition setting form

Appendix B

Software artifacts

This chapter provides a collection of software artifacts such as the XML structure and the XML schema definition of the XML file. In addition, the configuration file for the dynamic GUI is listed in section B.2.

B.1 Test case XML file structure




```
<?xml version="1.0" encoding="UTF-8"?>
<testcases>
    <inputparameters id="1">
        <workload>
            <order dueDate="455" orderID="1" product="billy_low"/>
            <order dueDate="159" orderID="2" product="billy_medium"/>
            <order dueDate="388" orderID="3" product="billy_complex"/>
            <order dueDate="95" orderID="4" product="billy_low"/>
            ...
            <order dueDate="97" orderID="29" product="billy_complex"/>
            <order dueDate="94" orderID="30" product="billy_complex"/>
        </workload>
        <strategy type="mast.saw.strategy.ShortestProcessingTime">
            <failureHandler>mast.saw.strategy.failurehandling.ArrivalRerouting</failureHandler>
        </strategy>
        <inventory use="no"/>
        <transport>
            <setofpallets number="10"/>
            <transportspeed speed="0"/>
        </transport>
        <shift time="600"/>
        <failures id="1">
            <failure>
                <machinename>DS3</machinename>
                <trigger type="time">203.49</trigger>
                <timetoresolve>80.69</timetoresolve>
            </failure>
        </failures>
    </inputparameters>
    <inputparameters id="2">
        <workload>
            <order dueDate="103" orderID="1" product="billy_complex"/>
            <order dueDate="411" orderID="2" product="billy_medium"/>
            <order dueDate="150" orderID="3" product="billy_low"/>
            <order dueDate="289" orderID="4" product="billy_complex"/>
            ...
        </workload>
        ...
    </inputparameters>
</testcases>
```

Figure B.1: Snippet of the test case XML file

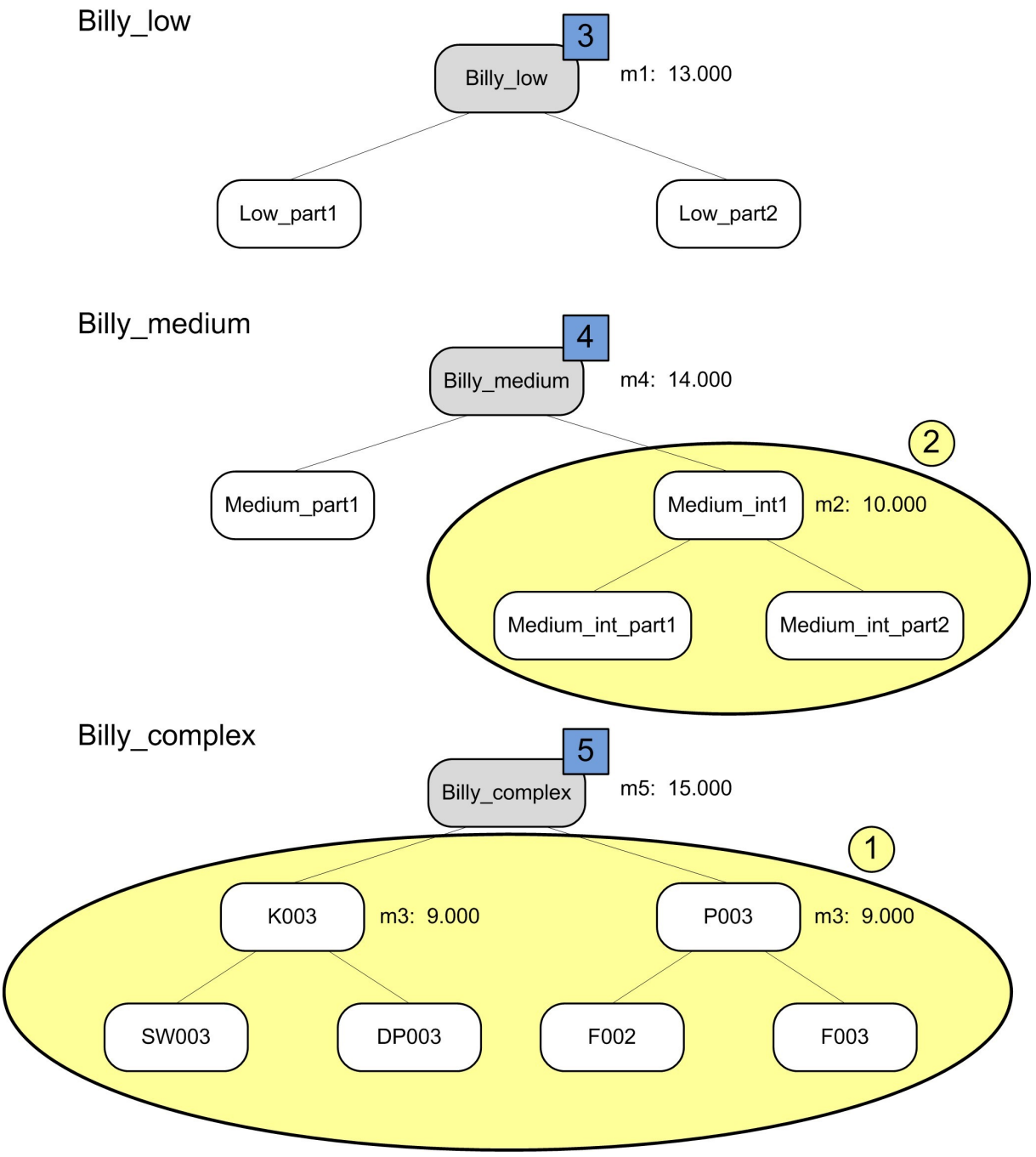


Figure B.2: Product tree of different products

B.2 Configuration file of the dynamic GUI

```
1      # General Messages for the Test Case Generation Application
2      # -> can be found in the messages.properties file of the application
3
4      # Specific Messages for the Test Case Generation Process
5
6      # Parameter Setting form and view
7      numberOfParams.label=25
8      param0.label=Shift
9      validParam0.label=isClass
10     param1.label=period_ time
11     validParam1.label=3600
12     param2.label=BusinessOrder
13     validParam2.label=isClass
14     param3.label=product
15     validParam3.label=billy_ low, billy_ medium, billy_ complex
16     param4.label=duedate
17     validParam4.label=1500.0, 2000.0, 2500.0, 3000.0
18     param5.label=Schedulingstrategy
19     validParam5.label=isClass
20     param6.label=identification
21     validParam6.label=FCFS, EDD, SPT
22     param7.label=priority
23     validParam7.label=null
24     param8.label=Workload
25     validParam8.label=isClass
26     param9.label=consist_ of_ bo
27     validParam9.label=null
28     param10.label=numberoforders
29     validParam10.label=25, 50, 100, 500
30     param11.label=Inventory
31     validParam11.label=isClass
32     param12.label=use
33     validParam12.label=true, false
34     param13.label=Failureset
35     validParam13.label=isClass
36     param14.label=componentname
37     validParam14.label=B3, B4, DS3, DS5
38     param15.label=timetoresolve
39     validParam15.label=600.0, 1200.0, 1800.0
40     param16.label=triggertime
41     validParam16.label=600.0, 900.0, 1200.0
42     param17.label=Transport
43     validParam17.label=isClass
44     param18.label=transportspeed
45     validParam18.label=0, 10, 30, 50
46     param19.label=numberofpallets
47     validParam19.label=15, 20, 25
48     param20.label=Strategy
49     validParam20.label=isClass
50     param21.label=identification
51     validParam21.label=FCFS, EDD, SPT
52     param22.label=priority
53     validParam22.label=null
54     param23.label=Failurehandlingstrategy
55     validParam23.label=isClass
56     param24.label=identification
57     validParam24.label=FCFS, EDD, SPT
58     paramSettingTitle.label=Parameter Setting
```

Code Fragment B.2: Configuration file of the dynamic GUI (output of the first phase)

B.3 Test case XML schema

```
<?xml version="1.0"encoding="UTF-8"?>
<!--W3C Schema generated by XMLSpy v2008 sp1 (http://www.altova.com)-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="testcases">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="inputparameters"maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="workload">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="order"
           maxOccurs="unbounded">
           <xs:complexType>
            <xs:attribute
             name="product"use="required"type="xs:string"/>
            <xs:attribute
             name="orderID"use="required"type="xs:integer"/>
            <xs:attribute
             name="dueDate"use="required"type="xs:integer"/>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="strategy">
        <xs:complexType>
         <xs:sequence>
          <xs:element
           name="failureHandler"type="xs:string"maxOccurs="1"
           minOccurs="0"/>
         </xs:sequence>
         <xs:attribute name="type"
          use="required"type="xs:string"/>
        </xs:complexType>
       </xs:element>
       <xs:element name="inventory">
        <xs:complexType>
         <xs:attribute name="use"
          use="required">
          <xs:simpleType>
           <xs:restriction
            base="xs:string">
            <xs:enumeration
             value="yes"/>
            <xs:enumeration
             value="no"/>
           </xs:restriction>
          </xs:simpleType>
         </xs:attribute>
        </xs:complexType>
       </xs:element>
       <xs:element name="transport">
        <xs:complexType>
         <xs:sequence>
          <xs:element
           name="setofpallets">
           <xs:complexType>
            <xs:attribute
             name="number"use="required"type="xs:integer"/>
           </xs:complexType>
          </xs:element>
          <xs:element
```

```
               name="transportspeed">
              <xs:complexType>
               <xs:attribute
                name="speed"use="required"type="xs:integer"/>
              </xs:complexType>
             </xs:element>
            </xs:sequence>
           </xs:complexType>
          </xs:element>
          <xs:element name="shift">
           <xs:complexType>
            <xs:attribute name="time"
             use="required"type="xs:integer"/>
           </xs:complexType>
          </xs:element>
          <xs:element name="failures"minOccurs="0"
           maxOccurs="1">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="failure"
              minOccurs="0"maxOccurs="unbounded">
              <xs:complexType>
               <xs:sequence>
                <xs:element
                 name="machinename"type="xs:string"/>
                <xs:element
                 name="trigger">
                 <xs:complexType>
                  <xs:simpleContent>
                  <xs:extension
                   base="xs:decimal">
                   <xs:attribute
                    name="type"use="required">
                    <xs:simpleType>
                     <xs:restriction
                      base="xs:string">
                      <xs:enumeration
                       value="time"/>
                      <xs:enumeration
                       value="workpieces"/>
                     </xs:restriction>
                    </xs:simpleType>
                   </xs:attribute>
                  </xs:extension>
                  </xs:simpleContent>
                 </xs:complexType>
               </xs:element>
               <xs:element
                name="timetoresolve"type="xs:decimal"minOccurs="0"
                maxOccurs="1"/>
              </xs:sequence>
             </xs:complexType>
            </xs:element>
           </xs:sequence>
           <xs:attribute name="id"
            type="xs:int"use="required"/>
           </xs:complexType>
          </xs:element>
          <xs:element name="simulation"maxOccurs="1"
           minOccurs="0">
           <xs:complexType>
            <xs:sequence>
             <xs:element
              name="simulationStep"minOccurs="0"maxOccurs="1">
              <xs:simpleType>
               <xs:restriction
                base="xs:int">
                <xs:minExclusive
```

```
                  value="1">
                 </xs:minExclusive>
                 <xs:maxExclusive
                  value="300">
                 </xs:maxExclusive>
                </xs:restriction>
               </xs:simpleType>
              </xs:element>
              <xs:element name="realtimeStep"
               minOccurs="0"maxOccurs="1">
               <xs:simpleType>
                <xs:annotation>
                 <xs:documentation>
                  real time in ms,
                  passed between
                  simulation steps
                  (0 = fastest
                  possible)
                 </xs:documentation>
                </xs:annotation>
                <xs:restriction
                 base="xs:int">
                 <xs:minExclusive
                  value="0">
                 </xs:minExclusive>
                </xs:restriction>
               </xs:simpleType>
              </xs:element>
              <xs:element name="framerate"
               minOccurs="0"maxOccurs="1">
               <xs:simpleType>
                <xs:annotation>
                 <xs:documentation>
                  number of frames
                  rendered every
                  second (system
                  time)
                 </xs:documentation>
                </xs:annotation>
                <xs:restriction
                 base="xs:int">
                 <xs:maxExclusive
                  value="1000">
                 </xs:maxExclusive>
                 <xs:minExclusive
                  value="1">
                 </xs:minExclusive>
                </xs:restriction>
               </xs:simpleType>
              </xs:element>
             </xs:sequence>
            </xs:complexType>
           </xs:element>
           <xs:element name="test"type="TestType"minOccurs="0"maxOccurs="1">
           </xs:element>
          </xs:sequence>
          <xs:attribute name="id"use="required"type="xs:integer"/>
         </xs:complexType>
        </xs:element>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
     <xs:complexType name="TestType">
      <xs:sequence>
       <xs:element name="trigger"minOccurs="1"maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="time"type="xs:double">
```

```
        </xs:element>
      </xs:sequence>
      <xs:attribute name="type"type="xs:string">
      </xs:attribute>
      <xs:attribute name="machineName"type="xs:string"/>
      <xs:attribute name="enable"type="xs:boolean"/>
    </xs:complexType>
   </xs:element>
   <xs:element name="testdata">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="param"maxOccurs="unbounded"minOccurs="1">
       <xs:complexType>
        <xs:attribute name="name"type="xs:string"/>
        <xs:attribute name="value"type="xs:string"/>
        <xs:attribute name="type"type="xs:string"/>
       </xs:complexType>
      </xs:element>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="description"type="xs:string"minOccurs="0"maxOccurs="1">
  </xs:sequence>
  <xs:attribute name="testcaseId"type="xs:string"/>
  <xs:attribute name="type"type="xs:string"/>
 </xs:complexType>
 <xs:complexType name="TestDataSet">
  <xs:sequence>
   <xs:element name="param">
    <xs:complexType>
     <xs:attribute name="name"type="xs:string">
     </xs:attribute>
     <xs:attribute name="value"type="xs:string"/>
     <xs:attribute name="type"type="xs:string"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Code Fragment B.3: XML schema of a test case (testcases-1.4.xsd)

Appendix C

Design of the ontology

This chapter is structured into three sections. The first section illustrates the relation between the dynamic GUI and the underlying ontology of the dynamic generic approach. In the second section, a short tutorial explains the addition of new test case parameters. The third section shows a solution how the test case layer of the elaboration part can be integrated into the ontology of the SAW project.

C.1 Process visualization of the dynamic generic approach

The following section shows the screenshots of how the different artifacts are related.

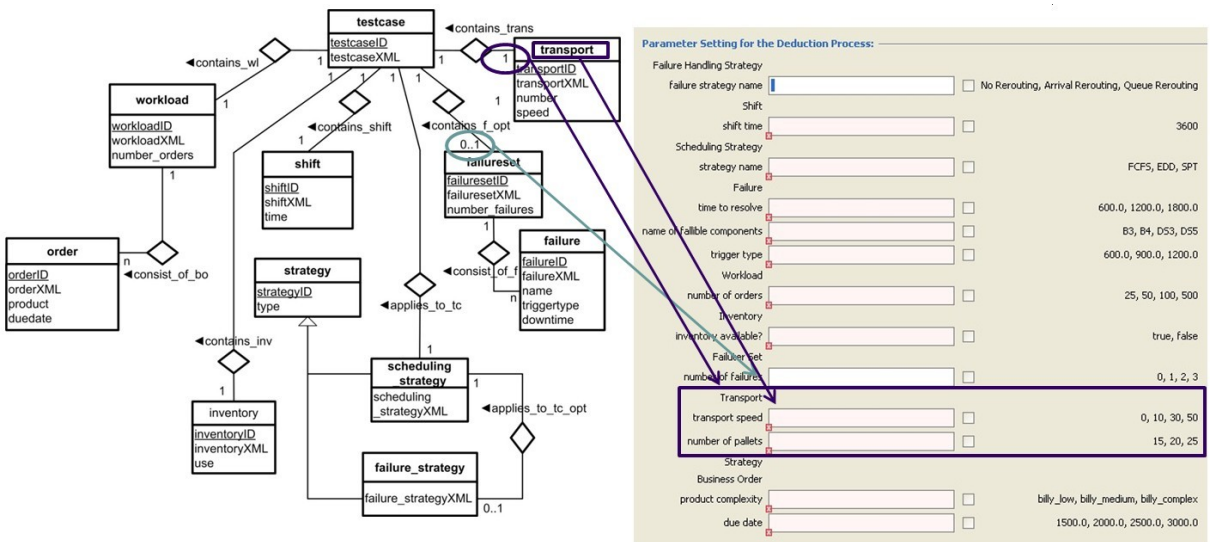


Figure C.1: Mapping EER diagram to GUI elements

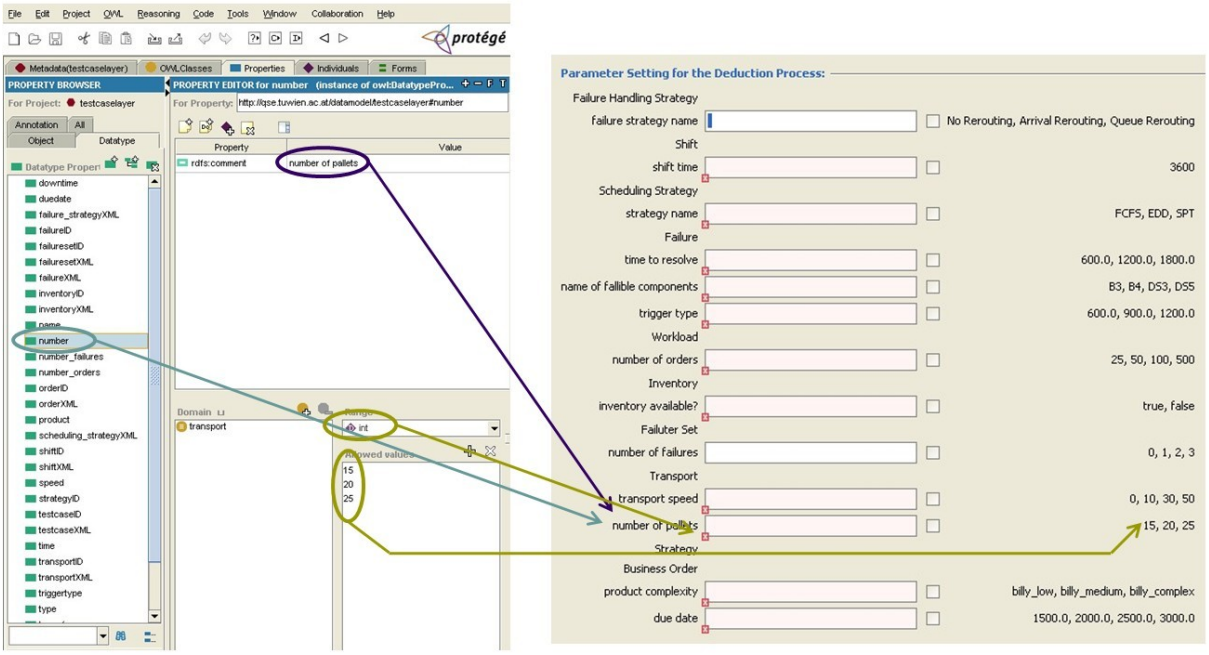


Figure C.2: Mapping Protégé view to GUI elements

C.2 Instruction for adding test case parameters

The failure_strategy entity of the already known EER diagram from the elaboration chapter are marked in figure C.3. The reason for this is that we start to enhance an existing ontology which is represented by the unmarked part of the EER diagram. Therefore no test case parameter for failure handling strategies exists at the beginning of the tutorial.

The challenge of the enhancement of the ontology is to add the following elements to the ontology:

- failure_strategy (entity)
- applies_to _tc _opt (relation)

The table 4.1 in the elaboration chapter informs that entities of the EER diagram are represented by OWL-Classes in the ontology. Furthermore, relations are represented by Object-Properties and attributes are represented by Datatype-Properties.

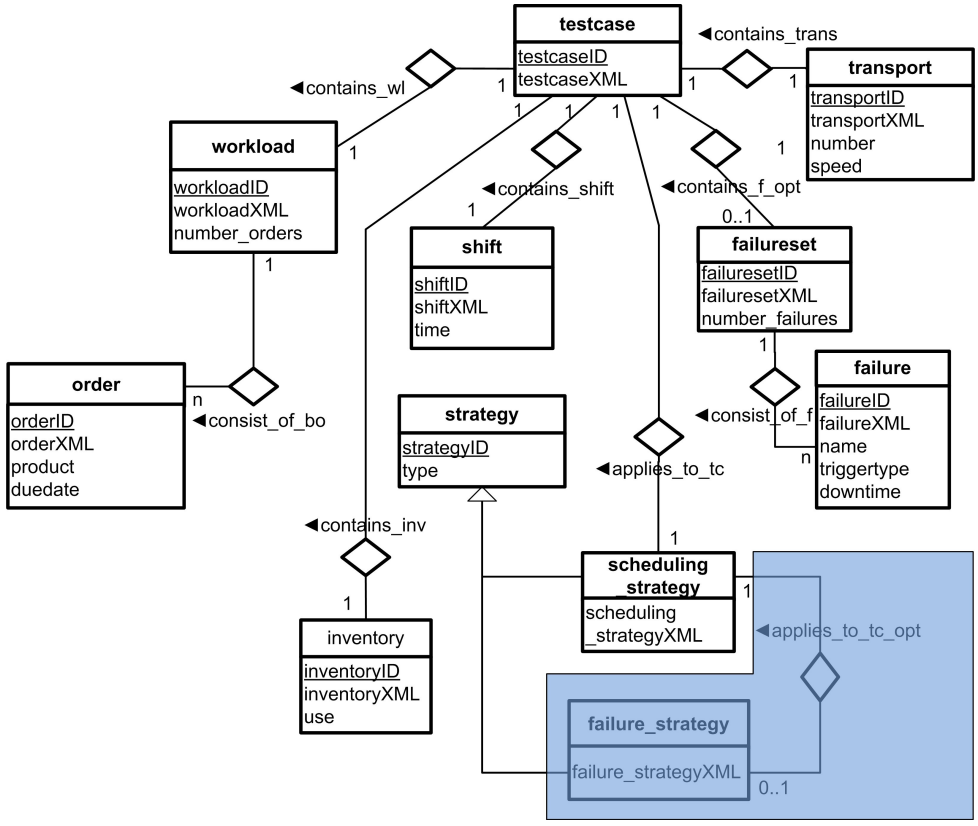


Figure C.3: Modified test case layer for enhancement tutorial (EER diagram)

1. Download and install the latest version of Protégé[1]
2. Start Protégé and open the existing ontology
3. Follow the step-by-step instruction (see screenshots)

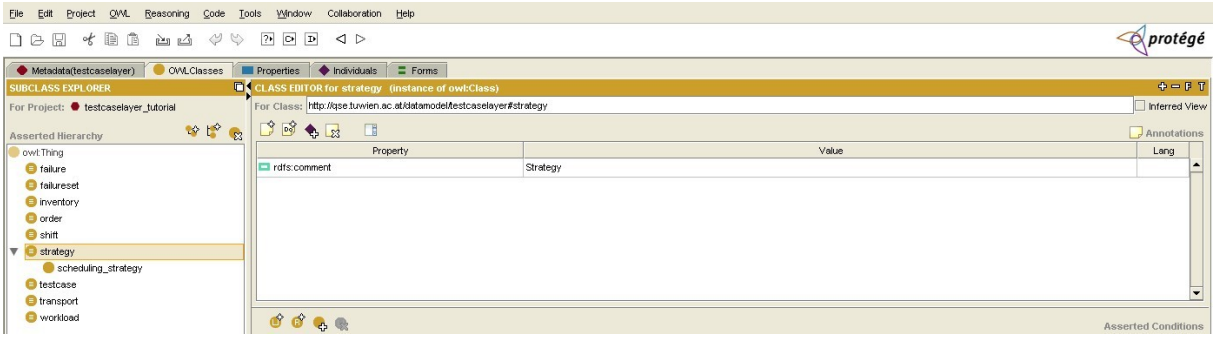


Figure C.4: Screenshot of the Protégé editor (OWL-Classes)

[1]http://protege.stanford.edu/download/download.html

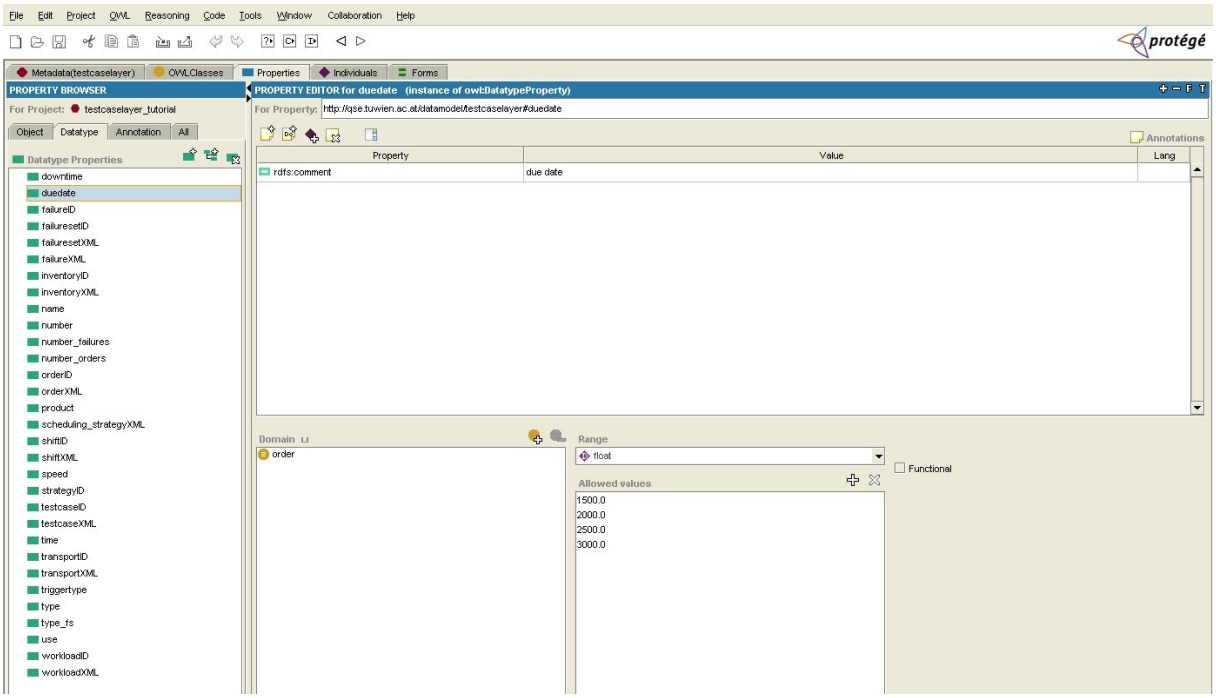


Figure C.5: Screenshot of the Protégé editor (Datatype-Properties)

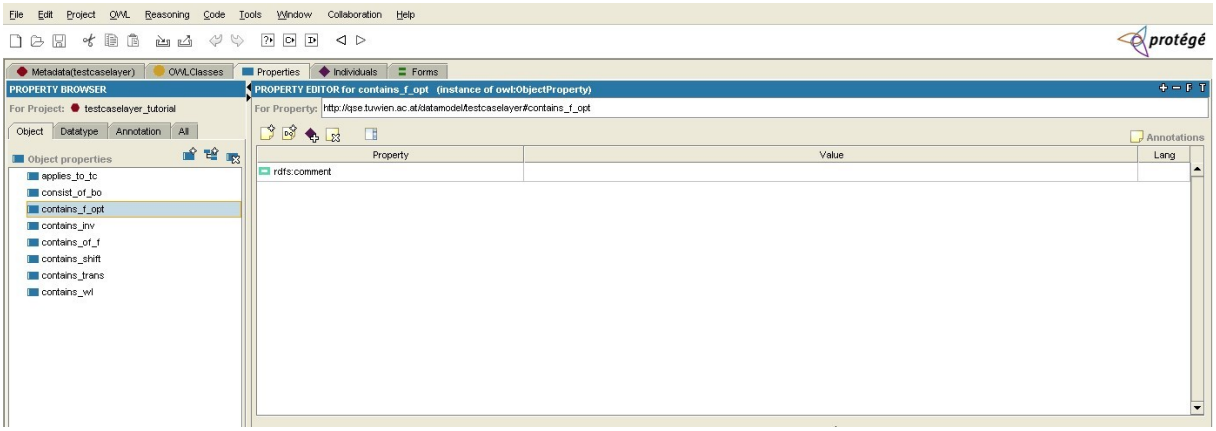


Figure C.6: Screenshot of the Protégé editor (Object-Properties)

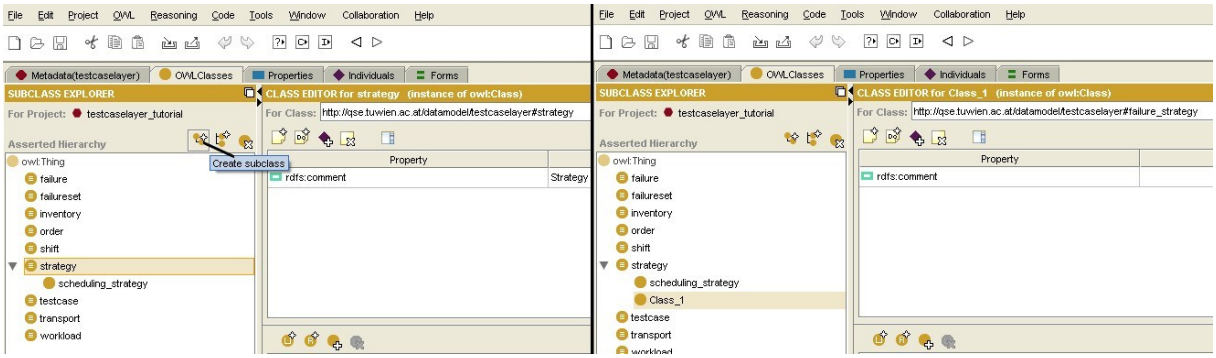


Figure C.7: Screenshot of the Protégé editor (new OWL-Class)

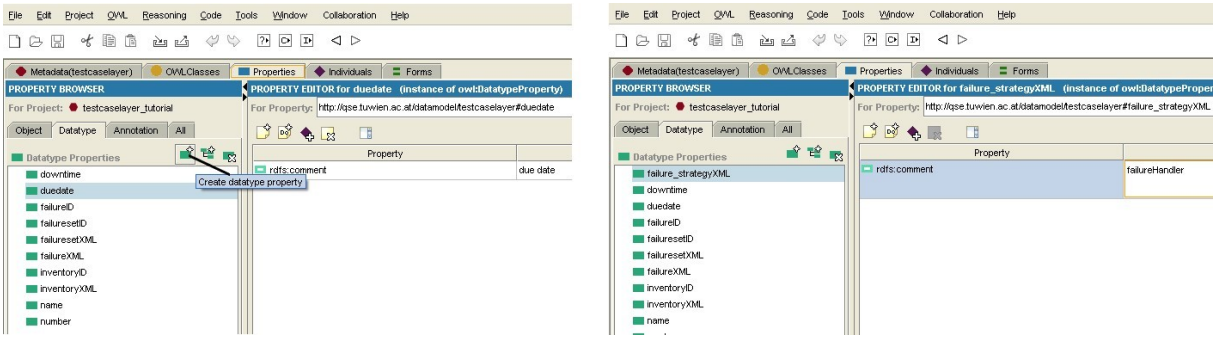


Figure C.8: Screenshot of the Protégé editor (new Datatype)

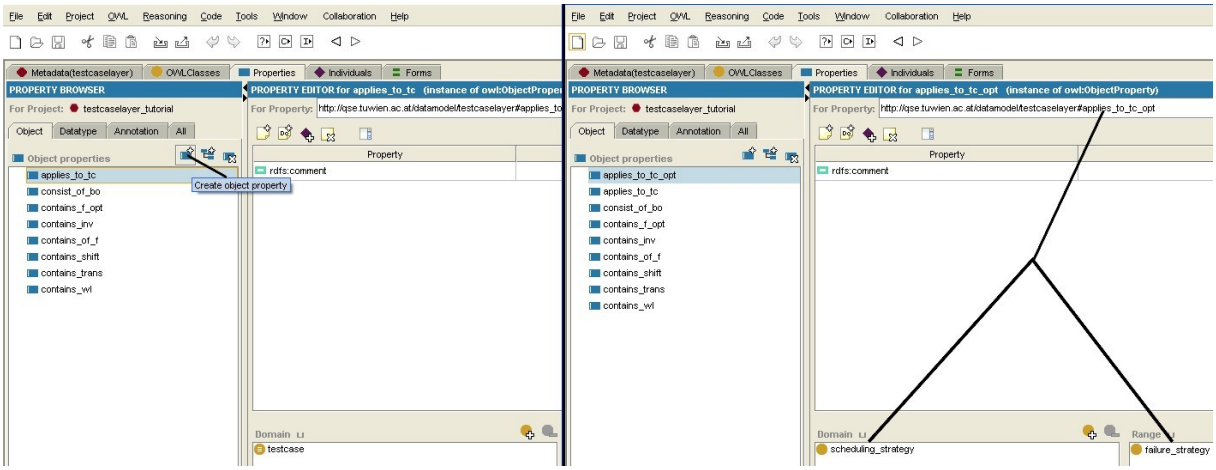


Figure C.9: Screenshot of the Protégé editor (new Object)

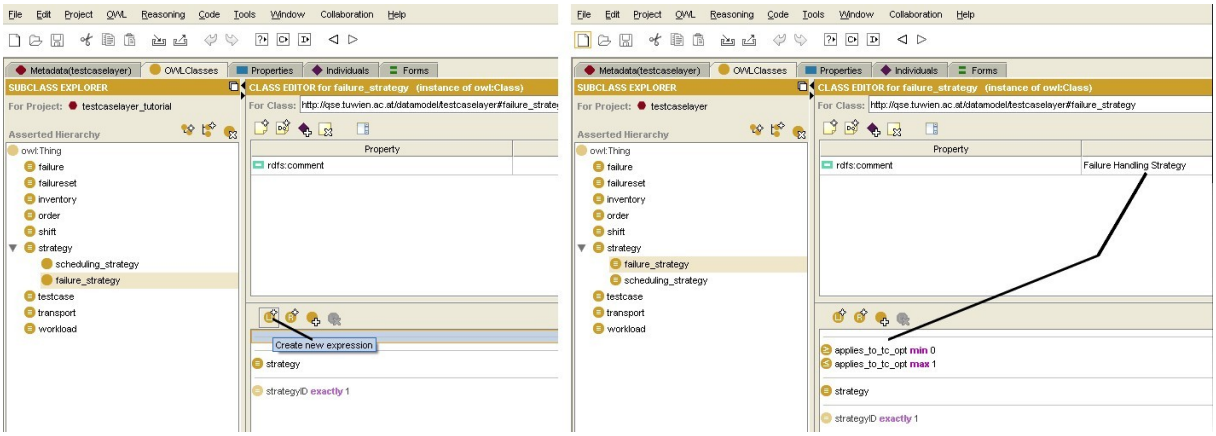


Figure C.10: Screenshot of the Protégé editor (new Expression)

C.3 EER Data model of the SAW project

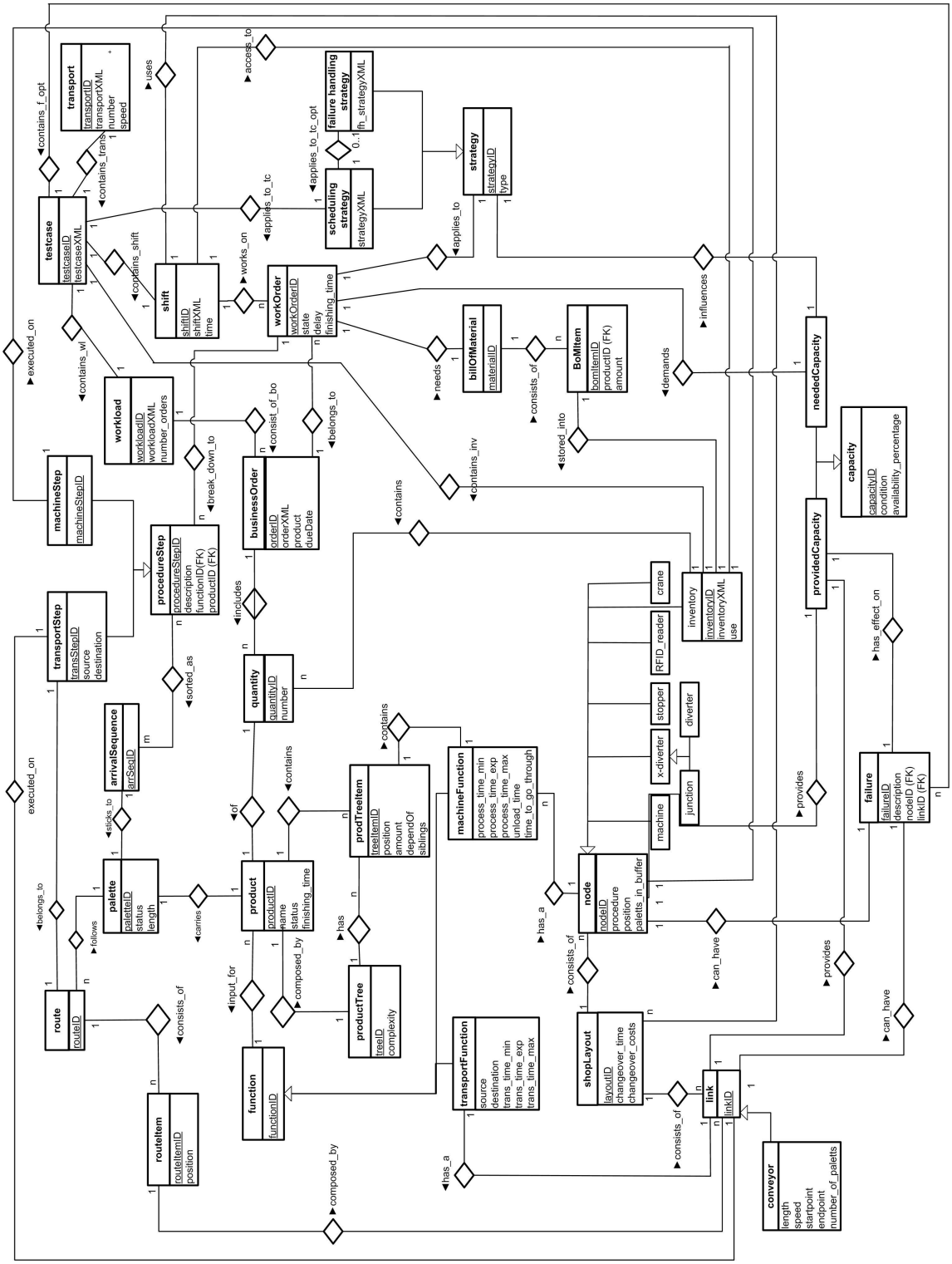


Figure C.11: Data model of the SAW project (EER diagram)