Dissertation

Symbolic Model-based Test Case Generation for Distributed Systems

Christian Schwarzl
Graz, 2012

Institute for Software Technology
Graz University of Technology

Supervisor/First reviewer: Univ.-Prof. Dr. Franz Wotawa
Second reviewer: Prof. Dr. Ana Rosa Cavalli

Abstract

The steadily increase of vehicular on-board network complexity caused by the continuous integration of driver assistance systems also requires high efforts for verification. Driver assistance systems are in general distributed over multiple Electronic Control Units (ECUs) connected via a network, which makes especially integration testing a costly and time-consuming task. The reason is that during integration testing the dependencies between the ECUs have to be resolved and corresponding input- and verification values have to be calculated in order to create an executable test case.

In this work a model-based test case generation approach from behavior models is presented, where a behavior model describes a part of an ECU functionality. In this approach Unified Modeling Language (UML) State Machines are used for behavior modeling. The existing ambiguities in the UML specification are removed by transforming the State Machines into Extended Symbolic Transition Systems (ESTSs) introduced in this work. An ESTS can describe timed behavior and contains transition types similar to those of UML State Machines. These transitions have additional properties enabling a straight forward model transformation.

In addition a test case generation algorithm for deterministic ESTSs using a deterministic communication scheme is proposed, where the deterministic communication scheme is realized by a global queue. The presented approach maintains the separation of behavior models during test case generation, which allows to distinguish their contribution to the joint functionality. This allows in combination with several state space limitation techniques a significant reduction of the considered state space during the test case generation and enhances the scalability of the presented approach.

This is supported by the experimental results obtained from the industrial use cases with STATION, which is an implementation of the presented approach. The results show that the presented state space limitation techniques are suitable to focus the test case generation on the relevant parts of the ESTSs and therefore makes the approach applicable in industry. In addition the impact on the test case generation time, communication dependency resolution and the achieved model coverage for various state space limitation techniques has been elaborated and is discussed in this work.

STATION has been integrated into an existing industrial tool chain in order to generate integration tests automatically executed on a Hardware in the Loop (HiL) system.

Kurzfassung

Aufgrund der kontinuierlichen Integration neuer Fahrerassistenzsysteme in das Automobil entstehen hohe Aufwände bei deren Funktionsabsicherung. Fahrerassistenzsysteme sind in der Regel über mehrere Steuergeräte (SG) verteilt und über ein Netzwerk miteinander verbunden. Dies macht speziell den Integrationstest zu einer kostspieligen und zeitaufwändigen Aufgabe. Der Grund dafür ist, dass für den Integrationstest die Abhängigkeiten zwischen den Steuergeräten aufgelöst und entsprechende Eingabe- bzw. Verifikationsdaten berechnet werden müssen.

Diese Arbeit stellt einen Ansatz zur modellbasierten Testfallerstellung auf Basis von Verhaltensmodellen vor, wobei ein Verhaltensmodell einen Teil der Funktion eines Steuergerätes beschreibt. In diesem Ansatz werden Unified Modeling Language (UML) Zustandsmaschinen (SM) zur Modellierung eingesetzt. Die in UML existierenden Mehrdeutigkeiten werden während einer Modelltransformation der Zustandsmaschinen in ein Extended Symbolic Transition System (ESTS) entfernt. Ein ESTS ermöglicht das Beschreiben von Zeitverhalten und verwendet ähnliche Transitionen wie UML Zustandsmaschinen mit zusätzlichen Eigenschaften, die eine einfache Modelltransformation ermöglichen.

Darüber hinaus wird ein Algorithmus zur Testfallgenerierung basierend auf deterministischen ESTSs sowie einer deterministischen Kommunikation vorgestellt. Dieser Ansatz erhält die Trennung der Verhaltensmodelle während der Testfallgenerierung und ermöglicht so deren Beitrag zur Gesamtfunktionalität zu bestimmen. Diese Information ermöglicht in Kombination mit anderen, den Zustandsraum limitierenden Methoden, eine starke Reduzierung des berücksichtigten Zustandsraumes während der Testfallgenerierung und verbessert so wesentlich die Skalierbarkeit dieses Ansatzes.

Dies wird von den experimentellen Ergebnissen basierend auf industriellen Anwendungen unterstützt die mit STATION erzielt wurden. STATION ist eine Implementierung des vorgestellten Ansatzes ist. Die Ergebnisse zeigen, dass die vorgestellten Methoden zur Limitierung des Zustandsraumes für den industriellen Einsatz geeignet sind und dadurch nur relevante Teile der Verhaltensmodelle berücksichtigt werden. Darüber hinaus werden in dieser Arbeit die Auswirkungen dieser Limitierungen auf die Generierungsdauer, die Auflösung der Abhängigkeiten in der Kommunikation sowie, die Modellabdeckung diskutiert.

STATION wurde als Teil einer integrierten Werkzeugkette zur Generierung von Integrationstests verwendet, die auf einem Hardware in the Loop (HiL) System ausgeführt wurden.

Danksagung

Ich möchte an dieser Stelle vor allem meinen Eltern für ihre Werte, Einsichten und Meinungen danken, welche sie mich über die Jahre gelehrt haben. Obwohl ich Letzterem öfter nicht zustimmen kann, machen sie einen Großteil meiner Persönlichkeit aus, die es mir nicht nur ermöglicht hat die Dissertation abzuschließen, sondern auch meinen eigenen Weg im Leben zu gehen.

Des Weiteren möchte ich meiner Lebensgefährtin Marlene danken, die während der Zeit des Schrei-bens dieser Arbeit oft persönliche Wünsche und Bedürfnisse zurückgestellt hat, um mir ein Weiterkommen zu ermöglichen.

Ich möchte auch meinem Betreuer Franz Wotawa sowie Bernhard Aichernig und Bernhard Peischl für die Unterstützung und die interessanten Diskussionen danken. Ihre Unterstützung hat maßgeblich zum Erfolg dieser Arbeit und des gesamten Projekts mit unseren Partnern aus der Industrie beigetragen.

Zu guter Letzt möchte ich mich auch beim Virtual Vehicle und im speziellen bei Daniel Watzenig bedanken, der mich stets unterstützt und mir die Möglichkeit gegeben hat, in einem spannenden Umfeld diese Dissertation abzuschließen.

Die Autoren danken dem "COMET K2 Forschungsförderungs-Programm" des Österreichischen Bundesministeriums für Verkehr, Innovation und Technologie (BMVIT), des Österreichischen Bundesministeriums für Wirtschaft, Familie und Jugend (BMWFJ), der Österreichischen Forschungsförderungsgesellschaft mbH (FFG), des Landes Steiermark sowie der Steirischen Wirtschaftsförderung (SFG) für die finanzielle Unterstützung.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....	....
place, date	(signature)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

....	....
Ort, Datum	(Unterschrift)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of hardware, software and software enabled embedded systems requires a sound validation and verification process to ensure that the product satisfies the customer's needs. During validation it is determined if the product provides the necessary functionality requested by the users. Simply spoken this means that it is checked if the product does what the user wants it to do. In contrast verification aims at ensuring that a product meets its requirements, which were elaborated at the beginning of the development process. In other words during verification it is assessed if the product was correctly developed with respect to its specification. The focus of this work lies on the verification of a product or in more detail on the enhancement of the test automation used for verification.

Testing is one strategy among others like formal correctness proofs or code inspection to show that a product fulfills its requirements. In contrast to the other stated methodologies, testing has the advantage that it examines the product as a whole and does not take only parts of it into account. For example an embedded system consists of a piece of hardware including a microprocessor, memory and software to serve a given purpose. While during code inspection or formal proofs usually only the software is considered, testing also takes the hardware into account, which can be seen as an additional execution constraint on the system as a whole.

This means that even if the software is proven to be correct, it might not run correctly because of hardware limitations. This could for example be the case if in the embedded code more memory is allocated than physically exists. In such scenarios code guaranteed to be correct could lead to incorrect behavior due to the unconsidered constraints of its underlying hardware platform or changes in the environment in which it is embedded. For this simple reason a partial verification – even if its correctness can be proved – is insufficient. Therefore the correctness of a system must be verified as a whole, which is done by testing.

1.1. Testing

The aim of testing is to *confirm* the functional correctness of the System Under Test (SUT) by performing experiments called test cases. A test case can be seen as a designed experiment on the

SUT in a controlled environment to determine if the SUT behaves in the intended way. A test case can be executed manually by a test engineer or automatically by a test automation framework. Either way a test case basically describes a sequence of interactions between the test environment and the SUT. During the execution of the test case the test environment provides – manually or automated – inputs to the SUT and observes its reactions or outputs. If the produced output after some given inputs match the intended ones, the test case is considered to be successful and receives the verdict **pass**. In case the outputs do not match the required ones the test case failed, which is indicated by the test verdict **fail**.

The main challenge of testing is that it cannot proof the correctness of the SUT, which was also pointed out by Edsger Dijkstra:

"Testing shows the presence, not the absence of bugs."

Although the overall correctness of the SUT cannot be shown by testing, the correctness of each executed test case however can be determined. Therefore in practice an appropriate set of test cases is created and used to show the correct behavior of the SUT under defined circumstances.

Unfortunately the definition of an appropriate set of tests cannot be given in general and has always to be tailored to the application and its specific requirements. For example this means that for a safety relevant system the validation requirements are much more rigorous than for uncritical systems, where an error does not cause severe injuries or is life threatening to humans. In addition even simple systems can have an infinite amount of different executions and system states. They are often caused by loops in the code and intensive usage of data, respectively. This situation usually prohibits a complete and exhaustive test of the SUT, where all possible executions are examined.

In the literature these challenges are often referred to as the state space explosion problem, which means the exponential growth of possible system states with increasing size. The state space explosion is the main challenge for testing and especially for the automatic generation of test cases, which easily leads to very long generation times and strongly limits their scalability. This issue has restricted the usage of test generation – or more general – formal methods in an industrial context, where these techniques were only applied in very dedicated problem sets. A possible solution to reduce the state space explosion problem is the limitation of the employed search methods. This means that only the required parts of the state space in order to fulfill a verification goal are taken into account, whereas the remainder is neglected. Such reduction techniques are introduced in this work and will be discussed in detail in the remainder.

The high number of test cases needed to verify a complex system also imposes great demands on the test execution, which must be efficient, reliable and has to deliver reproducible results. Due to these requirements a lot of effort has to be spent for development of test automation environments, which are specifically tailored to the SUT. They can consist of large hardware configurations and corresponding control software allowing the execution of test cases. Especially in the case embedded system tests the test hardware can become quite complex. The reason is that in such cases the hardware interfaces of the SUT have often to be read and evaluated in real time, which requires powerful computers. This also makes the creation of inputs difficult, which can

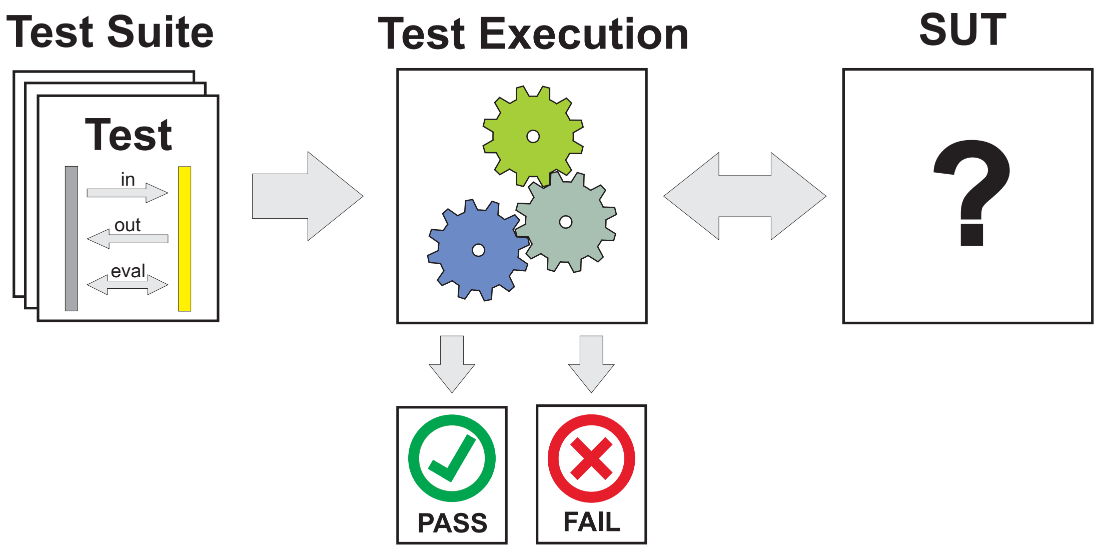


Figure 1.1.: Test Automation Framework

e.g. be a simple voltage level at a port, complex analog signal curves or messages sent on a bus corresponding to a given protocol.

Nevertheless a high degree of automation is beneficial especially in the case when the same tests have to be executed multiple times. This is for example the case after a bug has been fixed in order to check if undesired side effects have caused errors in the already existing and tested parts of the SUT. In such cases it is crucial that the test cases are executed every time in the exact same way to allow the comparison of the obtained results. This heavily favors the automatic execution of tests over a manual approach, which is due to the reliance on humans during execution error prone.

The process of an automated test case execution performed by a test automation framework is shown in Figure 1.1. In this figure the test cases are passed to the automation framework for execution, which communicates bidirectional with the SUT. The bidirectional communication between the SUT and the test automation framework is necessary to be able to provide the inputs and to observe the produced outputs from the SUT as defined in the test case. These test components are depicted as *in* and *out* in the test case. In order to allow an easier test management with a growing number of test cases they have to be structured. This can for example be achieved by the usage of *Test Suites*, which are collections of test cases and express functional and/or logical togetherness.

Moreover a test case contains an evaluation function like $\texttt{eval} \mapsto \{\mathbf{pass}, \mathbf{fail}\}$ to check the produced outputs for their correctness. Each evaluation function returns a verdict, which is either **pass** or **fail**. During the evaluation every output produced by the SUT is compared to the corresponding expected output defined in the test case. Since a test case can therefore contain multiple evaluation functions, all of them have to be taken into account during the determination of the final verdict. If only one of the evaluation functions returns a **fail** then the whole test case failed.

The calculation of the verdict is done by the test automation framework and is the result of the test execution. This is indicated by the outgoing arrows of the test automation leading to the two verdicts depicted below the test automation in Figure 1.1.

In general a test automation framework has to ensure that every needed input action and output observation of the SUT can be performed and assessed automated. This means it has to provide the necessary functionality to control the SUT via a single machine-readable interface. Usually this is realized by a software library used by the test engineer during test case development. The incorporation of a test automation library allows the test engineer to describe the test case using program code, where each statement represents the creation of an input, the observation of an output or an evaluation. Such developed test cases have to follow the predominant modeling or coding standard and are usually split into the following three phases:

1. **Initialization**: During the initialization phase the SUT is brought into a state, which allows the execution of the desired test. This means all the preliminaries which are required for a correct test execution are fulfilled after this phase. An initialization for example can be the creation of necessary objects in a software test or an activated ignition in an automotive test.
2. **Execution**: In this phase the desired test case is executed and the resulting test verdict is created. This is the main phase of the test process and shows if the SUT behaves correctly under predefined conditions. These conditions are partly specified during the initialization phase, because the same test case can lead to different verdicts depending on the current state of the SUT.
3. **De-initialization**: The post processing after a test case execution is done in the de-initialization phase, where the SUT is brought into a predefined state. This allows the test automation to setup the SUT for the next test execution or e.g. to deactivate it. Especially the latter becomes more important if heavy machines like engines are involved in the test, which could harm humans in the worst case.

The usage of the test automation and its corresponding libraries allow for an easier test execution, but the test cases still have to be created manually. For this reason the test case development and testing in general is a very laborious task and accounts for an high amount of the overall product development cost. Moreover the manual test case development is due to the usually high system complexity of the SUT error prone. The complexity of these systems often arises from non-deterministic and time dependent behaviors, which are caused for example by the communication of their components via a network. These systems require accurate test cases, which ensure the correct order for input actions and output observations, parameter values and timings. This forces the test engineer to consider multiple constraints at once, which complicates the test development considerably.

Given such test cases containing the input actions and output observations, the evaluation of the outputs in order to determine the test verdict still has to be done. This is done – as mentioned above – by comparison of the observed values with reference values, which are stored in the test case. It is assumed that these reference values are provided by an impeccable oracle and are therefore always correct. Unfortunately such an oracle does not exist in reality and the reference values have to be provided by test engineers corresponding to a given specification.

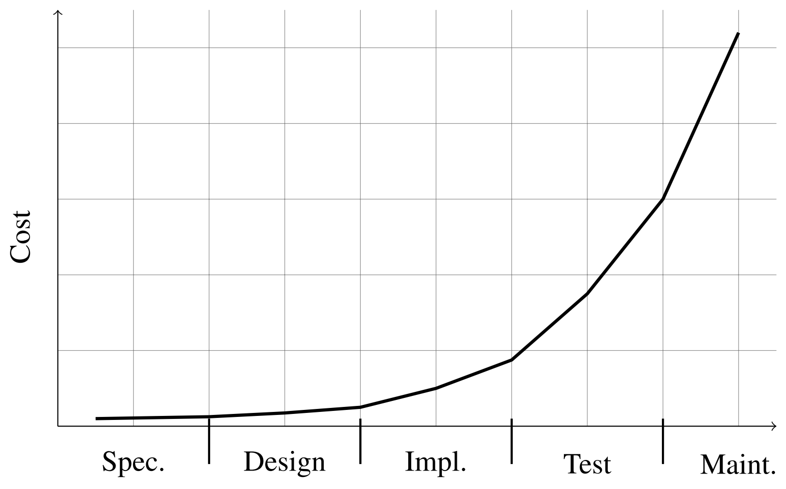


Figure 1.2.: Error fixing cost with respect to the development process.

A specification of the intended system behavior is usually given in a textual form, which is human readable and allows for an easy understanding. These kind of specifications often lack the required level of detail, which makes them incomplete, inconsistent and leaves room for various interpretations. These inconsistencies and ambiguities force the test engineers to incorporate implicit knowledge of the system gained by experience. However, such knowledge might get lost over time and due to its undocumented nature is difficult to incorporate into the specification for improvements.

The main challenge imposed by a flawed specification is that these flaws can cause very high cost if they are detected at the end of the development process. This can for example be the case during integration testing, where the separately developed components are tested interconnected for the first time.

The exponential growth of error fixing cost with respect to the development process is exemplary shown in Figure 1.2, where the process consists of a specification-, design-, implementation-, test- and maintenance-stage. It shows that an early error detection can tremendously reduce the error fixing cost. An early error detection can for example be achieved by the usage of formal models during the specification stage, which describe the intended system behavior.

Due to the formalism of these models their defined behavior can be simulated. The simulation of these behavior models allows to perform experiments and therefore helps to achieve a better understanding of the system already at an early development stage. In addition it is possible to avoid inconsistencies using automated checks and to reveal underspecified parts of the specification.

These behavioral models can also be used during the validation and verification phase at the end of the development process, where they can serve for example as test oracle.

1.2. Model-based Testing

Generally in Model-based Testing (MBT) the correctness of the SUT is verified against a formal specification of its intended behavior. This formal specification is provided in form of a behavioral model, which is unambiguous, assumed to be correct and machine readable.

Unambiguous in this context means, that the model semantics is uniquely defined. The underlying semantics specifies the meaning of the model, which can be seen as its unique behavior during a simulation. However, the uniqueness requirement still allows to describe the behavior of non-deterministic systems, but the non-determinism has to be expressed explicitly.

Unfortunately the model correctness assumption is often hard to fulfill in practice, because the model is created during a manual development process as well. Although techniques like model checking [1] exist, it is usually not possible to guarantee its overall correctness. Model checker tools like Spin [2] allow for an exhaustive model verification according to given properties. However, the overall correctness of the model can only be shown by a complete properties set, which practically cannot be defined. For this reason such models are assumed to be correct, which can be shown to a certain extend.

However, the usage of formal specifications or behavioral models in the development process is a very promising technique to enhance the overall system quality. Although the initial creation of the models needs additional effort, their usage in the early development stage and their high reusability during system evolution suggest their application. An advantage is the simulation of the models in an early development phase, which allows the detection of inconsistencies and incompleteness often detected much later in a non-model based process. The early error detection allows for a significant error fixing cost reduction according to Figure 1.2. Moreover they can be used as part of the documentation and discussions, because they provide a high level of detail often missing in textual specifications. These features allow to improve the efficiency of the whole development process if the models are used in an appropriate way.

Simply spoken the goal of model based testing is to determine if a SUT is correct by comparing its behavior to a formal specification. This raises two questions: First, how do we find out if the SUT is correct and second how do we do the comparison? The first question is easy to answer in this case, because we discuss challenges of model based testing in this work. For this reason we want to perform tests or – as we explained above – a set of defined experiments on the SUT to find out if it works properly. The answer to the second question depends on the knowledge of the internals of the SUT like its source code. If the internals are known then we can perform a so called white-box testing strategy, which takes this internal knowledge into account to achieve e.g. a good source code coverage. Otherwise the test strategy can only rely on the input/output behavior of the SUT, which is known as black-box testing. During black-box testing only the interface of the SUT is tested corresponding to its specification, where the internals – due to the lack of knowledge – are neglected.

In this work we focus on a model based black-box testing approach, where feasible inputs are generated from a model and are executed on the SUT. After the input execution the observable outputs of the SUT are captured and compared to the outputs specified by the model. If all observed outputs are correct with respect to the model then the test has passed and failed otherwise.

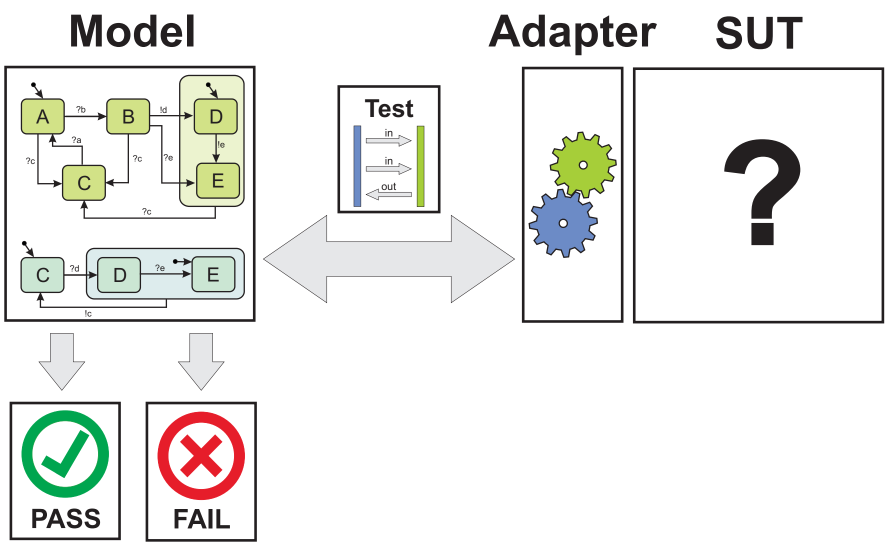


Figure 1.3.: Model-based Testing

This model based testing approach is in general depicted in Figure 1.3 and basically consists of a model, an adapter and the SUT. The model is the formal specification of the system behavior, which is assumed to be correct and complete with respect to the test goal. This means that the parts of the SUT which shall by tested are fully described in the model. From this model a test case – shown as the *Test* above the double arrow in Figure 1.3 – is generated and executed on the SUT. More specifically it indicates the set of inputs created from the model and the observed outputs from the SUT, which are used to calculate the test verdict by comparing them to the allowed outputs.

The adapter realizes the connection between the model and the SUT. It is often realized by a mapping between an abstract signal defined in the model and a concrete action executable or observable by the SUT. For example the abstract signal *pushButton* in a model could be realized via the adapter by controlling a robotic arm to push a real button. The same is true for output observations, which also can be physical changes like the flow of current or the pressure in a pipeline. This means that the adapter maps the model signals on an existing test automation framework to execute the abstract test cases created from the model on the SUT.

A model based testing and development approach based on behavioral models can improve the whole development process if it is integrated in an early development stages like requirements engineering and system design. It has several advantages like the reusability in the case of a specification change and allows the early verification of the system. In addition it enables the determination of the functional test coverage, meaning the coverage of the test cases with respect to the provided functionality of the SUT. This favors approaches based on formal specifications over test generation strategies based on rearranging atomic test case parts, which are for example focused on testing the system according to its usage [3, 4]. Although these methods have shown their applicability in industrial processes and proved to enhance the test development efficiency, they only can provide information about the test coverage on the basis of their usage models.

Since these usage models are only used for the purpose of test generation and do not describe the functionality of the SUT, they cannot provide any information about the functional coverage.

Example 1.1
Consider a usage model consisting of two states and one transition between them, where the transition contains an atomic test sequence. In order to generate test cases assume all possible traces through the model are generated, which is exactly the one atomic test sequence in this example. Given this trace or test case respectively, a full state- and transition-coverage is obtained, which do not necessarily reflect the number of covered SUT functionalities. □

Assuming a complex system providing many different functionalities is tested by a usage model as described in Example 1.1. In this case it is very unlikely that the generated single test cases covers all possible functionalities. For this reason the obtained model coverage measures can be misleading. This means that although the obtained full state- and transition-coverage suggest that the system is well tested, the actual coverage of tested functionalities can be far less. The reason is, that the coverage information based on a usage model can be seen as a measure for the efforts spent to test the system. In contrast the coverage obtained from behavioral models represents a ratio between the tested and the available functionalities.

Although the advantages stated above are promising, a model based testing approach imposes many challenges during its incorporation into the predominant process. These challenges include for example personal trainings, adaption of the currently used tool chain and investments into new hard- and software. In addition to that it also requires a mature test process, where a high degree of test automation is already available. Otherwise a model based approach is unlikely to be effective.

1.3. Challenges in the Automotive Industry

New and extended functionalities of a vehicle electronic system are often realized by the integration of new components into the already complex network or by extending the embedded software. Such electronic systems already consist of multiple connected buses serving different purposes and rely on different hardware standards and protocols. The most common buses used today are Controller Area Network (CAN)[5], Local Interconnect Network (LIN)[6], FlexRay[7], and Media Oriented Systems Transport (MOST)[8].

1.3.1. Bus Technologies

The LIN is the cheapest and simplest bus of an on-board network and is mainly used to connect sensors and actuators to an Electronic Control Unit (ECU). It supports a simple protocol, which allows the ECU to control the actuators and request data from the attached sensors.

The CAN bus is the most widely used bus in a vehicular on-board network and has been introduced about two decades ago. It has been continuously enhanced and provides now bandwidths up to 1 Mbps. The CAN bus is based on an asynchronous access protocol, allowing the attached

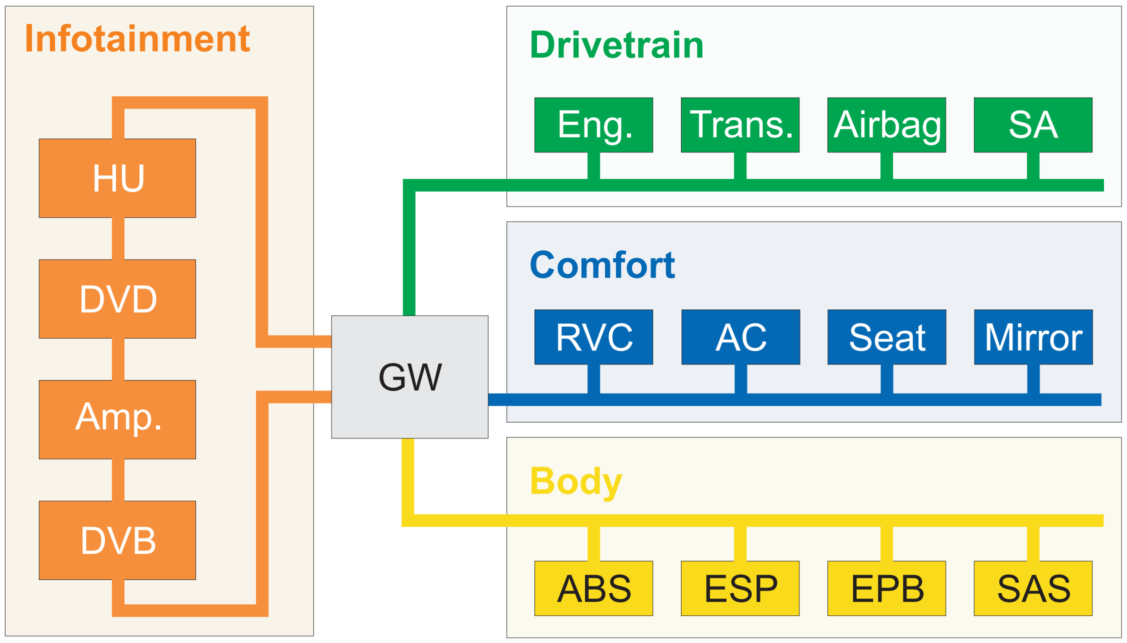


Figure 1.4.: Automobile on-board network example.

nodes to send data at any time. However, this can cause collisions if two nodes want to send at the same time, which results in a unpredictable delay until the data is actually sent. Due to this possible delay it cannot be guaranteed that a sent message will be received after a specific amount of time, which is a crucial property of real time systems.

The lack of real time capability of the CAN bus is remedied by the FlexRay bus, which was developed especially for this purpose. It does not rely on an asynchronous communication scheme for bus access, because every node has a dedicated time slot in which it is allowed to send data. This mechanism prevents the occurrence of collisions and allows to guarantee that a message is sent and received within a given period of time. However, the limitation of the allowed sending time due to the time slicing mechanism reduces the effectively available bandwidth of each node.

The MOST bus was introduced to provide a high bandwidth connection between the ECUs, which today operates at 150 Mbps. It is used mainly for the infotainment system, where a high bandwidth is required to be able to interchange multimedia content like video and audio streams. The MOST bus is based on a ring network topology and the physical interconnection are realized by optical fiber. The reason for the usage of the costly fiber technology is, that a network based on an electric interconnection becomes more susceptible to electromagnetic interference with an increasing bandwidth. The existence of many interference sources in a vehicle like the electrical starter, switched power supplies or other buses, suggested the usage of fiber optics to avoid the imposed challenges.

Example 1.2
Figure 1.4 illustrates an exemplary detail of an on-board network of a modern vehicle, which consists of four buses connected by the Gateway (GW). The bus connecting the infotainment

ECUs shown in orange is a MOST bus, while for Drivetrain- (green), Comfort- (blue) and Body- (yellow) ECUs a CAN bus is used. The Infotainment system consists in this example of four ECUs, namely the Head Unit (HU), Digital Versatile Disc (DVD) player, Amplifier (Amp.) and Digital Video Broadcast (DVB) receiver. They exchange mainly multimedia data requiring the high bandwidth provided by the MOST bus.

The Drivetrain consists of ECUs to control the Engine (Eng.), Transmission (Trans.), Airbag and Shock Absorber (SA). The Comfort bus connecting the components used in the passenger cabin enhancing the comfort during driving and consists of a Rear View Camera (RVC), an Air Conditioner (AC) and an ECU for seat (Seat) and mirror (Mirror) adjustments, respectively in this example.

The control of the car body is supported by the Electronic Stability Program (ESP), Anti Blocking System (ABS), Electronic Parking Brake (EPB) and a Steering Angle Sensor (SAS), which measured data are combined in order to enhance the stability during a drive. □

1.3.2. Quality Assurance

Especially for infotainment systems, which represent the Human Machine Interface (HMI) between the vehicle and the passengers and rely on information distributed over the whole electronic system, the validation of its provided functionality is a complex undertaking. The reason is that the infotainment system is not only limited to the entertainment of the passengers by providing multimedia contents, but has already been extended to be the central configuration and control point of the whole vehicle, where many customizations and adjustments of car settings are available. For this reason the used software has to gather, present and to change settings provided by different systems like driver assistant systems and body control.

Due to the interaction and the deep integration into other systems its functional correctness plays a crucial role to ensure that it does not cause any unwanted behavior, which in the worst case can lead to heavy injuries of the passengers. In addition errors in the infotainment system are also critical regarding their perception, because faults can be immediately observed and therefore have a direct influence on the overall customer satisfaction. Moreover the used hardware and software cannot be changed or updated easily, since it requires special devices not generally available. For this reason the number of needed maintenance tasks has to be kept as low as possible, because of the perceived vehicle reliability.

This situation imposes new challenges on the system validation, where each component usually cannot be tested alone anymore. This means that such components have to be integrated into the network for which they were designed in order to function properly. The construction of such networks used for testing is difficult, because not all ECUs are physically available or have reached a sufficient degree of maturity at a given point in time.

In order to be able to execute integration tests on an ECU network a Hardware in the Loop (HiL) system can be used to control the SUT and to perform real time measurements. A HiL system can be connected with the ECUs via the network and various hardware interfaces, which is needed in order to provide inputs and to track their outputs. Moreover it is possible to substitute physically unavailable ECUs by a simulation of their corresponding behavior models, if the ECUs are not part for the actual test.

1.4. Motivation

The increasing usage of models as behavioral specification in the industry allows the application of model based test strategies to verify the correctness of the SUT.

Especially the graphical Unified Modeling Language (UML) [9] has gained wide acceptance in the industry today. UML has been standardized by the Object Management Group (OMG), who intentionally underspecified parts of the specification to allow some implementation freedom for the tool developers. UML itself provides many different diagram types where we focus on statechart-, and object model-diagrams in this work. Especially state chart diagrams provides a simple way to describe the system behavior with State Machines (SMs), which are well suited to model communicating reactive systems. They support many modeling elements and structures like hierarchy, Nested State Machines (NSMs), parallel regions and pseudo-states, which allow an efficient model development.

The availability of many commercial UML modeling tools like RHAPSODY [10], ENTERPRISE ARCHITECT [11] or VISUAL PARADIGM [12] provides a sound basis for the widespread application of UML. Many tools use code generators to generate code representing the behavior described in the models. This code can be passed to a compiler to create an executable, which allows the simulation and a graphical animation of these models. The model simulation enables an early validation and verification of the described system.

The implementation freedom in UML affects the semantics of the models with respect to the used tool. The reason is that the code generators of the respective tools interpret the models differently, which leads to a different behavior of the simulation or semantics for the same model. This means that a model might cause errors or unexpected behaviors in one tool, whereas it was working as intended in another one. These tool specific interpretation of the model semantics also limits the model exchangeability between the tools itself.

However, the provided modeling possibilities, the introduced formalism and the semantics defined by the code generator suggest the usage of UML as modeling language. In addition the availability of mature tools allows their integration into existing development and test processes, where a high reliability is crucial. In addition the Extensible Markup Language Metadata Exchange (XMI) defines a standard data format for UML model exchange, which allows a simple import of these models in other tools.

The usage of code generators and compilers for model simulation requires complete and unique models. This means that all necessary information needed to describe the model semantics is available. Therefore these models form a sound basis for the introduction of automatic test generation strategies.

1.5. Problem Statement

Most newly introduced functionalities of a vehicle are realized by embedded systems consisting of multiple ECUs, which are connected via a network. These components work closely together in order to be able to realize such functionalities. Due to their distributed nature and their usually independent development, integration testing plays a crucial role to verify the correct behavior of the implemented functionalities. During integration testing the necessary components are connected to each other and are examined as a whole. The difficulties in integration testing are to create test cases, which incorporate the communication and data dependencies between the components.

Example 1.3
Let an embedded system consist of three components namely A, B and C, where a function a is implemented in component A and shall be executed during a test. Assuming that a is triggered by function b realized in component B and sends its result to C, a test case has to ensure that b is executed and component C is observed in order to test a. $\square$

As shown in Example 1.3 all involved components implementing a part of the whole system functionality have to be taken into account during an integration test. The manual development of such test cases is error prone due to the complexity imposed by these dependencies. In addition the intensive usage of data makes the test case development even more complex, because their values might change the behavior and influence the values used to assess the correctness of the SUT.

The generation of test cases from formal models is considered to be a promising approach to tackle these challenges. Over the past decades many different notations have been used to formalize the behavior of embedded systems and software. However, most of these approaches were developed and used in scientific applications, which are often not suitable for an industrial setting. The main reasons are the drawbacks encountered regarding usability, tool reliability and user support.

The usage of UML as the predominant modeling language and the strong tool support suggests its usage as the formal basis for test case generation. However, the implicit semantics imposed by the usage of a code generator has to be formalized and incorporated in order to enable the generation of test cases. The generated test cases are therefore correct with respect to the used modeling tool.

The main challenges in test case generation from a formal specification are the calculation of feasible input sequences, the according input parameter- and verification values and the handling of the state space explosion:

Feasible input sequences: During the execution of a test case input actions are performed on the SUT and its reactions are observed. Therefore the correct input order plays a crucial role for the test case generation. Especially in the case of a distributed system the communication dependencies between the components have to be taken into account as indicated in Example 1.3. Since the behavior of these components might also depend on time, this aspect also has to be considered. This means that a test case must specify exactly the correct input and the respective time at which the input has to be provided.

Input- and verification values: Assuming the input order and the according timings are known, the desired execution of a test case might still depend on the parameter values provided with the inputs. This is usually the case if the usage of conditional statements influences the execution result depending on variable values. This means that the same input sequence with the same timings but with different parameter values can lead to different results. For this reason the input values have to be calculated in addition during test case generation in order to produce unique test cases. The changing results depending on the varying input parameter values have also to be considered during the assessment of the outputs produced by the SUT. This means that a reference value, which is used for the assessment of an output in a test case, has also be calculated corresponding to the provided input parameters.

State space explosion: The term state space explosion refers in general to the exponential growth of the possible state number with the size of the underlaying code or model. This exponential growth also reflects the number of possibilities, which have to be considered during the test case generation. The vast growth is one of the main limiting factors for test case generation, because it has a direct impact on the needed generation time.

This tense situation is intensified by the presence of loops in the model and the usage of data. A loop – even in a small model – can cause an infinite amount of possible executions and therefore an infinitely large state space. The usage of data or variables increases the state space additionally, because each possible variable value reflects a possible state. The combination of multiple variable values leads to an enormous growth of the state space.

Example 1.4
Let x and y be two 8 bit variables, which means they have $2^8 = 256$ possible values in the range $[0..255]$. Assuming both variables are used in the same model, these two variables can already represent $2^8 * 2^8 = 32768$ different states. If these variables were 32 bit variables they could together have about $1.8E^{19}$ states. Assuming that one test case could be created in $1E^{-9}$ seconds, the generation of a test case for every possible state would take about 808 years in the second case. $\square$

In a distributed system, where the components communicate over a network, the communication raises another aspect regarding complexity. Especially in the case if an asynchronous communication scheme is used, the order of the messages sent over e.g. a bus cannot be predicted. The reason is that the components running independently of each other, which causes races if two components want to send messages at the same time. Since such situations are in general not solved deterministically any possible scenario has to be considered.

The whole model-based test generation process stands and falls with the correctness of the formal specification. If for example the model contains faults, then the generated test cases based on this model will also contain these faults. Due to this error propagation the model correctness plays a crucial role in this approach, but such models are usually developed manually which unfortunately is error prone.

However, since for verification it is sufficient to detect differences between the specification and the implementation the usage of a model-based test case generation approach is still feasible. The reason is that specification errors only lead to false positives, which can be used to localize and remove them from the specification. A false positive in this context means that an error in a correct implementation is found.

1.6. Thesis Statement

A model-based test generation approach with explicit model dependent restrictions can be used to generate test cases for large systems from specification languages widely spread in industry.

1.7. Contribution

The focus of the presented approach is the generation of test cases for integration testing, where the behavior of each component is specified in a separate UML State Machine. A set of State Machines is the starting point for our implemented prototype STATION, which transforms the State Machines into Extended Symbolic Transition Systems (ESTSs) to circumvent the ambiguities in UML. On basis of the ESTSs test cases are generated with respect to a given test purpose, which is specified directly in the UML State Machines. For this reason this approach is independent of the used UML modeling tool and allows in addition the transformation of other formal specifications.

Given the challenges described in Section 1.5, the test case generation approach presented in this work requires some restrictions on the system modeling in order to avoid the immanent state space explosion problem. For this reason we require the following restrictions to enable a fast and scalable test case generation:

- **Deterministic Models:** The restriction to deterministic models reduces the possible state space heavily and therefore the needed test case generation time. The reason is that for deterministic models exactly one execution is possible. In the non-deterministic case, meaning that for one input multiple transitions could be traversed at the same time, each of these possibilities represent a different execution. In order to be able calculate a verdict for a test case, all possible executions have to be taken into account.
- **Deterministic Communication Scheme:** The potential state space can be further reduced by the requirement for a deterministic communication scheme. This requires the absence of races in the communication between the components and models. Although this requirement cannot be held in a real embedded system connected via a network, this simplification allows to describe the system behavior in multiple models and vastly simplifies the model development.

The restrictions stated above also limit the applicability of the presented approach. However, these limitations allow the generation of test cases for a dedicated problem set, which is sufficient for our research context as described in Section 1.8.

Modeling: In this work we introduce an new methodology for the integration of a test automation and for the systematic definition of test cases directly in UML State Machines. This novel concept allows the test engineers to specify and generate test cases from a single tool and therefore within the same user interface.

Formal Model: Although the test case generation is restricted to deterministic models, a more general formal model named ESTS is introduced in this work. It extends the Symbolic Transition System (STS) defined in [13] mainly by delay transitions, which traversal is triggered by the elapse of time. In order to simplify the modeling also completion transitions and transition priorities are introduced. These elements do not add any additional semantics, but allow a simple and straight forward model transformation of UML State Machines into ESTSs.

Model Transformation: In order to incorporate the implicit semantics imposed by code generators for UML State Machines, we provide formal transformation rules for the transformation of State Machines into ESTSs. Since the available code generator shipped with the commercial tools can differ, the used transformation in this work is based on the semantics used in RHAPSODY. In addition we support an efficient modeling based on UML State Machines by supporting a wide range of modeling elements and structures like pseudo-states, state hierarchy and Nested State Machines.

Test Case Generation: In this work a systematic test case generation strategy based on test purposes is used. In this systematic approach the test purpose is specified in the UML State Machines and consists of a list of states and transitions, which have to be reached or traversed in the resulting test case, respectively.

State Space Limitation: Since the state space can become infinitely big it has to be limited in order to be able to generate test cases. In this work we introduce new techniques, which allow a fine grained tailoring of the used search algorithms to find ways through the ESTSs and UML State Machines, respectively. This techniques allow to precisely define the desired test purpose and enables the incorporation of heuristics to simplify its usage.

Conformance Relation: Since a deterministic model and deterministic communication scheme is required, a conformance relation between the model and the SUT based on **tioco** [14] has been used. This allows a precise definition of the correctness of the SUT behavior with respect to its specification.

The topics stated above were introduced and discussed in the following peer reviewed papers, which have been written throughout this research project:

- The works [15, 16] describe the model transformation of UML State Machines into Language of Temporal Ordering Specification (LOTOS)[17], where we used the tool TGV [18] to generate test cases corresponding to a given test purpose specified in the UML State Machines.
- In order to compare the scalability of an enumerative and a symbolic approach we introduced the ESTS and showed the model transformation from UML State Machines in [19, 20]. However, the handling of time and delay transitions were not taken into account in these papers. Based on this formal representation we applied a random and systematic test case generation approach, where we used the same test purpose definitions as above.
- The paper [21] shows a refined version of the ESTS and precisely defines its semantics. It provides a formal description of its structure and introduces the delay transition. The delay transition is triggered by the elapse of time and is similarly defined for UML State Machines. This new structure enables a straight forward model transformation of UML State Machines containing delay transitions.
- In [22] we introduce and discuss the model search strategies in order to reduce the state space during the test case generation. We show the basic components, their application

and implications on the state space and generation time, respectively. Based on these basic methodologies we introduce a heuristic, which allows a fast test definition while the considered state space still can be strongly reduced.

- Since the test case generation relies on the correctness of a specification or model, this crucial issue is addressed in [23]. In this work we propose the usage of test case generation techniques in order to analyze the models fully automated. We suggest the usage of assertions on these models, which are well known from the software development field. In addition to the condition evaluation during the execution we allow to perform checks on the test case generation result used for the model analysis. This means that the result, which is obtained from the test case generation algorithm, can be checked according to some given criteria. For example such an assertion could require that for a given transition five traces exist in another model ensuring its traversal.

Summarizing the made contributions, a symbolic formalism allowing an easy transformation from UML State Machines has been defined and a random and systematic generation of test cases including the automatic analysis of these models has been implemented and applied to industrial use cases.

1.8. Research Context

This thesis is the result of consecutive research conducted in a joint research project by the Area E of the Virtual Vehicle, the Institute of Software Technology of the Graz University of Technology and two industrial partners from the automotive industry.

The project was funded by the “COMET K2 Forschungsförderungs-Programm” of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economics and Labour (BMWA), Österreichische Forschungsförderungsgesellschaft mbH (FFG) and Das Land Steiermark and Steirische Wirtschaftsförderung (SFG). It had a duration of three years and was funded in order to establish a collaboration between practitioners and scientists to undertake industry related joint research.

The goal of the project was to analyse the predominant testing environment of the industrial partner and to develop a prototype implementing a model-based test generation approach based on the existing models and tool chain and to assess its scalability. Especially the integration of the prototype into the existing tool chain was of utmost importance, because short time changes to the test automation framework would cause significant cost likely to overwhelm the benefit of the new approach.

The prototype forms also the basis for a later commercial tool development, which is required for the usage in an industrial process to ensure its functionality, maintenance and support throughout its usage. In addition the developed approach was intended to be worked with interactively. For this reason the prototype has to provide fast results, which means the test generation must not take longer than a couple of seconds to be accepted by the test engineers.

These stringent requirements could only be achieved by reducing the state space as much as possible. Since it was sufficient and desired by the industrial partners to develop test sequences

– meaning that non-deterministic behavior can be neglected – the limitation to a deterministic communication scheme and also deterministic models was suggested. Although, these restrictions limit the applicability of the approach in a more general setting, it was still sufficient to enhance the current state-of-the-art of the industrial partners.

The restriction to a deterministic communication scheme prevents in general its usage for testing a real distributed system. However, in this context we assume that the observed results after the provision of certain inputs are always the same, regardless the concrete communication or implementation. This means that the used models or specifications represent an abstracted version of the implementation containing the system behavior in principal.

For this reason the test case generation can focus only on the generation of the required inputs distributed over different points of the SUT and the observations of their caused reactions.

1.9. Outline

The remainder of this thesis is organized as follows: Chapter 2 provides a formal definition of an ESTS and its corresponding paths and traces. In this chapter also the composition of ESTSs is presented and the used conformance relation is defined. Chapter 3 explains the test case generation algorithm used to create test cases corresponding to a given test purpose, which is also defined in this chapter.

Chapter 4 gives an overview of UML and State Machines including its supported modeling elements. UML State Machines are used as modeling language in the industrial setting considered in this approach and are transformed into ESTSs. The model tranformation is described in Section 4.2. In Chapter 5 the tool chain currently used for integration testing is described and the integration of the prototype STATION is explained, which implements the presented test case generation approach.

Chapter 6 shows the results obtained for two illustrative examples and two industrial use cases, where test cases for integration tests were generated. The thesis closes with Chapter 7, where the related work is discussed and a summary as well an outlook to future work is provided.

Chapter 2

Symbolic Framework

Parts of this chapter are taken from ´An Extended Symbolic Framework for Systematic Test Case Generation´ which is an ongoing work with Bernhard K. Aichernig and Franz Wotawa and from ´Compositional Random Testing Using Extended Symbolic Transition Systems´ [21] which is joint work with Bernhard K. Aichernig and Franz Wotawa.

2.1. Extended Symbolic Transition System

In order to allow the usage of UML State Machines to define the system behavior, we describe their semantics in terms of ESTSs as defined in this section. An ESTS is essentially a symbolic version of a Timed Labeled Transition System (TLTS) [14] extended by syntactical constructs. These syntactical extensions are introduced in order to maintain the structure of the UML State Machines and to enable a straightforward model transformation. This allows the direct incorporation of additional information provided by the test engineers like test definitions and state space exploration constraints as described in Section 3.

In this work, we provide a general definition of an ESTS, which includes deterministic as well as non-deterministic behavior in order to provide a basis for a later extension of the presented approach. However, the presented conformance relation and test case generation algorithm are restricted to deterministic systems.

2.1.1. Structure

The structure of an ESTS is closely related to an UML State Machines specification and contains similar elements. The ESTS introduced in this work is based on the STS defined in [13] and extends it by (1) timing groups, (2) delay transitions, (3) completion transitions, (4) transition priorities and (5) transition execution durations.

Due to the introduction of these additional elements the actions and guards of an UML State Machine can be directly transfered into an ESTS during a corresponding model transformation.

Definition 2.1 (Extended Symbolic Transition System)
An Extended Symbolic Transition System is a tuple $\langle S, L, A, P, T, G, q_0 \rangle$, where S is a set of states, L is a set of labels, A is a set of attributes, P is a set of signal parameters, T is the transition relation, G is a set of timing groups and q_0 is the initial configuration. [1] $\square$

The set $L_{io} = L_i \cup L_o$ is the union of input- L_i and output- L_o signals used for communication, where $L_i \cap L_o = \emptyset$ and $L_* = \{\tau, \gamma\}$ is a set of labels representing the unobservable- and completion-transition. An unobservable transition indicates a non-deterministic behavior, which cannot be observed by the environment. In the remainder we refer to transitions with a label in L_i and L_o as input- and output-transitions, respectively.

Delay transitions are transitions labeled with a delay $n \in \mathbb{N}_1$, where n specifies a time passage. The union of sets $L = L_{io} \cup L_* \cup \mathbb{N}_1$ is the set containing all labels and we use $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$ and $\mathbb{N}_1 = \mathbb{N}_0 \setminus \{0\} = \{1, 2, 3, \dots\}$.

Attributes A and signal parameters P are both variable sets used for the symbolic treatment of data. While attributes are properties of an ESTS, parameters are part of a signal and allow data transmission. The signal parameters of a transition with label l are given by the function $\mathbf{par}(l)$, where $\mathbf{par}(l) \subseteq P$ if $l \in L_{io}$ and $\mathbf{par}(l) = \emptyset$ otherwise. The attributes and parameters are disjunct meaning that $A \cap P = \emptyset$ and we use $V = A \cup P$ to address all variables.

Definition 2.2 (Variable Valuation)
A variable valuation is an ordered pair (v, u) of a variable $v \in V$ and a value $u \in \mathfrak{U}^v$, where $\mathfrak{U}^v$ is the set of possible values for v. $\square$

Given a subset of variables $Y \subseteq V$, we use $\mathfrak{U}^Y$ to denote the set of all variable valuations for Y. In addition we use the *parameter valuation* $\varsigma \in \mathfrak{U}^{\mathbf{par}(l)}$ and the *attribute valuation* $\iota \in \mathfrak{U}^A$ to refer to variable valuations of signal parameters and attributes, respectively. [2]

Definition 2.3 (Parameterized Input and Output)
A parameterized input is a tuple (l_i, ς), where $l_i \in L_i$ is an input label and $\varsigma \in \mathfrak{U}^{\mathbf{par}(l_i)}$ is its possibly empty parameter valuation. A parameterized output is defined similarly, where output labels $l_o \in L_o$ are used instead of the input labels. $\square$

The set of parameterized input- and output-labels is $\Sigma_{io} = \Sigma_i \cup \Sigma_o$, where Σ_i is the set of all parameterized inputs and Σ_o is the set of parameterized outputs. The set of parameterized labels $\Sigma = \Sigma_{io} \cup \Sigma_\tau$, where Σ_τ is the set containing a tuple $(l, \emptyset)$ for every label $l \in \{\tau, \gamma\} \cup \mathbb{N}_1$ representing the unobservable-, completion- and delay labels. For simplicity we denote parameterized labels in Σ_τ only by their labels in the remainder.

$$T \subseteq S \times L \times \mathfrak{F}(V) \times \mathfrak{T}(V)^A \times \mathfrak{P} \times \mathfrak{D} \times S \tag{2.1}$$

[1] In [13] a different naming convention is used, where states are locations, transitions are switches, attributes are location variables and parameters are interaction variables.

[2] In [24] the attribute valuation is named *context* and input signals with a parameter valuation are named *parameterized inputs*.

The transition relation of an ESTS is given in Equation (2.1), where $\mathfrak{F}(V)$ is a set of first order logic predicates over the variables in V, $\mathfrak{T}(V)^A$ is a set of attribute update functions in A over the variables V, $\mathfrak{P}$ is a set of priorities, $\mathfrak{D}$ is a set of transition execution durations.

A transition $t \in T$ is given as $(s, l, \varphi, \rho, p, d, s')$, where $s, s' \in S$ are its source- and target-state, $l \in L$ defines its label, $\varphi \in \mathfrak{F}(V)$ is its guard and $\rho \in \mathfrak{T}(V)^A$ is the attribute update function. The update of an attribute valuation ι' corresponding to a given attribute update function ρ is denoted as $\iota' = \rho(\iota \cup \varsigma)$ and we use the identity function `id` $\in \mathfrak{T}(V)^A$ to indicate that an attribute valuation remains unchanged.

The transition priority $p \in \mathfrak{P}$ is given as natural number $p \in \mathbb{N}_0$ and allows the definition of an *execution order* if a state has multiple outgoing transitions. The execution duration $d \in \mathfrak{D}$ defines a time lapse for a transition and is therefore a natural number $d \in \mathbb{N}_0$, which is used as time domain in this approach.

Definition 2.4 (Timing Group)
A timing group is a tuple $\langle c, S_g, T_g, T_r \rangle$, where $c \in C$ is its clock, C is a set of clocks, $S_g \subseteq S$ is a set of states, $T_g \subseteq T$ is a set of delay transitions and $T_r \subseteq T$ is a set of clock reset transitions. □

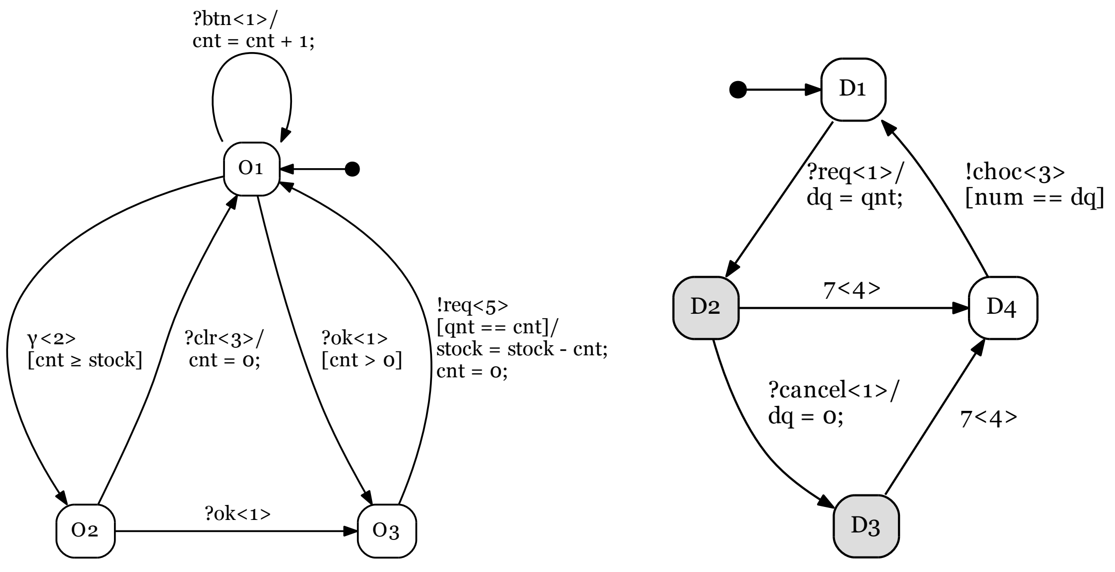
A timing group $g \in G$ defines a set of states S_g sharing the same clock c, which is used to keep track of the elapsed time. We use the *clock valuation* $\zeta \in \mathfrak{U}^C$ containing a variable valuation for each timing group clock $c \in C$, where C is a set of clocks of the timing groups in G unified with the ESTS clock c^* and $\mathfrak{U}^C$ is the set of all possible clock values.

The ESTS clock c^* keeps track of the elapsed time in the ESTS and behaves similar to a clock of a timing group with $T_g = T$, $S_g = S$ and $T_r = \emptyset$. The clock update function ξ is denoted as $\zeta' = \xi(\zeta, n)$, where n specifies the time difference used to update the clock valuation ζ of all clocks in C.

Definition 2.5 (Configuration)
A configuration of an ESTS is a triple $(s, \iota, \zeta) \in (S \times \mathfrak{U}^A \times \mathfrak{U}^C)$, where S is the set of states, $\mathfrak{U}^A$ the set of attribute valuations and $\mathfrak{U}^C$ the set of clock valuations. □

Corresponding to Definition 2.5 we denote a configuration as $q \in Q$, where $Q = (S \times \mathfrak{U}^A \times \mathfrak{U}^C)$ is the set of all configurations. In addition we use the initial configuration $q_0 = (s_0, \iota_0, \zeta_0)$, where $s_0 \in S$ is the initial state, $\iota_0 \in \mathfrak{U}^A$ is the initial attribute valuation and $\zeta_0 \in \mathfrak{U}^C$ is the initial clock valuation.

Example 2.1 (Chocolate Vending Machine)
The chocolate vending machine consists of two components, namely the Order Component and the Delivery Component shown in Figure 2.1a and 2.1b, respectively. In these figures the states belonging to a timing group are shown gray filled and the transition execution time is enclosed by angle brackets. The Order Component is used to define the number of chocolates which shall be delivered and the Delivery Component takes an delivery request in form of the signal *req(qnt)* from the Order Component, where qnt is its signal parameter.



(a) Order Component (b) Delivery Component

Figure 2.1.: Chocolate vending machine consisting of an Order- and Delivery-Component.

The number of chocolates to be delivered can be defined by repeatedly pressing a button *btn*, which increases the internal counter *cnt* realized as an attribute. If the counter reaches the number of available chocolates in stock – defined by the attribute *stock* – then the counter cannot be increased anymore and only the *ok* button can be pressed. After pressing the *ok* button a request *req* with the desired quantity is sent to the Delivery Component.

The Delivery Component takes the request from the Order Component and delivers the desired amount of chocolate, if the *cancel* button is pressed before the specified timeout *7*, which sets the delivery quantity dq to zero. □

2.1.2. Semantics

The behavior of an ESTS is described by the *executions* of its transitions. A transition execution moves the current state of the ESTS, which is also the source state of the executed transition, to the transition target state and updates the attribute valuation corresponding to its attribute update function ρ. A transition can only be executed if it is enabled according to Definition 2.6.

Definition 2.6 (Enabled Transition)
A transition is enabled for a given configuration and parameter valuation if its guard evaluates to true. A delay transition $t \in T_g$ of a timing group g has to satisfy $c \geq n$ in addition, where c is the clock of g and n is the delay of the delay transition. □

An input transition becomes enabled if a corresponding signal including its possibly empty parameters is received, whereas unobservable-, completion- and output-transitions become enabled as soon as their source state is reached and their guard is satisfied. Since delay transitions have to fulfill a time constraint in addition, it might be necessary that the timing group clock of its corresponding timing group is increased by $n - c$ to enable it.

A timing group clock c is increased by the transition execution duration d if a transition t with source- and target-state in S_g is executed. A timing group clock is also increased by ω if the ESTS waits $\omega \in \mathbb{N}_1$ time units for an input in a state $s \in S_g$ and is reset if a clock reset transition in $t \in T_r$ is executed, where a reset means that the clock value is set to zero.

The delay of a delay transition can be defined by a constant- or by an attribute value like $n := 100$ or $n := a$ if $a \in A$, which we call *static-* and *dynamic-delay*, respectively. The delay transition should not be confused with the transition execution duration. The transition execution duration specifies the amount of time which will elapse for the execution of a transition, whereas a delay transition defines the amount of time which has to elapse before the delay transition is executed. For example the transition execution time can be used to specify the amount of time needed for the update of an attribute valuation or for the time needed to send a message on a bus.

The execution of a completion transition is – in contrast to the unobservable transition – deterministic and is done (if enabled) regardless other enabled input- or delay-transitions. For this reason the possibility of a completion transition execution is checked before input- and delay-transitions are taken into account, which prevents their execution if a completion transition is enabled.

The completion transition is part of a *completion step*, which is a non-interruptible consecutive execution of multiple transitions. A completion step starts always with a delay- or input-transition followed by a number of unobservable-, output- and completion-transitions. A completion step ends if a state is reached, which has only enabled input- and delay-transitions or no enabled transition at all.

The transition priority is used to define an *execution order* if multiple transitions with the same label are enabled. The execution order is a ranking of transitions, where the highest ranked enabled transition will be executed and the remaining transitions will be ignored. This allows to maintain a deterministic behavior of an ESTS if it has multiple enabled transitions with the same label. However, this requires that every enabled transition of a state has a different priority, because otherwise it would not resolve a non-deterministic behavior.

In the remainder of this work we denote a highest ranked enabled transition as $(q, \mu, \varphi, \rho, p, d, q')$, where q is the configuration containing its source state and μ is its parameterized label.

Formally we describe the semantics of an ESTS in terms of a TLTS, which is the underlying system specification for the **tioco** [14] conformance relation. A TLTS has a simple structure consisting of states and transitions with labels, where each state represents exactly one possible system state. For this reason the variables, which are treated symbolically in an ESTS, have to be enumerated in a TLTS. This means that one state for every possible variable value has to be created. The labels are separated in inputs and outputs and define in combination with a transition the possible state changes. In a TLTS delay transitions are similar defined to those in an ESTS, where the delay – being the time needed to elapse before the transition is executed – is specified in their labels.

Definition 2.7 (Timed Labeled Transition System)
A Timed Labeled Transition System is a tuple $\langle S, L, T, s_0 \rangle$, where S is a set of states, L is a set of labels, T is the transition relation and s_0 is the initial state. □

The label set $L = L_i \cup L_o$ is the union of input- and output-labels and we use $L_\tau = L \cup \{\tau\}$ to refer to the label set including the unobservable τ label. The transition relation is defined in Equation (2.2), where $\mathbb{N}_0$ is the set of natural numbers used as labels for time delays.

$$T \subseteq S \times (L_\tau \cup \mathbb{N}_0) \times S \tag{2.2}$$

Corresponding to standard TLTS notation we denote a transition $(s, l, s') \in T$ as $s \xrightarrow{l} s'$ and use $s \xrightarrow{l}$ for $\exists l : s \xrightarrow{l} s'$. A transition $s \xrightarrow{l} s'$ of a TLTS interpreting an ESTS represents an update of a configuration and therefore includes the update of its current state, attribute- $\iota' = \rho(\iota)$ and clock- $\zeta' = \xi(\zeta)$ valuation.

A TLTS is *time additive* if Rule (2.3) and *time deterministic* if Rule (2.4) applies, where the former allows to merge consecutive transitions with a delay label into a single one and the latter states that two transitions with the same delay label have to lead to the same target state. We do not require the null delay rule as in [14], because completion transitions can cause such state changes as it will be shown later in this document.

$$s \xrightarrow{n} s' \xrightarrow{n'} s'' \equiv s \xrightarrow{n+n'} s'' \tag{2.3}$$

Rule (2.3) is especially important for the composition of multiple TLTS as described in Section 2.2, where in one TLTS multiple delay transitions can be used to synchronize with a delay transition of the other TLTSs.

$$\frac{s \xrightarrow{n} s' \quad s \xrightarrow{n} s''}{s' = s''} \tag{2.4}$$

Given the basic structure and its corresponding requirements on the time behavior of a TLTS, we can now describe the behavior of an ESTS in terms of a TLTS. The behavior of an ESTS $\mathcal{A}$ is defined by its *interpretation* $[\![\mathcal{A}]\!]_{q_0}$ as TLTS, where q_0 is the used initial configuration. In the resulting TLTS a state $s \in S$ represents a configuration $q \in Q$ and a label $l \in L$ represents a parameterized label $\mu \in \Sigma$.

The interpretation of an ESTS by a TLTS follows the rules shown in Equation (2.5) to (2.8), where the predicate $\mathbf{ct}(q)$ indicates if the given configuration q has an enabled completion transition.

$$\frac{(q, \mu, \varphi, \rho, p, d, q') \quad \mu \in \Sigma_i \quad \neg\mathbf{ct}(q)}{s \xrightarrow{\mu} s' \quad s' \xrightarrow{d} s''} \tag{2.5}$$

The interpretation of an input is shown in Equation (2.5), where ς is the signal parameter valuation leading to multiple transitions in the TLTS due to the enumeration of the possible values and to a single transition if $\varsigma = \emptyset$. The term $\neg\mathbf{ct}(q)$ ensures that the input can only be executed if no enabled completion transition exists and prevents the interruption of a completion step.

$$\frac{(q, \mu, \varphi, \rho, p, d, q') \quad \mu \in \mathbb{N}_1 \quad \mathbf{n}_{min}(q) \quad \neg\mathbf{ct}(q)}{s \xrightarrow{n_r} s'} \tag{2.6}$$

Equation (2.6) shows the mapping rule for the delay transitions, where $n_r = c - n$ is the remaining time due to the execution of the delay transition. In this equation the clock update $\zeta' = \xi(\zeta)$ includes the elapsed time until the execution of the delay transition n and its execution duration d, where each clock of a timing group having $t \in T_g$ is updated by $c' = c + n + d$.

In this equation only the delay transitions with the smallest delay are considered, which is denoted by the function $\mathbf{n}_{min}(q)$. This requirement ensures a deterministic behavior if multiple delay transitions having different delays are enabled. The execution of a delay transition is also prevented if an enabled completion transition exists, which is achieved by the predicate $\neg\mathbf{ct}(q)$.

$$\frac{(q, \mu, \varphi, \rho, p, d, q'') \quad \mu \in \Sigma_o \cup \{\tau\}}{s \xrightarrow{d} s' \quad s' \xrightarrow{\mu} s''} \tag{2.7}$$

The interpretation of output- and unobservable-transitions is shown in Equation (2.7), where both transition types can be directly represented by a TLTS and need no further processing.

$$\frac{(q, \mu, \varphi, \rho, p, d, q') \quad \mu = \gamma}{s \xrightarrow{d} s'} \tag{2.8}$$

A special case are completion transitions, which do not have a corresponding representation in a TLTS. For this reason the semantics is expressed in terms of the available labels, where a completion transition can be represented as delay transition, where the delay equals the transition execution duration as shown in Equation (2.8). This ensures that the created delay transition is executed immediately as it was required for enabled completion transitions.

The presented ESTS allows to describe an UML SM by conserving its actions and guards. For this reason several syntactic and semantic elements have been added to the STS introduced by Frantzen et. al [13]. The expressiveness of an ESTS is similar to a TLTS [14], which is a superset of the semantics provided by an UML State Machine.

2.1.3. Properties

In order to define the class of ESTSs considered in this work, we identify a list of properties first. A configuration is Λ-*complete* if for every parameterized label $\mu \in \Lambda$ at least one transition

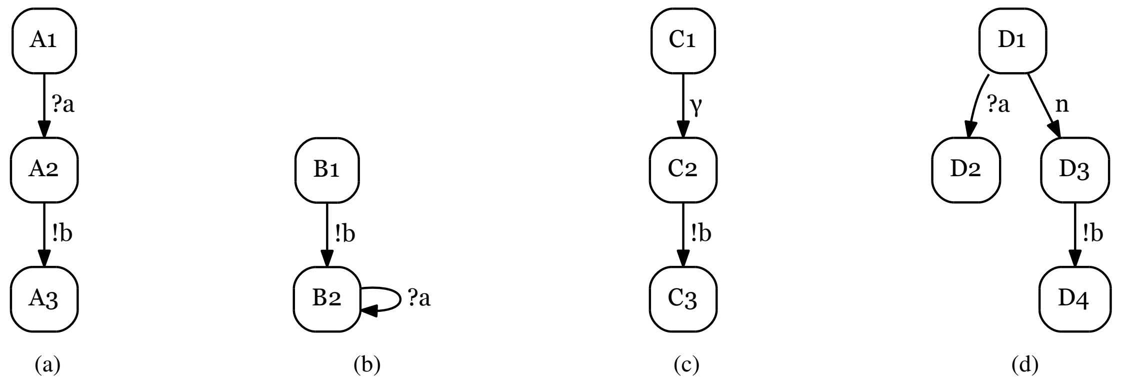


Figure 2.2.: Quiescence and deadlock states.

is enabled, where $\Lambda \subseteq \Sigma$. It is *complete* if it is Σ-complete and has *disjunct guards* if for a parameterized label only one transition can be enabled. A configuration has a *unique output* if it has disjunct guards, is Σ_o-complete and has only output transitions. This requirement reduces the capabilities of an ESTS essentially to the those of an Extended Finite State Machine (EFSM), where after an input exactly one output is possible. We define quiescence similar to the definition used in [14], as shown in Definition 2.8.

Definition 2.8 (Quiescence)
A configuration is quiescent if it cannot produce an output now or in the future, without receiving an input first. □

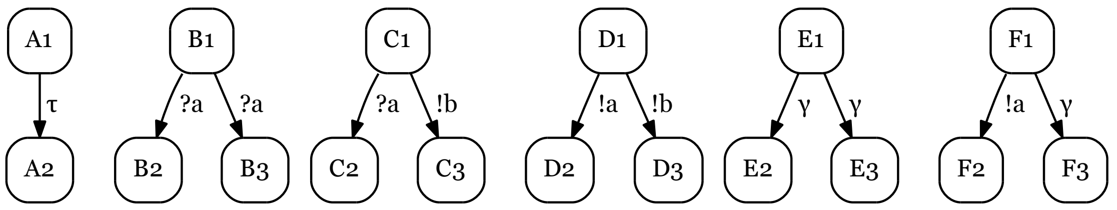
We denote a quiescent configuration as $\bar{q} \in \bar{Q}$, where $\bar{Q} \subseteq Q$ is the set of all quiescent configurations. Corresponding to the quiescence definition completion transitions are treated similar as delay transitions, which means that they can be executed before an output transition and prevent a configuration to be quiescent. A special case of a quiescent configuration is a deadlock, which does not accept an input in addition.

Example 2.2 (Quiescence and Deadlock)
In Figure 2.2 four ESTSs examples are presented, where only enabled transitions are shown and guards and actions were for simplicity omitted in the presentation.

In these examples the states *A1*, *A3*, *B2*, *C3*, *D2* and *D4* are quiescent, because the cannot produce an output before an input has been provided. In addition the states *A3*, *C3*, *D2*, *D4* are also in a deadlock, because they are quiescent and cannot receive any input. □

An ESTS is *uniquely prioritized* if it does not contain two transitions $t_1, t_2 \in T$ with their respective priorities $p_1, p_2 \in \mathfrak{P}$ such that $p_1 = p_2$ and has *no forced inputs*, which means that it is always possible to let time pass before an input is provided.

In this work we only consider deterministic ESTSs, which are uniquely prioritized, only contain static delays, do not contain unobservable transitions and where all configurations have dis-



(a) τ (b) Input (c) IO Conflict (d) Output (e) Completion (f) OC Conflict

Figure 2.3.: ESTS non-determinism examples.

junct guards. In addition we require that all quiescent configurations are Σ_i-complete and the non-quiescent configurations can only have a unique output.

Example 2.3 (Non Determinism)
Several examples of non-determinisms are shown in Figure 2.3, where the non-determinism of each example has a different cause. The precise meaning of non-determinism is not homogeneously defined and is used differently in the literature. However, in this example we want to provide a short overview of different non-determinism types.

The non-determinism in 2.3a is caused by the unobservable transition τ and 2.3b leads to two different states for the same input. The example shown in 2.3c shows an input/output conflict, where it is unclear if the output should be produced or it should be waited for the input first. In 2.3d two different outputs can be produced, where it is unknown a priory which one will be selected. This scenario is for example the result of an underspecification, where the details allowing a deterministic selection of the produced output is abstracted away. Example 2.3e is similar to 2.3b, where the resulting state is unknown. Finally in 2.3f shows an ESTS specific case, where it is unclear if the output should be produced. □

2.1.4. Paths and Traces

A path of an ESTS $\pi = q_1\mu_1 q_2\mu_2 \dots \mu_{n-1}q_n$ is an alternating sequence of configurations $q \in Q$ and parameterized labels $\mu \in \Sigma$ and $\mathbf{paths}(\mathcal{A})$ is the set of all paths of an ESTS $\mathcal{A}$. The concatenation of two paths of the same ESTS $\pi_1 = q_1\mu_1 \dots q_n$ and $\pi_2 = q'_1\mu'_1 \dots q'_m$ is given by $\pi_1 \circ \pi_2$, which leads to $q_1\mu_1 \dots q_n\mu'_1 \dots q'_m$ and it is required that $q_n = q'_1$.

Definition 2.9 (Accepted Path)
A path $\pi = q_1\mu_1 \dots q_n$ is accepted in $F \subseteq Q$, if its final configuration $q_n \in F$. □

A path is a *loop* if its final configuration equals its initial configuration meaning that $q_1 = q_n$ or similarly it is accepted in $\{q_1\}$. We use the term loop also for transitions, which source- and target-state are the same.

Definition 2.10 (Completion Paths)
A completion path is a path, which is accepted in the set of quiescent configurations $\bar{Q}$. □

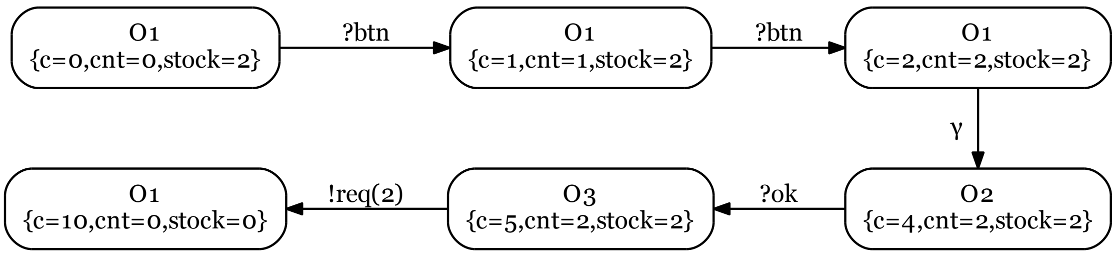


Figure 2.4.: Completion path π_O of the Order Component.

A completion path is denoted as $\vec{\pi} \in \mathbf{cpaths}(\mathcal{A})$, where $\mathbf{cpaths}(\mathcal{A}) \subseteq \mathbf{paths}(\mathcal{A})$ is the set of completion paths of an ESTS $\mathcal{A}$. A trace $\sigma = d_1\mu_1 d_2\mu_2 \ldots d_n\mu_n$ is a sequence of delays and parameterized in- and outputs $\mu \in \Sigma$ and the set of all traces of an ESTS $\mathcal{A}$ is given by $\mathbf{traces}(\mathcal{A})$. We use $\sigma = \mathbf{proj}_F(\pi)$ to denote the projection of parameterized labels of the path π, which abstracts the configurations and leads to a sequence of the remaining labels in the set F. The delays of a trace are calculated as the time difference between the connected configurations of the respective transition. For this reason a path projection has the same structure as a trace of a TLTS, which can be seen by the interpretation rules, and therefore can be interchanged.

Since we are interested in testing the observable behavior of the SUT – meaning to check the produced output for given inputs – the absence of an output has to be made explicit. This is done by the suspension $\Delta(\mathcal{A})$ of an ESTS $\mathcal{A}$, where self looping transitions with the quiescence label δ are added to every quiescent state. These transitions are called quiescent transitions and denote the desired absence of an output.

Definition 2.11 (Suspension)
The suspension $\Delta(\mathcal{A})$ of an ESTS $\mathcal{A}$ is an ESTS $\langle S, L^\delta, A, P, T^\delta, G, q_0\rangle$, where $L_o^\delta = L_o \cup \{\delta\}$ is the set of output labels of $\mathcal{A}$ increased by the quiescence label δ, $L^\delta = L_i \cup L_o^\delta \cup \mathbb{N}_1 \cup \{\gamma\}$ is the set of labels, $T^\delta = T \cup \{t^\delta\}$ is the extended transition relation and $t^\delta = (s, l, \varphi^\delta, \mathtt{id}, 0, 0, s')$. The remaining elements are taken over unchanged. □

The guard φ^δ of a quiescent transition t^δ is defined in such a way, that it evaluates to true if no output- or completion-transition is enabled for the given quiescent configuration $\bar{q} \in \bar{Q}$ as shown in Equation (2.9).

$$\varphi^\delta = \neg\left(\bigvee_{t\in T} \varphi : l \in L_o \cup \{\gamma\} \wedge q = \bar{q}\right) \tag{2.9}$$

The suspension is defined on deterministic ESTSs, but due to the similar structure and the interpretation defined in Section 2.1.2 it can be interchanged with a TLTS meaning that $[\![\Delta(\mathcal{A})]\!]_q = \Delta([\![\mathcal{A}]\!]_q)$.

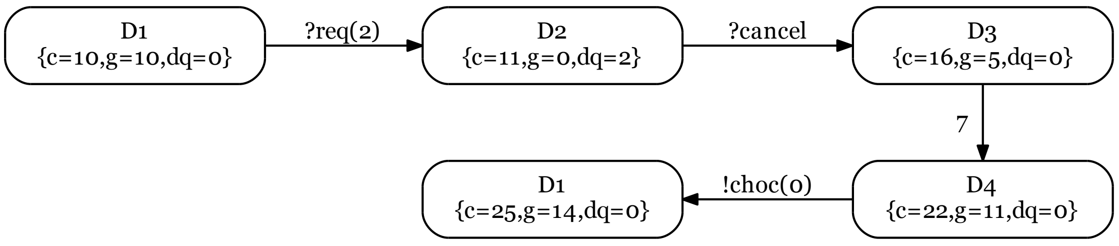


Figure 2.5.: Completion path $\pi_{\mathcal{D}}$ of the Delivery Component.

Example 2.4 (Paths and Traces)
In Figure 2.4 and Figure 2.5 a completion path of the Order Component $\pi_{\mathcal{O}}$ and the Delivery Component $\pi_{\mathcal{D}}$ of the vending machine explained in Example 2.1 are shown, where the path in the Delivery Component describes the reception of the input created by the path in the Order Component.

The nodes in these graphs represent a configuration, where the name of the state is written in the first line followed by the clock and attribute valuation in curly brackets. The ESTS clock c^* is denoted as c and the timing group clock in the Delivery Component as g. The edges show the parameterized labels, where we show the value of the signal parameters in round brackets after the label.

The corresponding traces to these paths are $0 \cdot \textit{?btn} \cdot 1 \cdot \textit{?btn} \cdot 1 \cdot \gamma \cdot 2 \cdot \textit{?ok} \cdot 6 \cdot \textit{!req(2)}$ and $0 \cdot \textit{?req(2)} \cdot 4 \cdot \textit{?cancel} \cdot 2 \cdot 7 \cdot 4 \cdot \textit{!choc(0)}$, where the relative time differences between the parameterized labels are shown. The delay of 6 in the first trace is the sum of 1 and 5, which are the transition execution durations of transition $(\textit{O2}, \textit{?ok}, true, \texttt{id}, 0, 1, \textit{O3})$ and $(\textit{O3}, \textit{!req}, \textit{qnt == cnt}, \textit{stock=stock-cnt;cnt=0;}, 0, 5, \textit{O1})$, respectively.

The timing group clock g in the path of the Delivery Component increases linear with time and is set to zero by a clock reset transition like $(\textit{D1}, \textit{?req}, \textit{true}, \textit{dq = qnt;}, 0, 1, \textit{D2})$ in this example. This transition causes a clock reset in the Delivery Component path shown in Figure 2.5, where the value of the clock g in the configuration with state *D2* becomes zero. □

2.2. Composition

In this section the behavior of a *system* is described, which consists of multiple *components*, which behaviors are described by ESTSs. The system components communicate with each other via signals defined in the corresponding ESTSs. For this reason we introduce the composition of ESTSs, which uses a global queue Γ to ensure a deterministic behavior of the system and to maintain the completion steps. In addition we also define the composition based on TLTS, where *full interleaving* between the TLTS is allowed meaning that all possible executions are contained and completion steps are neglected.

Based on these systems we define the conformance relation in order to verify if the SUT is correct with respect to a given specification provided as ESTS. The verification is performed by the execution of test cases, which is also shown in this section. In addition the possible observations

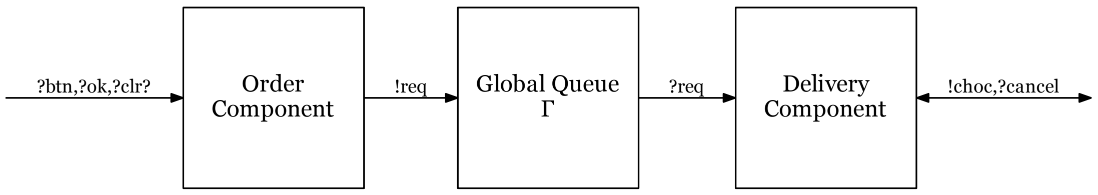


Figure 2.6.: Architecture of the vending machine.

of a system are defined, which distinguishes between signals used for communication between the ESTSs and those observable during testing.

Example 2.5 (System Architecture)
The system architecture used for the composition of the chocolate vending machine components is shown in Figure 2.6, where the Order Component, the Delivery Component and the Global Queue is shown. The global queue is used to synchronize the communication between the components and ensures a deterministic communication. □

2.2.1. ESTS Composition

The behavior of a system consisting of ESTSs is defined by their composition, where their communication is synchronized. A synchronous communication means that two ESTSs proceed together if one ESTS sends and the second ESTS receives the corresponding signal at the same time. In process algebra the synchronization of two processes p and q is usually denoted as $p||q$, which was also used for Labeled Transition Systems (LTSs) [25] and will also be used similarly in this approach for the composition of ESTSs. In order to maintain the distinction of the synchronized ESTS variables, they are decorated with the corresponding names like $\Sigma^{\mathcal{A}}$ for all parameterized labels of ESTS $\mathcal{A}$.

The synchronization of two ESTSs $\mathcal{A}$ and $\mathcal{B}$ is denoted $\mathcal{A}||_E\mathcal{B}$ and follows the Rules (2.10) to (2.14). The result of the synchronization is an ESTS $\mathcal{AB}$, where $L_i^{\mathcal{AB}} = L_i^{\mathcal{A}} \cup L_i^{\mathcal{B}}$ is the set of input labels $L_o^{\mathcal{AB}} = L_o^{\mathcal{A}} \cup L_o^{\mathcal{B}}$ is the set of output labels $S^{\mathcal{AB}} = S^{\mathcal{A}} \times S^{\mathcal{B}}$ is the set of states and $V^{\mathcal{AB}} = V^{\mathcal{A}} \cup V^{\mathcal{B}}$ is the set of variables.

The set of labels is $L^{\mathcal{AB}} = L_i^{\mathcal{AB}} \cup L_o^{\mathcal{AB}} \cup \mathbb{N}_1 \cup \{\tau, \gamma\}$ and it is required that $S^{\mathcal{A}} \cap S^{\mathcal{B}} = \emptyset$, $L_{io}^{\mathcal{A}} \cap L_{io}^{\mathcal{B}} = \emptyset$ and $V^{\mathcal{A}} \cap V^{\mathcal{B}} = \emptyset$. Correspondingly $Q^{\mathcal{AB}}$ is the set of configurations and $T^{\mathcal{AB}}$ is the transition relation being the minimum set fulfilling the following rules:

$$\frac{q_{\mathcal{A}} \xrightarrow{\mu} q'_{\mathcal{A}} \quad \mu \in (\Sigma_i^{\mathcal{A}} \cup \mathbb{N}_1) \setminus \Sigma_o^{\mathcal{B}} \quad q_{\mathcal{A}} \in \bar{Q}^{\mathcal{A}} \quad q_{\mathcal{B}} \in \bar{Q}^{\mathcal{B}}}{(q_{\mathcal{A}}, q_{\mathcal{B}}) \xrightarrow{\mu} (q'_{\mathcal{A}}, q_{\mathcal{B}})} \tag{2.10}$$

Rule (2.10) states that an input- or delay-transition in $\mathcal{A}$, which is not in $\mathcal{B}$, can only be executed if both ESTSs are in a quiescent configuration. This rule is used to maintain the uninterrupted execution of a completion step.

$$\frac{q_{\mathcal{A}} \xrightarrow{\mu} q'_{\mathcal{A}} \quad \mu \in (\Sigma_o^{\mathcal{A}} \cup \{\tau, \gamma\}) \setminus \Sigma_i^{\mathcal{B}}}{(q_{\mathcal{A}}, q_{\mathcal{B}}) \xrightarrow{\mu} (q'_{\mathcal{A}}, q_{\mathcal{B}})} \tag{2.11}$$

In contrast to Rule (2.10) a completion-, unobservable- or output-transition not in $\mathcal{B}$ can always be executed independent of quiescence.

$$\frac{q_{\mathcal{B}} \xrightarrow{\mu} q'_{\mathcal{B}} \quad \mu \in (\Sigma_i^{\mathcal{B}} \cup \mathbb{N}_1) \setminus \Sigma_o^{\mathcal{A}} \quad q_{\mathcal{A}} \in \bar{Q}^{\mathcal{A}} \quad q_{\mathcal{B}} \in \bar{Q}^{\mathcal{B}}}{(q_{\mathcal{A}}, q_{\mathcal{B}}) \xrightarrow{\mu} (q_{\mathcal{A}}, q'_{\mathcal{B}})} \tag{2.12}$$

$$\frac{q_{\mathcal{B}} \xrightarrow{\mu} q'_{\mathcal{B}} \quad \mu \in (\Sigma_o^{\mathcal{B}} \cup \{\tau, \gamma\}) \setminus \Sigma_i^{\mathcal{A}}}{(q_{\mathcal{A}}, q_{\mathcal{B}}) \xrightarrow{\mu} (q_{\mathcal{A}}, q'_{\mathcal{B}})} \tag{2.13}$$

The Rules (2.12) and (2.13) are similar to the Rules (2.10) and (2.11), but allow the execution of transitions not defined in $\mathcal{A}$ instead of $\mathcal{B}$.

$$\frac{q_{\mathcal{A}} \xrightarrow{\mu} q'_{\mathcal{A}} \quad q_{\mathcal{B}} \xrightarrow{\mu} q'_{\mathcal{B}} \quad \mu \in \Sigma_{io}^{\mathcal{A}} \cap \Sigma_{io}^{\mathcal{B}}}{(q_{\mathcal{A}}, q_{\mathcal{B}}) \xrightarrow{\mu} (q'_{\mathcal{A}}, q'_{\mathcal{B}})} \tag{2.14}$$

Rule (2.14) shows the synchronized execution of the transitions in both ESTSs in case their labels are equal according Definition 2.12.

Definition 2.12 (Equality)
Two parameterized labels $(l_o^1, \varsigma^1) \in \Sigma^1$ and $(l_o^2, \varsigma^2) \in \Sigma^2$ are equal if they have the same labels $l^1 = l^2$ and parameter valuations $\varsigma^1 = \varsigma^2$. Two traces are equal if for all $i = 1, \ldots, m$ the parameterized labels $(l_o^1, \varsigma^1)_i$ and$(l_o^2, \varsigma^2)_i$ are equal, where m is their number of parameterized labels. □

The reason for the separate treatment of input- and delay-transitions in Rules (2.10) and (2.12) instead of treating them like output- or completion-transitions in Rule (2.11) and (2.13) is to prevent full interleaving. Due to the restrictions to quiescent configurations for input- and delay-transitions, they cannot be processed before an output transition.

During the composition also new timing groups have to be created, where for each timing group in the ESTSs $\mathcal{A}$ and $\mathcal{B}$ a timing group $g^{\mathcal{AB}} \in G^{\mathcal{AB}}$ in the composition ESTS $\mathcal{AB}$ with $S_g^{\mathcal{AB}} = S_g^{\mathcal{A}} \times S_g^{\mathcal{B}}$ is added. The sets of delay- and reset-transitions of each created $g^{\mathcal{AB}}$ contain the transitions created by the Rules (2.10) to (2.14) for the corresponding delay- and reset-transitions in $g^{\mathcal{A}}$ and $g^{\mathcal{B}}$.

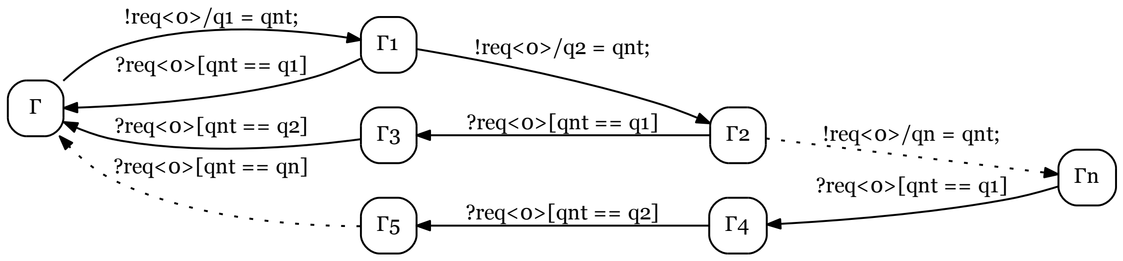


Figure 2.7.: Global queue example for the chocolate vending machine.

Since a system contains ESTSs communicating via a global queue Γ, we first show the structure of the queue. The global queue Γ is an ESTS, where $L^\Gamma = (L_i^\mathcal{A} \cap L_o^\mathcal{B}) \cup (L_i^\mathcal{B} \cap L_o^\mathcal{A})$ is the set of all labels used for communication between the ESTSs and S^Γ is the set of states containing a state for every possible order of signals in L^Γ which can be produced by the communicating ESTSs. Since Γ is only used for synchronization purposes of ESTSs it does not contain unobservable-, completion- or delay-transitions and all transition guards are true. The set of attributes $A^\Gamma = P^\mathcal{A} \cup P^\mathcal{B}$ of the queue Γ contains an attribute for every signal parameter for temporary storage and the set $P^\Gamma = P^\mathcal{A} \cup P^\mathcal{B}$ contains the signal parameters of both ESTSs.

Example 2.6 (Global Queue Example)
In the case of the chocolate vending machine the Order Component $\mathcal{O}$ sends an order request signal *req* to the Delivery Component $\mathcal{D}$, where the delivery quantity is specified by the signal parameter *qnt*. The signals of the global queue are calculated by $L^\Gamma = (L_i^\mathcal{O} \cap L_o^\mathcal{D}) \cup (L_i^\mathcal{D} \cap L_o^\mathcal{O})$, which equals $(\{\mathit{btn,clr,ok}\} \cap \{\mathit{choc}\}) \cup (\{\mathit{req},\ \mathit{cancel}\} \cap \{\mathit{req}\})$ and finally leads to $L^\Gamma = \{\mathit{req}\}$.

The corresponding global queue is shown in Figure 2.7, where the dashed lines indicate an arbitrary long sequence of intermediate states. The shown queue is therefore unbound and does not limit the number of queued signals. □

In the remainder $(\mathcal{A}\|_E\Gamma)\|_E\mathcal{B}$ is abbreviated by $\mathcal{A}\|_\Gamma\mathcal{B}$ and we use $M = \mathcal{A}_1\|_\Gamma\mathcal{A}_2\|_\Gamma \dots \|_\Gamma\mathcal{A}_m$ for a *system* consisting of $m \geq 2$ ESTSs communicating via the queue Γ.

2.2.2. TLTS Composition

Since the composition of ESTSs is only used to describe the system behavior, we introduce the composition of TLTSs in addition to define the test execution. The main difference between these two compositions is the treatment of full interleaving, which is prevented for synchronization $\|_E$ and allowed for $\|_L$, where $\|_L$ denotes the synchronization of two TLTSs following the Rules (2.15) to (2.17).

The synchronization $\mathcal{A}\|_L\mathcal{B}$ of two TLTSs $\mathcal{A}$ and $\mathcal{B}$ leads to a TLTS $\mathcal{AB}$, where $L_i^{\mathcal{AB}} = L_i^\mathcal{A} \cup L_i^\mathcal{B}$ is the set of input labels, $L_o^{\mathcal{AB}} = L_o^\mathcal{A} \cup L_o^\mathcal{B}$ is the set of output labels, $L^{\mathcal{AB}} = L_i^{\mathcal{AB}} \cup L_o^{\mathcal{AB}} \cup \mathbb{N}_0 \cup \{\tau\}$

is the set of labels and $S^{\mathcal{AB}} = S^{\mathcal{A}} \times S^{\mathcal{B}}$ is the set of states.

$$\frac{s_{\mathcal{A}} \xrightarrow{\mu} s'_{\mathcal{A}} \quad \mu \in L^{\mathcal{A}}_{\tau} \setminus L^{\mathcal{B}}_{io}}{(s_{\mathcal{A}}, s_{\mathcal{B}}) \xrightarrow{\mu} (s'_{\mathcal{A}}, s_{\mathcal{B}})} \tag{2.15}$$

$$\frac{s_{\mathcal{B}} \xrightarrow{\mu} s'_{\mathcal{B}} \quad \mu \in L^{\mathcal{B}}_{\tau} \setminus L^{\mathcal{A}}_{io}}{(s_{\mathcal{A}}, s_{\mathcal{B}}) \xrightarrow{\mu} (s_{\mathcal{A}}, s'_{\mathcal{B}})} \tag{2.16}$$

$$\frac{s_{\mathcal{A}} \xrightarrow{\mu} s'_{\mathcal{A}} \quad s_{\mathcal{B}} \xrightarrow{\mu} s'_{\mathcal{B}} \quad \mu \notin \{\tau\}}{(s_{\mathcal{A}}, s_{\mathcal{B}}) \xrightarrow{\mu} (s'_{\mathcal{A}}, s'_{\mathcal{B}})} \tag{2.17}$$

The difference between these two synchronizations might lead to different system behaviors for the same models, which means that $\mathcal{A}\|_E\mathcal{B}$ is not necessarily equal to $[\![\mathcal{A}]\!]_{q_0}\|_L[\![\mathcal{B}]\!]_{q_0}$. The reason is that the composition $\|_E$ ensures the defined execution of the completion step, whereas the LTS composition $\|_L$ does not.

The result of the shown synchronization rules for ESTS and TLTS as well contains the labels used between the transition systems and those used as system inputs and outputs. We refer to the former as *internal* and the latter as *external* signals. This means during black box testing we are interested in checking if the external signals conform to a specification while neglecting the internal ones. This means for a single ESTS that all input and output labels are external, because none of them is used for internal communication. Given an ESTS composition $\mathcal{A}\|_\Gamma\mathcal{B}$ the set of internal signals $\check{L}_{io}$ is represented by the label set of the queue L^Γ, while the set of external signals $\hat{L}_{io}$ equals $(L^{\mathcal{A}}_{io} \cup L^{\mathcal{B}}_{io}) \setminus L^\Gamma$.

Example 2.7 (External and Internal Signals)
The vending machine example described in Example 2.1 contains external- and internal-signals, where the external signals define the possible interactions with the customer and the internal signals are used to pass orders between the components. This means in this example that $\hat{L}_{io} = \{$*?btn,?ok,?clr,?cancel,!choc*$\}$ are the external- and $\check{L}_{io}\{$*?req,!req*$\}$ are the internal-signals, respectively. □

2.2.3. Conformance

In the remainder of this work we use the set of *suspension traces* $\mathbf{straces}(\mathcal{A})$ of the deterministic ESTS $\mathcal{A}$, which contains a trace $\mathbf{proj}_F(\pi)$ for all $\pi \in \mathbf{cpaths}(\Delta(\mathcal{A}))$, where $F = \hat{\Sigma}_{io} \cup \mathbb{N}_1 \cup \{\delta\}$ is the union of parameterized labels corresponding to the external signals $\hat{L}_{io}$ of the ESTS $\mathcal{A}$, its delay- and quiescence-labels. The SUT is assumed that it can be represented by a deterministic ESTS $\mathcal{I}$, which is correct with respect to a given specification in form of an ESTS $\mathcal{A}$ if the conformance relation shown in Equation (2.18) holds.

$$\forall \sigma \in \mathbf{straces}(\mathcal{A}), \exists \sigma' \in \mathbf{straces}(\mathcal{I}) : \sigma = \sigma' \tag{2.18}$$

For the presented conformance relation we assume only deterministic ESTSs and allow partial specifications. Therefore it is similar to $\mathbf{tioco}_{\mathrm{M}}$ discussed in [14], where a timeout M has to elapse in order to detect the absence of an output. In this approach the amount of time M can be defined on a transition basis due to the introduced transition execution time, which allows a refined specification but does not change the concept.

2.2.4. Test Cases

In this approach test cases are created from a path or system path, where all input signal parameters and attribute values are already known. In order to create a test case we add input transitions leading to a **fail** state to every state in the test case for every output not already defined in the path. This ensures that the test case detects unintended outputs, which causes the test verdict **fail**. A $\mathbf{pass}$ state is added by replacing the last state in the path corresponding to a test generation step.

Definition 2.13 (Test Case)
A test case is an ESTS $\mathcal{TC} = \langle S^{\mathcal{TC}}, L^{\mathcal{TC}}, A^{\mathcal{TC}}, P^{\mathcal{TC}}, T^{\mathcal{TC}}, G^{\mathcal{TC}}, q_0^{\mathcal{TC}} \rangle$, where $S^{\mathcal{TC}} = S \cup \{\mathbf{pass}, \mathbf{fail}\}$ is a set of states extended with the test verdicts $\mathbf{pass}$ and $\mathbf{fail}$ and $L^{\mathcal{TC}} = L_i^{\mathcal{TC}} \cup L_o^{\mathcal{TC}} \cup \mathbb{N}_1$ is the set of mirrored labels. □

The states $\mathbf{pass}$ and $\mathbf{fail}$ are sink states, which ensures that the execution of the test case is stopped if one them is reached. Mirrored in this context means that the input labels $L_i^{\mathcal{TC}} = L_o^{\mathcal{I}} \cup \{\delta\}$ are equal to the output including the quiescence labels and $L_o^{\mathcal{TC}} = L_i^{\mathcal{I}}$ the input labels are equal to the output labels of the SUT $\mathcal{I}$. The remaining elements of the test case are similar to those of the SUT.

As usual the SUT is rejected by the test case if it reaches a state **fail** and it behaves as expected if a $\mathbf{pass}$ state is reached during the execution. The quiescent states of a test case are $L_i^{\mathcal{TC}}$-complete to be able to react to every output produced by the SUT.

Due to the restriction on a deterministic ESTS a corresponding test case has only one controllable at a time, which means that only one output in the test case – which corresponds to an input of the SUT – is allowed at a time. For this reason the test case is a sequence leading to a $\mathbf{pass}$ state without branches to other states than **fail**. Therefore the test case explicitly states all inputs and outputs, which is required by the conformance relation shown in Equation (2.18).

A test case $\mathcal{TC}$ is executed on a SUT $\mathcal{I}$ by the composition $[\![\mathcal{I}]\!]_{q_0} \|_L [\![\mathcal{TC}]\!]_{q_0}$ of their respective interpretation as TLTS. The reason is that in this case the special rules for ESTSs ensuring the completion steps must not be applied.

During the interpretation of an ESTS as LTS the execution time is treated differently for inputs and outputs. Whereas for inputs a delay transition is created after the input as shown in Equation (2.5) it is the other way round for output transitions corresponding to Equation (2.7). Since the input- and output-labels are swapped for a test case, the position of the created delay transitions in its interpretation has to be swapped too. This means that for test cases the delay transition for an input is created before and for an output after the respective transition.

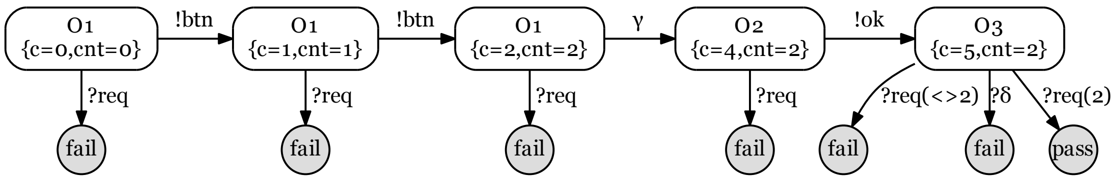


Figure 2.8.: Test Case

Example 2.8 (Test Case)
A test case for the vending machine is shown in Figure 2.8, which was created from the path of the Order Component shown in Figure 2.4. We omit the presentation of the attribute *stock*, which has the value 2 throughout the test case for simplicity. In this test case the output signal *!req* is expected from the SUT after the button has been pressed twice followed by an acknowledgment *?ok*. If the output of the SUT is received by the test case after the input sequence for the SUT has been provided, then the test cases has the verdict **pass** indicated by the correspondingly named state. If an output is produced by the SUT before it is allowed by the specification the test case reaches a **fail** state. □

A more detailed discussion on test cases and their properties can be found in [26], where non-deterministic systems based on LTSs are considered.

Chapter 3

Test Case Generation

Parts of this chapter are taken from ´An Extended Symbolic Framework for Systematic Test Case Generation´ which is an ongoing work with Bernhard K. Aichernig and Franz Wotawa, ´Test Sequence Generation from Communicating UML State Charts: An Industrial Application of Symbolic Transition Systems´ [20] which is joint work with Bernhard Peischl and ´Generation of executable test cases based on behavioral UML system models´ [19] which is joint work with Bernhard Peischl.

In this section the generation of test cases based on a set of communicating ESTSs to support test engineers and to improve the test development efficiency is described. The main focus of this approach lies on integration testing of functions, which are distributed over multiple hardware units connected via a network. The behavior of each function is specified by a set of ESTSs and serves as basis for the oracle calculation used in the generated tests.

Due to the state space explosion problem and the corresponding possibility of an infinite number of paths we consider only deterministic ESTSs for the test generation algorithm. The imposed restrictions allow a faster test case generation, whereas the descriptive power is still sufficient for the industrial context of this work.

Since we are interested in providing algorithms to improve the test development efficiency, the test goal still has to be defined. The test goal is expressed by a test purpose, which is basically a list of transitions. The test purpose definition is described in Section 3.1 which transitions have to be covered by a generated test case.

Given a test purpose, the generation of a test case is split into the following three steps:

1. **Structural search:** During the structural search symbolic paths describing a way through an ESTS are created. At this stage only the connections of the states via transitions is taken into account and guards or variable valuations are neglected. This leads to possibly infeasible paths, which in addition may depend on signals from other ESTSs in the system.
2. **Communication resolution:** The possible dependencies of an ESTS on the signals sent by another ESTS are resolved during the communication resolution. This is realized by an additional structural search in those ESTSs which can potentially send the desired signal. The paths of these structural searches are then inserted in the paths created in the first step.
3. **Input value calculation:** In order to create an executable test case, feasible input values have to be created for the symbolic paths generated in the steps before. This is achieved

by the creation of a path constraint, which consist of the guards and update actions of a symbolic path. The path constraint is a set of conjugated conditions, which can be passed to a constraint solver to calculate feasible input values.

A detailed description of the structural search is given in Section 3.2, the communication resolution is explained in Section 3.3 and the calculation of input values is discussed in Section 3.4.

3.1. Test Purpose Definition

A test purpose definition consists of a selection of transitions describing the test goal in terms of an ESTS. It has to be created by the test engineer in order to provide a formal definition of the desired SUT execution for the test generation algorithm.

The definition of a test purpose based on a LTS which can be processed by the tool TGV[18] is described in [27]. The usage of a symbolic test purpose based on an IOSTS is discussed in [28], where a similar approach is used in this work. However, the approach in this work is more restrictive, because we use transitions which also requires that a certain state has been reached before a specified label in the test purpose. This reduces the synchronous product of the specification and the test purpose used to specify the set of corresponding test cases.

The test purpose used in this work is split into an ordered list of *generation steps*, where each step describes which transition of the test goal has to be reached after another.

Definition 3.1 (Generation Step)
A generation step is a tuple $\langle I, E\rangle$ of two transition sets, where $I \subseteq T_M$ are the included transitions, $E \subseteq T_M$ are the excluded transitions and T_M is the union of ESTS transitions of a system M. □

The included transitions I define the goal of the generation step ν, where at least one of the included transitions has to be traversed by a corresponding test case. Given a system $M = \mathcal{A}_1 \|_\Gamma \ldots \|_\Gamma \mathcal{A}_m$ of ESTSs , in this approach it is required that all included transitions are from one ESTS of M and – since they are an integral part of a test definition – the set of included transitions is not empty. The transitions contained in the set of excluded transitions E are treated in the opposite way in comparison to the included transitions and therefore must not occur in a corresponding test case.

Definition 3.2 (Test Purpose)
A test purpose is an ordered list $(\nu_1, \nu_2, \ldots, \nu_m)$ of generation steps, where ν_m is a generation step $\langle I, E\rangle_m$ with index $m \in \mathbb{N}_1$. □

Example 3.1 (Test Purpose)
In Figure 3.1 a test purpose is shown, which consists of two generation steps having one included transition namely $(\mathit{O2}, \mathit{?ok}, \mathit{true}, \mathtt{id}, 0, 1, \mathit{O3})$ and $(\mathit{D2}, \mathit{?cancel}, \mathit{true}, \mathtt{id}, 0, 1, \mathit{D3})$, respectively. The first transition belongs to the Order Component and the second to the Delivery Component,

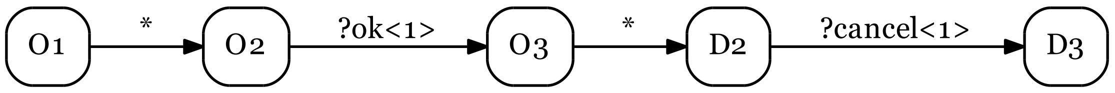


Figure 3.1.: Test Purpose

where the path of the Order Component $\pi_{\mathcal{O}}$ is shown in Figure 2.4 and the path of the Delivery Component $\pi_{\mathcal{D}}$ is depicted in Figure 2.5. Together they fulfill this test purpose, because both of the desired transitions are covered by the respective paths.

The test purpose starts from the state *O1*, which is the initial state defined in the initial configuration q_0 for the Order Component. The reason is that in the presented approach the test case generation always starts from the initial configuration. □

Given the formalized test goal in form of a test purpose, which has to be provided by a test engineer or domain expert, the generation of corresponding test cases is described in the following sections.

3.2. Structural Search

Structural search means in this approach the creation of paths through an ESTS, where only its structure is taken into account, which is given by its contained states and transitions. This means that during the structural search variable valuations are neglected and the guards and attribute update functions of the transitions on such a path have to be treated symbolically. Symbolically means that variable valuations are not available as concrete values, but their possible values are defined by the guards and update actions along the path.

Definition 3.3 (Symbolic Path)
A symbolic path $t_1 t_2 \ldots t_m$ is a sequence of ESTS transitions $t \in T$ with index $m \in \mathbb{N}_1$. □

The structural search is used to create a set of symbolic paths $\mathbf{sympaths}(\nu, s)$ for a given generation step ν, where the first state s_1 of each symbolic path $\psi \in \mathbf{sympaths}(\nu, s)$ has to be the given state $s \in S$. The concatenation of two symbolic paths of the same ESTS $\psi_1 = t_1 \ldots t_n$ and $\psi_2 = t'_1 \ldots t'_m$ is given by $\psi_1 \circ \psi_2$, which leads to $t_1 \ldots t_n t'_1 \ldots t'_m$ where it is required that the target vertex of t_n is equal to the source vertex of t'_1.

In addition it is required that each of these symbolic paths covers a transition in I and does not cover any of the transitions in E, which are defined in the given generation step ν. The search for a symbolic path stops after an included transition has been found, which means that any consecutive transitions are neglected.

Given that the excluded transitions are already taken into account during the structural search, the number of the resulting symbolic paths can be reduced in an early stage of the test generation process and can therefore also reduce the overall time needed for the test generation.

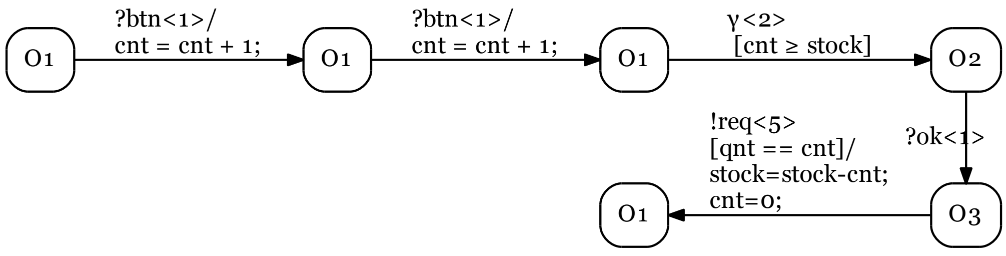


Figure 3.2.: Symbolic path ψ_O of the Order Component

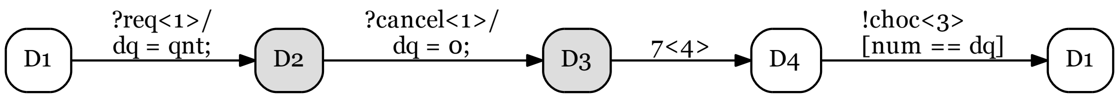


Figure 3.3.: Symbolic path $\psi_{\mathcal{D}}$ of the Delivery Component

Since loops can occur in an ESTS an infinite number of symbolic paths might exist, which would prevent the halting of the structural search algorithm. For this reason several search limitations were introduced to ensure a finite number of paths as described in Section 3.2.1. These limitations are a set of parameters passed to the search algorithm, which can be tailored to the specific needs of the SUT by this mechanism.

Example 3.2 (Symbolic Paths)
In Figure 3.2 and 3.3 the symbolic paths corresponding to the paths with concrete values depicted in Figure 2.4 and 2.5 of the chocolate vending machine are shown. □

3.2.1. Search Limitations

A limitation of the search and therefore of the number of symbolic paths which can be found, also limits the capabilities of the test case generation. This means on the one hand, that test cases might exist but will not be found due to these restrictions and on the other hand they can be used to tailor the generation process to reduce the considered state space and therefore to speedup the generation. In order to allow a SUT specific tailoring of the search, the following limitations are introduced in this work and can be applied to every test purpose step independently:

1. **Loop Number:** The loop number defines an upper bound for the enumeration of loops in an ESTS, which in other words is the number of iterations of the loop. The number of iterations of each transition in the symbolic path is tracked by a counter during the structural search. If the counter exceeds the provided loop number in $\mathbb{N}_0$, the search for further transitions is aborted for this path.

2. **Length:** The length property defines the maximum number of transitions of a symbolic path. If the maximum length is reached any further transitions are neglected and the path

is omitted if it does not cover any of the included transitions.

3. **Non-Target Exclusion:** This parameter has a boolean value and restricts the selection of the transitions taken into account leaving a given state. This means if the parameter non-target exclusion is true, then only transitions which are in the included transitions I of the generation step are considered and the remaining ones are neglected. This selection is only performed for states which have outgoing transitions in I. In any other case all transitions are taken into account during the search.

4. **Completion Loop Number:** The completion loop number has a similar effect compared to the loop number parameter, but limits the number of enumerations for loops consisting only of completion transitions. This means it provides an upper bound for the number of iterations of completion loops, which is necessary in order to prevent an infinite number of resulting symbolic paths. The reason is that during the structural search no ESTS parameter- nor attribute-valuations are available and therefore the guards cannot be evaluated, which could end such loops. However, the same amount as for the loop number could be used, but the introduction of this additional parameter allows a more fine grained control of the search.

3.2.2. Completion Step Generation

The generation of a completion step means the extension of the symbolic paths generated during the structural search by completion transitions, which might follow after an included transition was found. Since the last transition of a symbolic path is an included transition of the generation step, possibly following completion- or output-transitions are not taken into account yet. For this reason another structural search is performed to append the missing transitions.

In this second search – starting from the target vertex s'_m of the last transition t_m of a symbolic path in $\mathbf{sympaths}(\nu, s)$ – only completion- and output-transitions are allowed and the result of this search is a set of symbolic completion paths $\mathbf{sympaths}_c(\nu, s'_m)$. During this search the excluded transitions of the generation step ν are taken into account again, which prevents the occurrence of these transitions in the resulting symbolic completion paths.

After the generation of the symbolic completion paths for each symbolic path $\psi \in \mathbf{sympaths}(\nu, s)$, the set of symbolic paths $\mathbf{sympaths}(\nu, s)$ is updated as shown in Equation (3.1) and contains the concatenations of the symbolic path ψ with its symbolic completion paths.

$$\mathbf{sympaths}(\nu, s) := \{\psi \circ \psi_c \mid \psi \in \mathbf{sympaths}(\nu, s) \wedge \psi_c \in \mathbf{sympaths}_c(\nu, s'_m)\} \quad (3.1)$$

In the remainder of this work we refer with $\mathbf{sympaths}(\nu, s)$ only to its updated version and accordingly with symbolic paths to those extended by their completion steps. Since the structural search is performed only on one ESTS, the communication between the other components of a system M have to be incorporated as described in Section 3.3.

3.3. Communication Resolution

A symbolic path created during the structural search contains input- and output-transitions, which labels are either part of the internal- or external communication. This means in the case of internal communication that they are used for communication between the components of a system. Input- and output-transitions which are part of the internal communication of a symbolic path cause a dependency between the ESTS sending an internal signal and the one receiving it. These dependencies have to be resolved during the test case generation for systems containing multiple components.

In general the dependency resolution of a symbolic path is done by iterating through its transitions and the insertion of corresponding symbolic paths of "other" ESTSs. Other ESTSs are in this case those ESTSs of a system M to which the transition of the symbolic path – which dependencies shall be resolved – does not belong.

The inserted symbolic paths reflect the production of an output causing the execution of a transition of the symbolic path or the reaction to an output produced during the execution of the symbolic path. We say that a symbolic path *produces* a given output if it contains the corresponding transition labeled with the given output signal. A symbolic path producing an output signal in order to resolve an input dependency is called *sender path* and an inserted symbolic path reacting to a produced output is named *update path*.

The creation of sender- and update-paths is explained in Section 3.3.1 and Section 3.3.2, respectively.

3.3.1. Input Generation

In this section the creation of sender paths for an input transition of a symbolic path, which is part of the internal communication is explained. A sender path is a symbolic path of an ESTS producing a specified output and is created using the structural search as described in Section 3.2. This means that a sender path is in the set $\mathbf{sympaths}(\nu', s)$, where ν' is a generation step and s is a state of the ESTS in which the search is performed. The included transitions of the generation step ν' are the transitions producing the desired output and therefore depend on the input transition to be resolved.

Assuming we have a system M consisting of two components $\mathcal{A}$ and $\mathcal{B}$ and we have created a symbolic path $\psi^{\mathcal{A}}$ on component $\mathcal{A}$. Input generation means the search for sender paths in other system components as in this case $\mathcal{B}$. For this reason a symbolic path from the current state of $\mathcal{B}$ has to be created, which produces the according output signal defined by the input transition of $\psi^{\mathcal{A}}$.

Given a sender path for an input transition of a symbolic path, the insertion of the sender path into the symbolic path leads to system path.

Definition 3.4 (System Path)
A system path of a system M is a symbolic path consisting of concatenated symbolic paths from different components of M, where its states are tuples containing a state of each ESTS in M. □

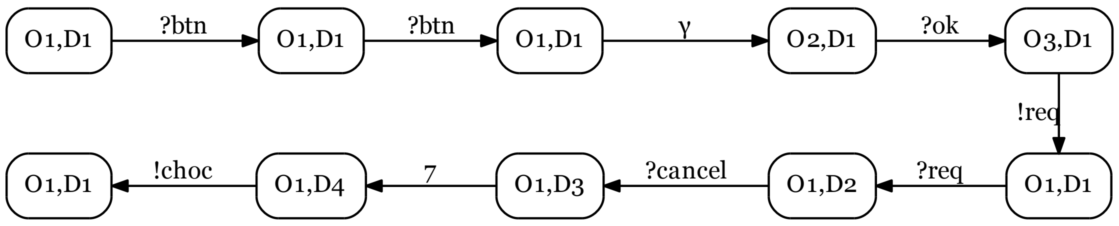


Figure 3.4.: System path η_{OD} of the vending machine.

The insertion of $\psi^{\mathcal{B}} = b_1 t_1^b \dots t_{k-1}^b b_k$ in $\psi^{\mathcal{A}} = a_1 t_1^a \dots t_{j-1}^a a_j$ at index $i \in \mathbb{N}_1$ is denoted by $\psi^{\mathcal{A}} \prec_i \psi^{\mathcal{B}}$, where $1 \le i \le j$ and η is the resulting system path shown in Equation (3.2).

$$\eta = (a_1 b_1) t_1^a \dots t_{i-2}^a (a_{i-1} b_1) t_1^b \dots t_{k-1}^b (a_{i-1}, b_k) t_i^a \dots t_{j-1}^a (a_j, b_k) \tag{3.2}$$

As shown in Equation (3.2), the $\psi^{\mathcal{B}}$ is inserted as a whole and its first transition is at position i after the insertion. This procedure has to be used in order to fulfill the requirement for uninterruptible completion steps.

Continuing the example above and assume that $\psi^{\mathcal{B}}$ is a sender path for an input transition of $\psi^{\mathcal{A}}$ at index i then the sender path is inserted before this transition by $\psi^{\mathcal{A}} \prec_i \psi^{\mathcal{B}}$. This ensures that the sender path produces the necessary output before it is processed in $\psi^{\mathcal{A}}$.

Example 3.3 (System Path)
The system path $\eta_{\mathcal{OD}} = \psi_{\mathcal{D}} \prec_1 \psi_{\mathcal{O}}$ is shown in Figure 3.4, where only the transition labels are shown for simplicity, $\psi_{\mathcal{O}}$ is the symbolic paths of the Order Component and $\psi_{\mathcal{D}}$ is the symbolic path of the Delivery Component. □

Since the sender path also considers completion steps, the transition producing the desired output is not necessarily the last transition of the sender path. Moreover the completion step of the sender path could contain additional output transitions, which can influence the behavior of the system and therefore have an impact on the symbolic path, as discussed in Section 3.3.2.

Due to the fact that a component can receive signals from multiple other components of a system M, it is also possible that multiple components could produce a sender path for an input transition. For this reason every component $m_S \in M_S$ has to be taken into account during the input resolution process, where $M_S \subseteq M$ is the subset of components of system M capable of producing a desired output.

This is achieved by the creation of a sender path set using the structural search of each $m_S \in M_S$, which are then united in the set of all sender paths Ψ_S. The sender paths in Ψ_S are then inserted in the symbolic path ψ at the position of the input transition i. During the insertion of the sender paths a set of system paths Ψ is created, which communication dependencies can be further resolved. In order to prevent an infinite loop in this algorithm, which can be caused by

ESTSs with mutual dependencies, an upper limit for resolution attempts of the same input was introduced.

The excluded transitions of a generation step ν can contain – in contrast to its included transitions – transitions of an ESTS of any system component. This means that the transitions considered during the structural search of the sender paths can also be restricted by a generation step. The generation step ν' used for the creation of sender paths contains therefore the same excluded transitions as ν, where the included transitions of the generation step ν were used for the creation of the symbolic paths initially. This approach allows for this reason a fine grained control of the test case generation algorithm.

Example 3.4 (Sender Path)
Given the symbolic paths $\psi_{\mathcal{O}}$ and $\psi_{\mathcal{D}}$ shown in Figure 3.2 and 3.3, respectively the path $\psi_{\mathcal{O}}$ is a sender path for the first transition $(\mathit{D1}, \mathit{?req}, true, \mathit{dq = qnt;}, 0, 5, \mathit{D2})$ in the symbolic path of the Delivery Component. □

3.3.2. State Updates

The creation of update paths has to be done in the case an output transition of a symbolic path is executed, which was not intentionally created for an input transition as shown in Section 3.3.1. The reason is that the produced outputs can change the state in another component on which the test case generation in general and the input generation in particular can depend later on. Therefore it is necessary to keep track of all state changes of the whole system in order to ensure correct test cases.

The reactions of other components are described by update paths, which are symbolic paths and created on the model receiving the output. The update paths are again created based on a structural search, but the resulting path set only contains paths where their first transition can receive the produced output. This means that this transition must have the corresponding input label. The remainder of an update path is its completion step and therefore only consists of completion- and output-transitions.

An output transition can only send to exactly one component of the system, because of the requirement of mutual exclusive label input- and output-sets of the components. However, due to the fact that a state can have multiple outgoing transitions labeled the same input signal, multiple paths have to be created during the state update. This is true even in the deterministic case, because during the structural search variable valuations are neglected which prevents the evaluation of guards. For this reason the selection of the transition with the guard evaluating to true cannot be done at this point and therefore all possibilities have to be considered.

After the set of update paths Ψ_U has been created, its contained paths are used to build a new set of system paths, where the update paths are inserted after the output production. This means for a transition at index i in the path $\psi^{\mathcal{A}}$ producing an output that a corresponding update path $\psi \in \Psi_U$ is inserted $\psi^{\mathcal{A}} \prec_{i+1} \psi$ at position $i + 1$.

Example 3.5 (State Update)
Assume the test purpose in Example 3.1 consists only of the first generation step with the included transition $(\mathit{O2}, \mathit{?ok}, true, \mathtt{id}, 0, 1, \mathit{O3})$ and $\psi_{\mathcal{O}}$ is the correspondingly created symbolic

path. In this case the output transition $(O3, !req, qnt == cnt, stock{=}stock\text{-}cnt;cnt{=}0;, 0, 5, O1)$ in ψ_O causes a state update in the Delivery Component. In this example the state update consists only of the transition $(D1, ?req, true, dq = qnt;, 0, 1, D2)$. □

3.3.3. Delay Transitions

In this section the treatment of delay transitions during the test case generation is explained. The main difference of delay transitions in comparison to input- and output-transitions in this case is, that delay transitions do not depend on another transition to be executed. This means that no transition with a corresponding input- or output-signal is needed for their execution. The reason is that a delay transition can become enabled only by the elapse of time, which advances in all components at the same time. Therefore a delay transition can become enabled in a component, which at that stage of the test case generation is not part of the communication resolution currently performed.

Therefore this behavior has to be considered during test case generation explicitly, because the delay transitions can change the behavior of the system. This is achieved by the insertion of the symbolic paths caused by the delay transition into the symbolic paths, which are currently resolved. These symbolic paths are built similar to the update paths, but start with a delay- instead of an input-transition and are created in the same component as the delay transition.

In this approach this time dependent behavior is taken into account during test case generation by time queues, where each time queue belongs to a symbolic path currently processed and contains all delay transitions which can become enabled by the elapse of time.

Definition 3.5 (Time Queue)
A time queue is a list $((t_1, d_1), (t_2, d_2), \ldots, (t_k, d_k))$ of tuples (t_j, d_j), where $t_j \in T$ is a delay transition, $d_j \in \mathbb{Z}$ is the remaining time until the delay transition becomes enabled, $\mathbb{Z}$ is set of integers, $k \in \mathbb{N}_1$ is the number of tuples and $1 \leq j \leq k$ is the list index. □

A time queue is ordered ascending by the remaining time d meaning that $d_j \leq d_{j+1}$ for all its contained tuples with index $1 \leq j \leq k - 1$. This order is maintained during the insertion and removal of elements from the time queue and ensures that the delay transition with the smallest remaining delay time is always at the first position with index $j = 1$.

The remaining delay time of a time queue entry d can according to its definition become negative. This is necessary because we do not consider true parallelism of the system components, but treat their execution as a sequence. This means that d can become negative if the path execution time of an inserted symbolic path is greater than the delay of the delay transition. However, this is implied by the used ESTS composition and does not influence the time queue order.

Definition 3.6 (Path Execution Time)
The path execution time is the time in $\mathbb{N}_0$ needed for the execution of a symbolic path. □

The path execution time ψ_d is basically the sum of the delays and the execution durations of its contained transitions. However, timing groups might influence the calculation of the path execution time as defined before, because transition executions within a timing group might be considered by the delay n of the corresponding delay transition. In this context the execution of a transition *within* a timing group means that the source- and target-state of the transition are in the timing group states S_g increasing the timing group clock c.

Since a delay transition becomes enabled if $c \geq n$, the transition executions within a timing group are taken into account twice when the transition delay n is added. For this reason the transition executions within the timing group have to be removed from the path execution time as shown in Equation (3.3), where ψ_d is the path execution time, d is the transition execution duration, ψ_n is the set of delay transitions contained in ψ, n is the transition delay and ψ_d^g is the execution duration spent within the corresponding timing group g.

$$\psi_d = \sum_{t \in \psi} d + \sum_{t \in \psi_n} (n - \psi_d^g) \tag{3.3}$$

Unfortunately it is not possible to calculate the path execution time by skipping the contained delays, because in the case that a transition delay n is greater than the execution duration ψ_d^g the time difference $(n - \psi_d^g)$ has to elapse before the execution of the delay transition.

In contrast if the transition execution duration ψ_d^g is greater than the transition delay n then ψ_d^g is used instead of their difference $(n - \psi_d^g)$ as shown in Equation (3.3). The reason is that the delay transition is executed immediately after the the last transition executed within g and therefore no additional time elapse for the delay transition has to be considered.

Example 3.6 (Path Execution Time)
The path execution time ψ_d of the symbolic path $\psi_{\mathcal{D}}$ of the Delivery Component shown in Figure 3.3 is the sum of the execution durations and delay transition timeouts minus the amount of time spent within a timing group. The states *D2* and *D3* of $\psi_{\mathcal{D}}$ are contained in the only timing group of the Delivery Component and are shown gray filled. The set $\psi_n = \{(D3, 7, true, \mathtt{id}, 0, 4, D4)\}$ contains therefore only the one transition connecting these two states.

In terms of Equation 3.3 this means that the sum of transition execution durations $\sum_{t \in \psi_{\mathcal{D}}} d = 1 + 1 + 4 + 3$ and sum of reduced transition delays $\sum_{t \in \psi_n} (n - \psi_d^g) = 7 - 1$, where $n = 7$ is the transition delay and $\psi_d^g = 1$ is the transition execution duration of transition with label *?cancel*. This leads to $\psi_d = 9 + 6 = 15$, which corresponds to the path $\pi_{\mathcal{D}}$ in Figure 2.5, where the input *?cancel* was provided at time $c = 15$. In this case 4 time units elapsed at the state *D2* before the input was provided. □

Since the path execution time can only be computed for a fully known path, which are not available during the dependency resolution, the delay transitions including their remaining delays have to be kept up to date during the generation process. The resolution process is performed stepwise, where each *resolution step* means the creation of symbolic paths for input generation or a state update as described above.

The time queue entries are updated at the beginning and the end of each resolution step during the communication resolution. At the beginning the timing group entries are removed and added corresponding to the current state of the ESTSs, where we denote by current state of an ESTS the target state of the last transition which has been already resolved. This means for example for a symbolic path consisting of five transitions from the same ESTS, where already three transitions are resolved, that the current state of this ESTS is the target state of the third transition. At the end of a resolution step the timing group entries are updated corresponding to the execution duration of the transition resolved in this resolution step.

During the first step, the delay transitions which source state is not contained in the current states are removed from the time queue, where the current states are a set containing the current state of each ESTS in the system. The reason for this time queue update is that the current state set could have been changed by the preceding resolution step. This means that delay transitions with a source state not in this set cannot become enabled and therefore must not be considered during the current resolution step.

However, the update of the current state set could allow other delay transitions to become enabled, which therefor have to be added to the time queue. Since the time queue is ordered by the remaining delays, the delay transitions to be added have to be inserted at the right position, which is determined by the corresponding delay.

The change of the current state set caused by the execution of a transition within the same timing group has to be treated specially, because the remaining time is passed to another delay transition belonging to the same timing group. The difference to the time queue update approaches described above is, that a removal and subsequent adding of transitions in the same timing group would cause a clock reset. A clock reset at this time means that the time advances since the time the timing group was entered first is lost and would lead to incorrect results. For this reason the delay transitions, which do not belong to the current state set anymore, are replaced by another delay transition of the same timing group which does. If the current state does not have an outgoing delay transition, then the remaining time still has to be kept until the timing group is left or the corresponding time queue entry is updated by an other delay transition.

As mentioned above, the second update of the time queue entries is done after the communication resolution. During this phase the remaining delays of each element in the time queue are decreased by the execution duration of the transition, which was processed in the current resolution step.

Due to the requirement for deterministic ESTSs for test case generation, the source state of a delay transition does only have outgoing transitions which can become enabled by an input or by the elapse of time. For this reason the execution of a delay transition can be caused by two reasons: Either more time elapses after the communication resolution than needed to execute the delay transition or a specific amount of time is elapsed intentionally to cause an execution.

In the first case the remaining time of the first time queue entry is checked whether it is already enabled, which would cause its execution. This means in this case that the remaining time of the delay transition is less or equal to zero and that its source state is in the current state set. The latter is ensured due to the existence of a corresponding time queue entry, because other delay transitions would have been removed already.

In the second case it is required that the transition of the first time queue entry is the same transition currently processed in the resolution step. In this case the execution of the delay transition is part of the test case and has to be ensured. This is done by elapsing the remaining delay time of the transition given in the queue head causing its execution.

Example 3.7 (Time Queue Update)
After the timing group has been entered by the transition with label *?req* in the path $\pi_{\mathcal{D}}$ of the Delivery Component, the timing queue has one entry $((t_7, 7))$, where t_7 is the transition with label *7* and source state *D2*.

The execution of $?cancel$ changes the configuration of the Delivery Component and correspondingly its state from $D2$ to $D3$, but does not leave the timing group. However, the delay transition to be executed has changed from the transition with source state *D2* to the one with source state *D3*. For this reason the entry $((t_7, 7))$ in the timing queue has to be updated, which means that the transition in the tuple is changed correspondingly.

Since the execution of the transition with label *!cancel* takes 1 time unit, this elapse of time has also to be reflected in the timing group. For this reason the remaining time of all time queue entries are decreased by the transition execution time of the last executed transition. In this scenario this means that the remaining delay of 7 is reduced by one leading to the time queue $((t_7, 6))$. □

Since it is not known a priori if the execution of a delay transition is required or has to be prevented to fulfill the test goal, the following three resolution strategies are used to compensate the lack of information at this time. Each strategy leads to a set of different traces, where the delay transition is always treated differently.

1. *Execute*: This strategy assumes that the delay transition is executed. Therefore its execution is incorporated into the trace, for which the according traces performing the completion steps are built.
2. *Prevent*: The execution of the delay transition is prevented in this scenario, which is done by adding constraints to the trace, so that no valuation exists satisfying the transition guard. For this reason the additional constraint is the inverse of transition guard and the transition does not occur on the final trace.
3. *Stable*: The strategy used in this case is again the prevention of the delay transition execution, but it is achieved by changing the state before enough time elapses. For this reason additional traces are created leading to *stable states*, which do not belong to the same timing group as the currently overdue delay transition. We use the name *stable states* for states of an ESTS, which are not changed by a delay transition for a specific amount of time.

The union of the resulting traces created by these three strategies is the result of a single delay transition handling during the test case generation.

3.4. Variable Valuation

The result of a structural search and the communication resolution, respectively is a set of symbolic paths, describing potential executions or behaviors of the system. However, the symbolic paths incorporate only the structure of each ESTS and do not involve the variable valuations, which are required in order to ensure that the system is executed in the intended way.

For this reason a feasible variable valuation for each symbolic path has to be calculated in order to build paths as defined in Section 2.1.4, which can be executed and form the resulting test case. The variables of a symbolic path cannot be freely chosen, because they are restricted by the transition guards φ and can have dependencies between them caused by attribute update functions ρ.

In order to calculate feasible signal parameters for a given attribute valuation, which is given by the initial configuration of the ESTS, the guards and attribute update functions of a symbolic path ψ have to be consolidated. The consolidation is done by the building of a path constraint Φ, which is a conjunction of the guards combined with the attribute update functions. This means that a path constraint is a predicate with unknown variable valuations. In order to generate a feasible path a variable valuation ensuring the path constraint evaluates to true has to be found. The calculation of values fulfilling given conditions is done with constraint solvers, which are especially developed for such purposes.

A path constraint is built bottom up, meaning that its corresponding symbolic path is iterated from the last to the first transition. During the iteration the initially empty path constraint is extended by adding the transition guards. At this stage it is possible to incorporate additional constraints, which could be a condition imposed by a test engineer to force a certain signal parameter valuation.

The creation of the path constraint is performed stepwise, where for each transition a variable replacement, which is denoted by $\triangleleft$, and the constraint extension is performed. These two actions are shown in Equation (3.4), where Φ_i is the updated path constraint, i is the index of the transition with guard φ_i and attribute update function ρ_i which are incorporated and Φ_{i+1} is the path constraint built up to the transition at position $i + 1$. Since the path constraint is built backwards Φ_i has a smaller index than its predecessor Φ_{i+1} and the values of i depend on the system path meaning that $1 \leq i \leq k$, where k is the number of its contained transitions. In the case $i = k$ where no predecessor exists, only the guard of the transition is considered and $\Phi_k = \varphi_k$.

$$\Phi_i = (\Phi_{i+1} \triangleleft \rho_i) \wedge \varphi_i \tag{3.4}$$

During variable replacement $\Phi_{i+1} \triangleleft \rho_i$, the variables in Φ_{i+1} are replaced by terms of the attribute update function. This means that the terms, defining the calculation of the variable update value, are used instead of those variables, which values are changed in the update function.

In addition it is required that the signal parameter names are renamed in the path constraint to ensure their uniqueness. Otherwise the constraints on input signal parameters could interfere with each other if the same signal occurs multiple times on the symbolic path and therefore could prevent the calculation of feasible parameter values.

The last step of the path constraint creation is the replacement of the attributes $\Phi_i \triangleleft \iota_0$, where ι_0 is the initial attribute valuation. After this step only signal parameters remain in the path constraint, for which values have to be calculated.

Given a path constraint consisting only of constraints on signal parameters, a constraint solver is used to calculate feasible parameter values. Feasible means in this case that all conditions of the path constraint are satisfied. Since the conditions in the path constraints can be mutual exclusive no valuation can be computed and makes the path infeasible and cannot be used as a test case. For this reason the infeasible paths are dropped and will not be considered further on. The reason for the occurrence of mutual exclusive conditions on a symbolic path is that during the structural search the variable values are not considered. This means that also transitions are considered which executions are possible, but prevented due to the used guards.

In this approach the constraint solver shipped with GNU PROLOG is used, which could simply replaced by another one. The selection of a constraint solver imposes some restrictions on the test generation process, because for example the GNU PROLOG constraint solver is restricted to positive integers and also limits the length of the constraint to be solved.

The result of the constraint solver contains the values of each parameter contained in the passed constraint. This means in this context all signal parameters of every transition on the symbolic path, which allow the execution of the symbolic path on the ESTSs. Due to this execution the attribute values are also calculated and stored in configurations. The symbolic path in combination with the calculated signal parameter and attribute valuations represents a path of an ESTS, consisting of parameterized labels and configurations.

$$\begin{aligned}
!choc[num = dq] &\mapsto num = dq \\
7 &\mapsto num = dq \\
?cancel/dq = 0; &\mapsto num = 0 \\
?req/dq = qnt; &\mapsto num = 0 \\
!req[qnt = cnt]/\rho^* &\mapsto num = 0 \wedge qnt = cnt \\
?ok &\mapsto num = 0 \wedge qnt = cnt \\
\gamma[cnt \geq stock] &\mapsto num = 0 \wedge qnt = cnt \wedge cnt \geq stock \\
?btn/cnt = cnt + 1; &\mapsto num = 0 \wedge qnt = cnt + 1 \wedge cnt + 1 \geq stock \\
?btn/cnt = cnt + 1; &\mapsto num = 0 \wedge qnt = cnt + 1 + 1 \wedge cnt + 1 + 1 \geq stock
\end{aligned} \tag{3.5}$$

Example 3.8 (Path Constraint)
Formula (3.5) shows the backward construction of the path constraint for the system path $\eta_{\mathcal{OD}} = \psi_{\mathcal{D}} \prec_1 \psi_{\mathcal{O}}$ depicted in Figure 3.4, where the transitions including their guards and actions are shown to the left of $\mapsto$ and ρ^* is $stock = stock - cnt; cnt = 0;$.

In this example we assume that $stock = 2$ and $cnt = 0$, which leads to the path constraint $\Phi = (num = 0) \wedge (qnt = 0 + 1 + 1) \wedge (0 + 1 + 1 \geq 2)$. This path constraint Φ is passed to a constraint solver which calculates the values $num = 0$ and $qnt = 2$, which are the parameter values of the signals $!choc$ and $!req$, respectively. □

3.5. Additional Generation Constraints

Although the number of possible generated test cases is already limited by the restrictions imposed during the structural search, additional refinement methods are presented in the remainder allowing to control the generation process itself. In general these methods are used to limit the state space taken into account and are limiting the generation possibilities. As for the structural search they allow the test engineer a fine grained approach to enhance the scalability and tailor the generation process to the specific needs of the application.

Test Case Number: A simple method to reduce the number of found test cases is to limit their number directly by aborting the generation process if a given level has been reached. This limit can be applied to each test purpose step, allowing to search the state space in a band, whose bounds depend on the allowed test case number.

Feasible Path Selection: This approach is applied on the feasible paths found for a test purpose step and selects one or a number of paths with respect to some given criteria. The criteria used in this approach involve metrics like the path length and its coverage. Multiple selectors are available for each metric:

- **Length** selects the paths according to the number of contained transitions, like the shortest, longest, median, random or all.
- **Coverage** calculates and sorts the paths according to their additional state or transition coverage and selects a given number of paths, whereas the paths with the most additional contribution to the metric are used first.

Timer Resets: Due to loops in the ESTSs the number of the maximum allowed resets of timers has to be defined in order to avoid an infinite number of paths. A timer reset in this context means to create a sender path, which causes the execution of a loop in an ESTS and ends in the same state as before. If this state has also an outgoing delay transition, whose execution is prevented by this sender path, then a timer reset occurred. Since this behavior can be repeated infinitely often due to the loops, a restriction has to be defined for the test case generation, where the variable valuation cannot be taken into account at this time.

Value Restriction: To allow the test engineer the selection of a specific value or value range for an attribute or signal parameter, an additional condition can be provided. This condition is added to the path constraint and ensures in this way that the restricted values in the resulting test case are within the given bounds.

Non-Target Exclusion: Another restriction called Non-Target Exclusion (NTE) in the remainder is introduced in this work, neglecting other outgoing transitions from the same source state as a target transition.

3.6. Speedup Techniques

Due to the possible exponential path growth and the resulting high computational effort, various techniques described below were introduced to reduce the computation time. They are based on well known strategies like divide-and-conquer, caching and the efficient use of state-of-the-art hardware to reduce generation time.

Feasible Input Data: The goal of this technique is to identify infeasible paths as soon as possible, which prevents further communication resolutions and therefore the number of paths. Although, this strategy reduces the number of paths considered, it does not limit the state space and keeps the full path finding capabilities. This is achieved by using the constraint solver not only at the end of the generation process, but also during the communication resolution phase. This allows to filter infeasible state update- or input generation-paths directly after their creation. The algorithm used to create the path constraint for these paths is the same as the one used at the end of the communication resolution phase.

Variable Caching: This method extends the *feasible input data* technique by storing the values found during the constraint solving in a temporary storage. These values are used in further feasibility checks involving the solving of a path constraint. Since the usage of concrete values instead of their value ranges could lead to an unsatisfiable constraint, this possibility has to be taken into account during the constraint solving process. In this approach we try to solve the constraint again, but without any cached variables. If a solution can be found then the cached variables are updated with the new values or the path is refused due to its infeasibility.

Constraint Minimization: The replacement of variables by assignments and the final replacement of attributes by their initial valuation can lead to a lot of terms in the path constraint containing static calculations. For example the terms $y = 3 + x$ and $x = 2 + p$ are part of a transition action. Due to the replacement of variables during the building of the path constraint the term $y = 3 + 2 + p$ has to be solved by the constraint solver. The minimization step performs those calculations on the path constraint, which do not depend on an unknown variable value. This means that in the example term the value for $3 + 2$ is calculated leading to $y = 5 + p$. The approach has more impact on the overall generation speed if variable caching is used, since due to the additional replacements even more terms can be minimized.

Path Constraint Splitting: This method splits the path constraints corresponding to the test purpose steps to reduce its size and therefore the needed calculation time. On the one hand the constraint splitting can lead to infeasible variable valuations for the remaining test cases, because only a subset of the available variables are considered at the same time. On the other hand the path constraint splitting can reduce the effort needed during test case generation substantially, if it is applied during the sender- and output-path generation. The reason is that infeasible paths can be excluded as soon as possible and are therefore not considered in the following resolution steps.

Multi-Threading Support: The independence of the paths between each other suggests the usage of multiple threads, allowing parallel processing. Although the performance increase is linear at best, given the complexity of test case generation the available processing capacity should be used as good as possible.

Chapter 4

System Modeling

Parts of this chapter are taken from ´Test Sequence Generation from Communicating UML State Charts: An Industrial Application of Symbolic Transition Systems´ [20] which is joint work with Bernhard Peischl, ´Test case generation in practice for communicating embedded systems´ [23] which is joint work with Bernhard Peischl and ´Test case generation in practice for communicating embedded systems´ [22] which is joint work with Franz Wotawa.

4.1. Unified Modeling Language

Although the semantics of an ESTS is defined and therefore is a sound basis for test case generation, they are not convenient for system modeling. For this reason we use UML State Machines in this work to remedy this limitation and show a straight forward model transformation of a State Machine into an ESTS. The usage of UML was encouraged due its strong tool support and was also used in the industrial tool chain in which this work took place.

UML State Machines are a variant of Harel state charts [29] and support a wide range of syntactical modeling elements to foster an efficient model development. From a semantic point of view an UML SM is similar to an EFSM, where a transition consists of inputs and outputs. In contrast the inputs and outputs are modeled using distinct transitions in an ESTS and therefore an ESTS has a higher expressiveness than an EFSM and an UML State Machine, respectively. However, this higher expressiveness is not considered in this work, because deterministic ESTSs are required for test case generation.

4.1.1. State Machines

Since the ESTS was defined similarly to a State Machine, we use the same names as used for the ESTS also in this section in order to define the State Machine.

Definition 4.1
An UML State Machine *is a tuple $\langle W, L, T, V, \iota_0 \rangle$, where W is set of vertices, L is a set of labels, T is the transition relation, V is a set of variables and ι_0 is the initial attribute valuation.* $\square$

The set of vertices $W = S \cup R$ is the union of the set of *states* S and a set of *pseudo states* R, where pseudo states are syntactical modeling elements and $S \cap R = \emptyset$. The label set $L = L_S \cup \mathbb{N}_1 \cup \{\epsilon\}$ is the union of a set of signals L_S, time delays $\mathbb{N}_1$ and the empty label ϵ. The set of variables $V = A \cup P$ consists of the set of attributes A and the set of signal parameters P, where $A \cap P = \emptyset$, the attributes are properties of the SM and the signal parameters are assigned to a signal and used to pass data between the SMs.

A variable valuation of a SM is the same as described in Section 2.1.1 for an ESTS and we use correspondingly ι to denote an attribute- and ς to denote a signal parameter valuation.

$$T \subseteq W \times L \times \mathfrak{F}(V) \times \mathfrak{T}(V)^A \times W \tag{4.1}$$

The transition relation is given in Equation (4.1), where W is a set of vertices, L a set of labels, $\mathfrak{F}(V)$ is a first order logic predicate over the variables V and $\mathfrak{T}(V)^A$ is a set of actions. An action $\alpha \in \mathfrak{T}(V)^A$ is a sequence of attribute update functions ρ and send actions χ. An attribute update function ρ changes the attribute valuation ι using the signal parameters ς to ι' and is denoted by $\iota' = \rho(\iota \cup \varsigma)$. A send action χ produces an output sent to a specified SM, where the output consists of a signal label and a parameter valuation. We denote a send action χ as $send(\mathcal{A}, l(\varsigma))$, where $\mathcal{A}$ is the target SM, l the label of the produced output and ς the signal parameter valuation.

Definition 4.2 (Action Concatenation)
The concatenation of two actions $\alpha_a = \rho_1^a, \ldots, \rho_n^a$ and $\alpha_b = \rho_1^b, \ldots, \rho_m^b$ is given by $\alpha_a \circ \alpha_b = \rho_1^a, \ldots, \rho_n^a, \rho_1^b, \ldots, \rho_m^b$, where each of the attribute update functions in both actions could also be send actions. $\square$

A transition $t \in T$ is a tuple $(w, l, \varphi, \alpha, w')$, where $w \in W$ is its source vertex, $l \in L$ is its label, $\varphi \in \mathfrak{F}(V)$ is its guard, $\alpha \in \mathfrak{T}(V)^A$ is its action and $w' \in W$ is its target vertex.

A transition represents an allowed state change from the source vertex to the target vertex if its guard evaluates to true. The action of a transition defines the outputs to be produced and the update of the attribute valuation. The transitions of an UML SM are similar to those defined for an ESTS and also serve the same purpose. A transition is said to be *outgoing* of a vertex if a given vertex is the transition source vertex and *incoming* if it is its target vertex.

A *completion transition* in UML is a transition connecting two states in S and is labeled with the empty label ϵ. A *timed transition* specifies the passage of time or a specific point in time, which are denoted with *after* and *at*, respectively in a State Machine. This means for example that a transition can be labeled with *after(100)* or *at(Jan. 1st 2035)*, where 100 defines the relative time amount which has to elapse and could be provided for example in milliseconds.

A *default transition* is a special transition, which source vertex is an *initial pseudo state*, has a guard always evaluating to true and is labeled with the empty label[1]. The initial pseudo state does not have a corresponding element in an ESTS, because a default transition is used to define the initial state of a State Machine, which is defined by the initial configuration in an ESTS. However,

[1]UML does not define the empty label, but it is used in this work to make the absence of another label explicit.

a default transition also has an action allowing the production of outputs and the change of the attribute valuation, which has to be considered during the model transformation as shown in Section 4.2.

States and Hierarchy

A state $s \in S$ has one entry- α_n and one exit- α_x action, where $\alpha_n, \alpha_x \in \mathfrak{T}(V)^A$ and both of them might be empty. In UML the states in S are distinct by *submachine states*, *composite states*, *simple states* and *orthogonal states*, where the orthogonal states allow the representation of parallelism within a State Machine and are not considered in this work.

A submachine state is a state in which another state machine is inserted. An inserted state machine is also called a *Nested State Machine (NSM)* and the State Machine holding the submachine state is referred to as the *containing* State Machine. Since nested State Machines are not fully drawn in the graphical representation of a State Machine, they can be used for example to describe a behavior on multiple abstraction layers, where each behavior refinement could be defined in a NSM.

A composite state can contain other composite states, pseudo states or simple states, where a simple state can only contain pseudo states but must not contain other states or a NSM. Basically this means that a state containing other states is named a composite state, while a state having no contained states is called a simple state.

The recursive definition of the states allows the representation of a *state hierarchy* by a tree data structure, where every vertex is represented by a node and its containing state is defined by an edge. The root node of a state hierarchy is the State Machine itself and each of the leaves is a simple- or pseudo-state. In this work we refer with *level* of a vertex to the number of nodes on the path in the hierarchy tree from the node representing the vertex to the root node.

A vertex w containing another vertex w' is said to be the *parent* of w', while w' is said to be the *child* of w. A pseudo state cannot contain any other vertices and therefore can never be a parent, whereas simple states can only be parents of pseudo states.

Example 4.1 (State Hierarchy)
In Figure 4.1 the state hierarchy of the Keyless Access Controller (KAC) State Machine depicted in Figure 4.3 is shown, where the root state *StateMachine* indicates the State Machine itself and has two child states namely *CarStopped* and *CarMoving*. The levels of the vertices in the state hierarchy are indicated by *L0* to *L3*, where *L0* is an imaginary level referring to the State Machine. This means that the states *CarStopped* and *CarMoving* are on level 1, *CarLocked*, *CarUnlocked*, *Normal* and *Warning* on level 2 and the remaining states are on level 3. □

In the remainder we refer with $\mathbf{children}(w)$ of a vertex $w \in W$ to the set of all childless vertices $W' \subseteq W$ contained in w if the vertex w has any children. In the case the vertex w does not have any children we refer to a set containing only the vertex w itself. All childless vertices means that a vertex is in the result set regardless if it is a child of other vertices contained in W.

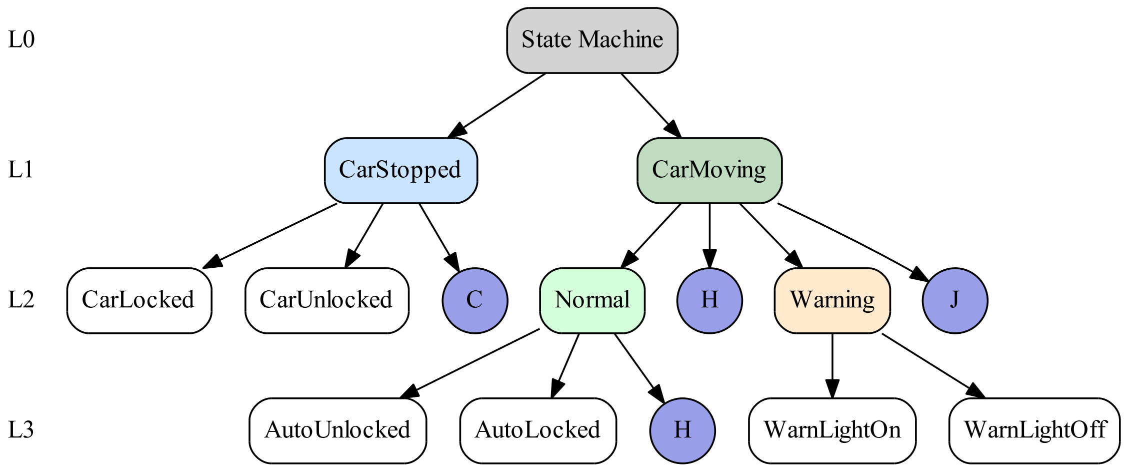


Figure 4.1.: State hierarchy of the State Machine shown in Figure 4.3.

Similarly we use $\mathbf{simple}\,(s)$ to denote the set of all contained simple states $S' \subseteq S$ of a state $s \in S$ if s is a composite state and otherwise to denote the set only containing the state s.

In addition we refer with $\mathbf{ancestors}\,(w)$ of a vertex $w \in W \cup \{s_\epsilon\}$ to the set of states $S' \subseteq S$ such that $\mathbf{children}\,(w) \subseteq \mathbf{children}\,(s)$ and $s \in S'$, where s_ϵ explicitly denotes a non existing state. In principle this means that the set $\mathbf{ancestors}\,(w)$ contains all parent states of w, where a parent state has to contain the vertex w and all its possible children. The non existing state s_ϵ is used to explicitly express the absence of a state later in this work and does not have any parent states. For this reason $\mathbf{ancestors}\,(s_\epsilon) = \emptyset$.

We use $\mathbf{default}\,(s)$ to refer to the default simple state $s' \in \mathbf{simple}\,(s)$ if s is a composite state and to the state s itself if it is a simple state. A default simple state is a simple state if it also is the target vertex of a default transition.

Example 4.2 (Ancestors, Children and Simple States)
The State Machine in Figure 4.3 contains many examples of e.g. parents and ancestor sets, whereof a few will be discussed here in detail.

Given the state hierarchy depicted as tree in Figure 4.1 it can be seen that the parent of *Normal* and *Warning* is *CarMoving*, which in return means that the *Normal* and *Warning* are children of *CarMoving*. Since *CarMoving* does not have a parent state the union of the ancestor sets $\mathbf{ancestors}\,(\mathit{Normal})$ and $\mathbf{ancestors}\,(\mathit{Warning})$ therefore contains only their common parent $\{\mathit{CarMoving}\}$. The reason is that the state *State Machine* shown in Figure 4.1 is only imaginary and does not exist in the corresponding State Machine in Figure 4.3, where no such state is depicted. Moreover the states *CarMoving* and *CarStopped* neither have any parent, which means that their corresponding set of ancestors is empty.

The state *AutoLocked* is a child of *Normal*, which is a child of *CarMoving* itself. This means in this case that both states are ancestors of *AutoLocked*, which can be expressed by the set $\mathbf{ancestors}\,(\mathit{AutoLocked}) = \{\mathit{Normal}, \mathit{CarMoving}\}$. In contrast to *AutoLocked* the parent of *CarLocked* does not have a parent and therefore the ancestor set $\mathbf{ancestors}\,(\mathit{CarLocked})$ contains only its parent $\{$*CarStopped*$\}$. In general this means that the ancestor set $\mathbf{ancestors}\,(s)$ contains all states starting from the parent of the given state s up to *State Machine*.

In comparison to $\mathbf{ancestors}\,(s)$ the set $\mathbf{children}\,(s)$ is built the opposite way, which means that it takes the states lying on a higher level than the given state s into account. For example the set $\mathbf{children}\,($*CarMoving*$)$ is the set of vertices $\{$*AutoUnlocked*, *AutoLocked*, *WarnLightOn*, *WarnLightOff*, H_1, H_2, $J\}$ where H_1 is the history state and J the junction state contained in *CarMoving* and H_2 the history state contained in state *Normal*. The set $\mathbf{children}\,($*CarStopped*$)$ contains the vertices *CarLocked*, *CarUnlocked* and C, where C is the condition state in the composite state *CarStopped*.

The set $\mathbf{children}\,(s)$ is more general in comparison to $\mathbf{ancestors}\,(s)$, because it contains vertices instead of states, where states are a subset of vertices. The reason for this difference is that states can contain pseudo states but not vice versa, meaning that pseudo states cannot have any children. For this reason we use the set $\mathbf{simple}\,(s)$ to refer to the states out of the vertices set $\mathbf{children}\,(s)$ for a given state s, which means that $\mathbf{simple}\,(s) \subseteq \mathbf{children}\,(s)$.

For this reason the set $\mathbf{simple}\,($*CarMoving*$)$ contains in contrast to $\mathbf{children}\,($*CarMoving*$)$ only the states $\{$*AutoUnlocked*, *AutoLocked*, *WarnLightOn*, *WarnLightOff*$\}$, where the pseudo states are missing. Similarly $\mathbf{simple}\,($*CarStopped*$)$ is the reduced set $\{$*CarLocked*, *CarUnlocked*$\}$ without the condition state. □

Example 4.3 (Default Transitions)
In order to define the entered child state of a parent state having an incoming transition, default transitions are used as shown in Figure 4.3, where the default transition source vertex is the initial pseudo states depicted as red dot. Therefore for every parent state at each level having multiple child states a default transition is required. In this State Machine the default state $\mathbf{default}\,($*CarMoving*$)$ is *Normal*, which has itself a default state $\mathbf{default}\,($*Normal*$) =$ *AutoLocked*. The initial state of the State Machine is also defined via a default transition, which is the state *CarStopped* in this case. □

Due to the state hierarchy a transition can either be *local* or *external*, which are both additional properties of a transition having a composite state as source- and one of its contained states as target-vertex. A transition is said to be external if it leaves and reenters the composite state before the target state is reached. In contrast a transition is said to be local if it does not leave the composite state. From a graphical point of view this means that an external transition crosses the border of the composite state while the local transition does not. The difference in the semantics of these two transition types is, that the entry- and exit-action of the composite state are taken into account for external but not for local transitions.

Pseudo-states

Pseudo states are a special kind of vertices to allow a more efficient graphical modeling. They are syntactic elements and do not provide any semantics. However, they can be transition source- and target-vertices and therefore be used to connect transitions to each other. The connected transitions in combination with the used pseudo states describe their semantics, where the pseudo states considered in this work are described below. The listed pseudo states are those provided by RHAPSODY, which was used in this work, and do not fully comply to the UML specification [9].

- **Initial Pseudo State:** The initial pseudo state is used as source vertex for default transitions, where a default transition uniquely defines the initial state, which is contained in a State Machine or composite state. Every composite state or the SM itself has to contain exactly one initial pseudo state. A default transition is only needed if a state or a SM contains more than one states. In the case that they contain only one state, the initial state is already uniquely defined and the default transition can be omitted.
- **Deep History State:** An incoming transition of a history state represents a virtual transition to the last visited simple state contained in the composite state or SM, which is the parent of the history state. Virtual in this context means that the transition does not exist explicitly, but the State Machine behaves as if a transition between its source vertex and the last visited simple state would exist. This requires that the source vertex of an incoming transition of a history state must not be the same or another history state. In fact a history state can only have exactly one outgoing transition, which defines the default state if the composite state containing the history state was never visited before and therefore no information about the last visited simple state is available.
- **Condition State:** A condition state has one incoming and multiple outgoing transitions, where the guard of the incoming transition is extended by the guards defined on the outgoing transitions. For this reason a condition state can avoid the creation of multiple outgoing transitions with the same trigger from the same source vertex and therefore simplifies the graphical model representation and enhances its readability.
- **Junction State:** Junction states are used to simplify the graphical representation by merging its incoming transitions into one outgoing transition. The outgoing transition of a junction state is used to define the target vertex, which is common to all incoming transitions. The outgoing transition must have an empty label and neither a guard nor an action.
- **Entry- and Exit Points:** Entry- and exit points allow the connection of two vertices, where one of the vertices lies inside and the other one outside of a NSM. Since a NSM is graphically represented as another independent SM such connection is not possible due to the graphical restrictions. Entry- and exit points always have to be used in pairs, where one point – similar to the vertices which shall be connected with a transition – is inside and one point is outside of the NSM. This pair represents a connection in the graphical representation, to which transitions can be connected. An entry point can have multiple incoming but no outgoing transitions, whereas for an exit point only one outgoing and no incoming transitions are allowed.
- **Diagram Connector:** Diagram connectors can be used within a state machine to connect two vertices beyond the graphical bounds of multiple NSMs. Such a connection could also be realized with several entry- and exit-points, but diagram connectors foster an easier modeling in such scenarios.

UML specifies additional pseudo states like join and fork, which are used in combination with parallel regions to describe concurrent behavior. Since we only consider deterministic UML SMs

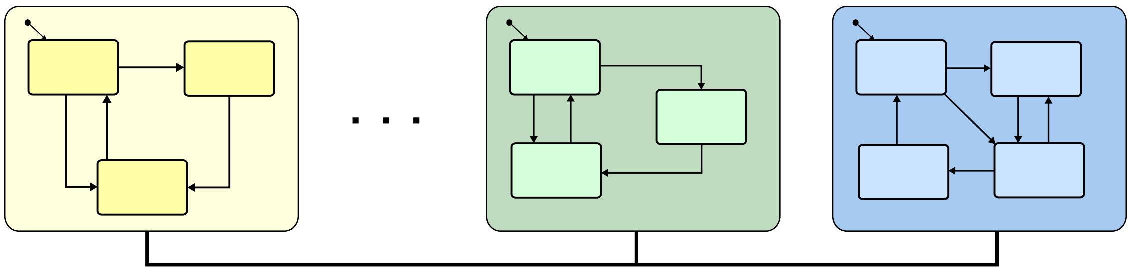


Figure 4.2.: Keyless Access System Architecture

and the pseudo states explained above we omit their description in this work.

Example 4.4 (Keyless Access)
An illustrative example is shown in Figure 4.2, which consists of three parallel running components namely the Keyless Access Controller (KAC), Key Location Detector (KLD) and the Power Controller (PC). The depicted system implements a simple keyless access controller, which allows to lock, unlock, enter and to start a vehicle without using the key or its integrated remote control. This means that in this example it is sufficient that the key is inside the car in order to start its engine, rather than using the key to operate the ignition lock.

In order to achieve the intended functionality these three components have to interact with each other by sending messages over the shared network. The behavior of each of these components is defined in an UML State Machine, where the corresponding State Machines are shown in Figure 4.3 (KAC), Figure 4.4 (KLD) and Figure 4.5 (PC), respectively. Throughout this example the name of signals, which are sent from other State Machines, start with *ev* while externally sent signals start with *ext*. We also use in all State Machines the variable *status* to refer to a signal parameter, whereas all other occurring variables are attributes.

The Keyless Access Controller is the main component in this system and defines under which circumstances the vehicle will be locked and unlocked, respectively. It also provides some basic functionality for handling exceptional system usage, where in this case a warning light is switched on instead of following the usual behavior. An exceptional system usage is for example that the key is thrown out of the door window while the car is still moving. Under normal circumstances, this means in this case the car is stopped, the vehicle will simply be locked and the power for the on-board network will be turned off. However, such a behavior is not desired in case the vehicle is still moving, which could cause life threatening injuries and is therefore a very important aspect of the overall safety of the car.

The KAC distinguishes mainly between two states of the car, namely if it is in motion or not. The difference of this property is described by the two states *CarMoving* and *CarStopped*, which define a different behavior for basically the same inputs. In case the car is stopped the vehicle will be locked or unlocked with respect to the location of the key, which is determined by the KLD. This means that the car will be locked if the key gets out of range of the Key Location Detector and unlocked if it gets in range.

In case the car is in motion, the car becomes locked automatically if its speed exceeds the

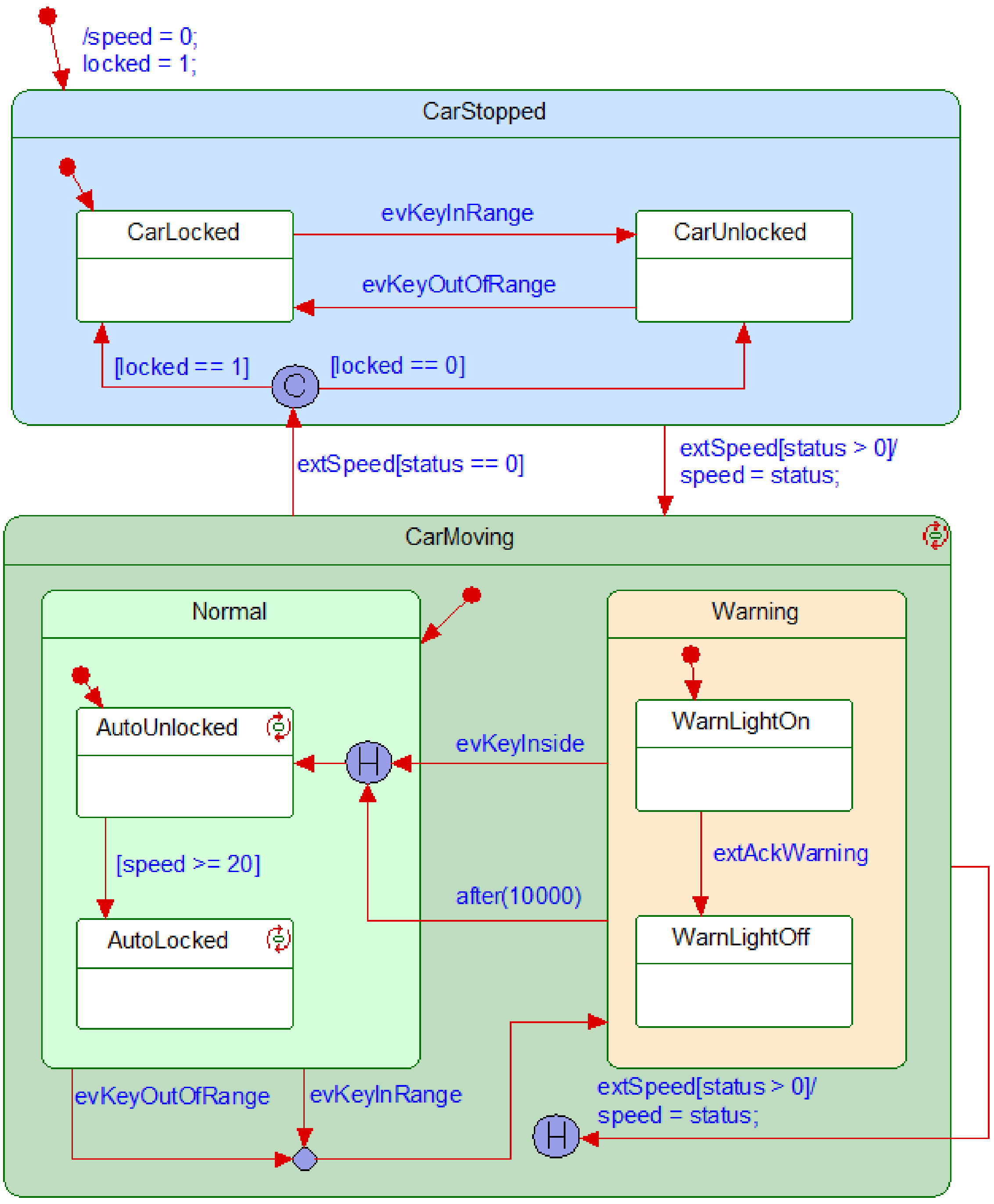


Figure 4.3.: Keyless Access Controller (KAC)

limit of 20, which does not define its unit and refers to its internal representation. The handling of exceptional system usage is described by the state *Warning* and its sub-states, where a warning light is turned on. This warning can be acknowledged by the driver and will be turned off automatically after 10000 time units.

The Keyless Access Controller contains several pseudo states, completion- and timed transitions in order to define its behavior. For example it contains two history states H_1 and H_2 contained in the states *CarMoving* and *Normal*, respectively, which are drawn as blue filled circles with an H in it.

This State Machine also shows one condition state contained in *CarStopped*, which is depicted similarly to the history state but with a C in the circle instead of a H. Finally also a junction state is used which is again contained in state *CarMoving* and is simply drawn as blue filled circle.

In the KAC State Machine also various transition types are used, like the completion transition between the states *AutoUnlocked* and *AutoLocked* and the timed transition with source vertex *Warning* and target vertex H_2. In addition also exit- and entry-actions are used, which are indicated by the green ellipse surrounded by circular arranged arrows in the name bar of a state. In this example the states *CarMoving*, *AutoLocked* and *AutoUnlocked* have at least an exit- or an entry action.

In this example we assume that the car is initially locked and the key is out of range of the KLD, which is specified by the actions and the target vertices of the respective default transitions of these State Machines.

The job of the Key Location Detector is to determine the position of the key, which is separated into three zones in this example namely *out of range*, *in range* and *inside*. Out of range means that the key is in such a far distance to the car that it cannot be detected at all. The key is in range if it can be detected by the KLD but is still outside the car and it is inside the car otherwise.

These zones are reflected in the KLD State Machine shown in Figure 4.4, which contains states with the corresponding names. This State Machine tracks the location of the key and notifies the other components, namely the KAC and PC in the case of a location change. The notification is realized by send actions, which are indicated by the keyword *send* in the transition actions.

The behavior of the Power Controller is defined by an even simpler State Machine. The PC basically switches between a high- and low power mode of the on-board network, where the system is in high power mode if the key is in range or inside the car and in low power mode otherwise. After the Power Controller is in the low power mode for a certain amount of time, which is 1000 time units in this example, the power supply will be turned off. □

Model Limitations

Since this work is focused on the automatic generation of test cases using symbolic execution, some restrictions to the system model are applied to reduce the size of the state space. First the allowed data types for attributes and signal parameters are limited to integer or boolean data types,

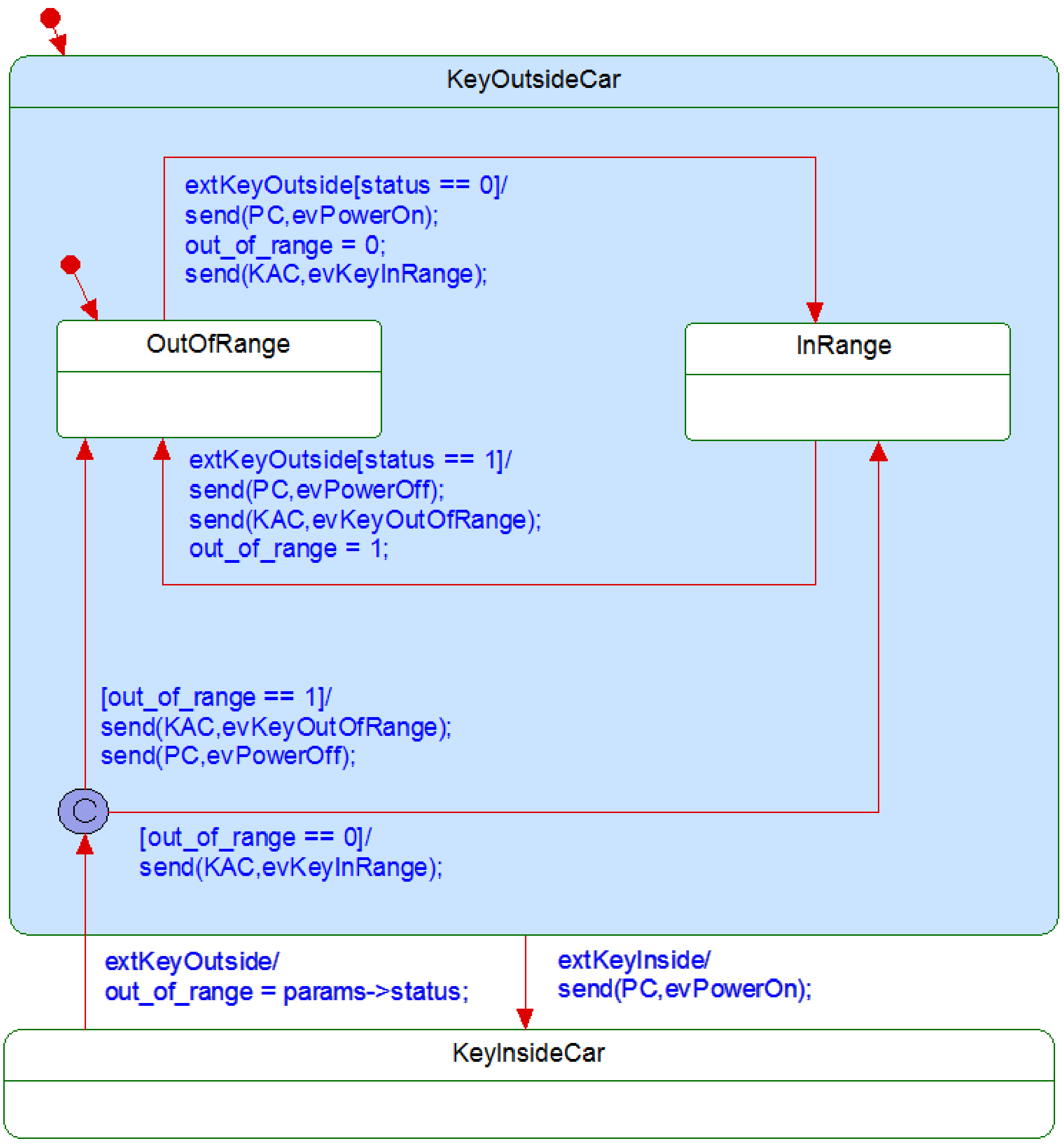


Figure 4.4.: Key Location Detector (KLD)

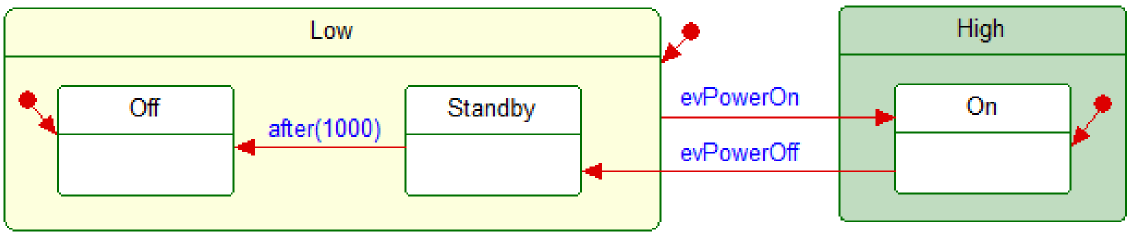


Figure 4.5.: Power Controller (PC)

because of the restrictions imposed by the constraint solver used during the test case generation process.

In addition we do not allow parallel regions, which semantics is defined ambiguously in UML and the accompanied growth of the state space. However, parallelism can be expressed by the usage of multiple ESTS described below, which provides a precise meaning of the behavior but uses a determined communication scheme. In this work we only consider timed transitions specifying a relative passage of time elapsed in their source vertices.

4.1.2. Annotations

Annotations contain arbitrary information and are linked to modeling elements like states and transitions. This means that for example State Machines can be enriched with information regarding but not influencing the expressed behavior. We refer with annotations to stereotypes and tags as defined in the UML specification.

A stereotype consists of a name and allows the categorization and grouping of modeling elements. The used names have to be unique within the UML model, where a stereotype is enclosed by $\ll$ and $\gg$ in its graphical representation.

Stereotypes can be used for example to logically group the states and transitions of a State Machine, which are part of the dedicated functionality like error handling. This allows an automated processing depending on their intended usage.

A tag is in contrast to stereotypes a key value pair, where the key is also the tag name and the value consists of a character sequence. A modeling element of an UML model can contain a list of tags, which keys have to be unique within this list.

4.1.3. Object Model Diagram

An Object Model Diagram (OMD) is used in this approach to select the State Machines used to describe the system behavior, where a system consists of a set of components and the behavior of these components is described by State Machines. Together they are a formal specification of the SUT which is used for test case generation.

Since RHAPSODY was part of the the industrial tool chain in which this work takes place, it was used as modeling tool which imposed certain modeling rules in order to allow code generation. RHAPSODY follows the same principle as in object oriented software development, where the code is structured in classes defining the behavior by its functions and objects are their instantiations. In the case of UML modeling the behavior of a class is described by State Machines, which are therefore embedded in a class instantiated and linked in an OMD. A link in an OMD is the graphical representation of a reference to another object, which in this case is needed in order to send signals from one State Machine to another.

Example 4.5 (Object Model Diagram)
Figure 4.6 shows the corresponding OMD of the keyless access system explained in Example 4.4,

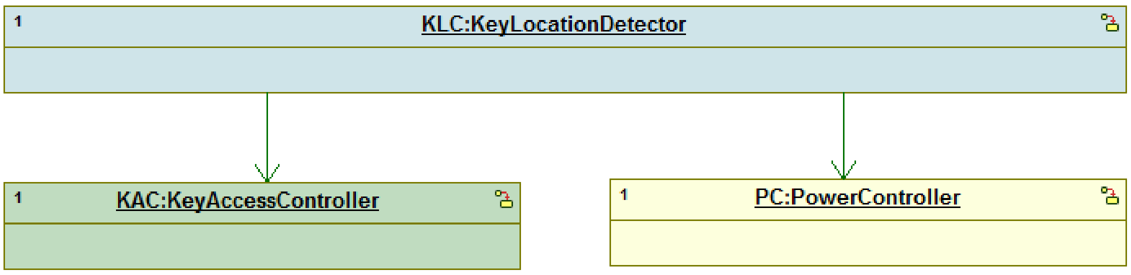


Figure 4.6.: Object Model Diagram of the Keyless Access Control System

where for each component an object was created. The object name is written before a colon followed by the corresponding class name as shown in this figure, where the name of every graphical representation of an object is presented in this way. For example *PC:PowerController* states that the object has the name *PC* and is an instantiation of the class *PowerController*.

The green arrows indicate the directions in which signals are sent meaning in this example that the KLD sends signals to the KAC and the PC, whereas the KLD and PC do not send a signal at all. □

4.2. Model Transformation

Since the system is specified based on UML State Machines, which lack the required formalism needed for test case generation, the state machines have to be transformed into ESTSs. Since an ESTS is an extension of a STS with elements originating from the UML specification like completion- and delay-transitions, the transformation is a straight forward task and can be done fully automatically. It is partially described in [20], where a formal definition is already given. However, the approach presented there does not include delay transitions.

The transformation of a SM into an ESTS is performed independently of other State Machines of the same system, meaning they are transformed one after another. The transformation is split into three tasks, namely pseudo state removal, model flatting and the mapping of the flat State Machine on an ESTS, which are described in the following sections. In these sections we use $w \xrightarrow{w,l,\varphi,\alpha} w'$ to indicate an UML transition with source vertex w, target vertex w', label l, guard φ and action α.

4.2.1. Pseudo State Removal

Before an UML State Machine can be transformed into an ESTS, all pseudo states have to be removed. Since they do not provide additional semantics they can be represented by a set of newly created transitions. We omit the definition of the transformation of diagram connectors, entry- and exit points and nested State Machines, because they do not add any semantics and only influence the graphical representation. Therefore all incoming transitions into a pseudo-state can be simply relinked to the target state of the outgoing transition, which can be removed together with the pseudo state.

Condition State

A condition state $r_c \in W$, as it is defined in RHAPSODY, consists of one incoming and several outgoing transitions, where the incoming transition contains the trigger and the outgoing transitions have different guards. A condition state can be removed after the creation of a new transition for each of its outgoing transitions. The created transitions have the source vertex of the incoming and the target vertex of the outgoing transition as shown in Equation (4.2).

$$\frac{w \xrightarrow{l_1,\varphi_1,\alpha_1} r_c \quad r_c \xrightarrow{\epsilon,\varphi_2,\alpha_2} w'}{w \xrightarrow{l_1,\varphi 1 \wedge \varphi_2,\alpha_1 \circ \alpha_2} w'} \tag{4.2}$$

During the removal of the condition state also all its incoming and outgoing transitions have to be removed, which became obsolete after the creation of the new transitions.

Junction State

The removal of a junction state r_j used in RHAPSODY is similar to the removal of a condition state, where in contrast the junction state has several incoming transitions but only one outgoing transition. This outgoing transition defines the target vertex and has an empty label ϵ, a guard always evaluating to true and a possibly non empty action α. The transformation is performed as described in Equation (4.3), where we use $\top$ to denote a guard always evaluating to true.

$$\frac{w \xrightarrow{l_1,\varphi_1,\alpha_1} r_j \quad r_j \xrightarrow{\epsilon,\top,\alpha_2} w'}{w \xrightarrow{l_1,\varphi_1,\alpha_1 \circ \alpha_2} w'} \tag{4.3}$$

History State

In general a history state r_h can be transformed by creating a new transition from the source vertex of the incoming transition of this history state to every simple state contained in the composite state s_c, which is the parent of the history state.

However, since the information of the last visited state – to which the history state shall return per definition – is not explicitly stored in the State Machine, it has to be added during removal of the history state. This is achieved by adding an attribute a' in the State Machine for each of its contained history state. Each attribute a' stores an unique identifier of all simple states reachable via the history state, which are given by $\mathbf{simple}\,(s_c)$.

The value of the newly created attribute a' is initialized with a value, which does not represent a simple state and therefore has to be updated in the case of a state change. The update is achieved by the attribute update function ρ', which assigns an unique identifier representing one of the reachable simple states. Since this assignment has to be done every time one of these states is reached, the update function ρ' is concatenated to the entry action of the corresponding simple state.

In order to keep the intended semantics of the history state, the newly created attribute has to be considered in the guard of each transition, which was created for the history state. For this reason a condition φ' is conjunct with the guard of every created transition leading to one of the states in $\mathbf{simple}\,(s_c)$, where φ' requires that the attribute a' equals the unique identifier representing the state. The creation of the set of transition for a history state r_h is shown in Equation (4.4), where s_c is its parent composite state.

$$\frac{w \xrightarrow{l,\varphi,\alpha} r_h}{\left\{w \xrightarrow{l,\varphi\wedge\varphi',\alpha} s_s \mid s_s \in \mathbf{simple}\,(s_c)\right\}} \tag{4.4}$$

An additional transition, defining the default state if none of the reachable simple states has been visited before, has to be created too. This transition has also an extended guard, where the added term φ' requires that attribute a' has to have its initial value. This is shown in Equation (4.5), where `id` is the identity update function.

$$\frac{w \xrightarrow{l,\varphi,\alpha} r_h \quad r_h \xrightarrow{\epsilon,\top,\mathtt{id}} s}{w \xrightarrow{l,\varphi\wedge\varphi',\alpha} s} \tag{4.5}$$

The approach described above is correct for all modeling constitutions involving history states, but is inefficient in the case that the source vertex w of an incoming transition of a history state r_h is in $\mathbf{ancestors}\,(r_h)$, meaning that the source vertex w is an ancestor of r_h. The reason is that in this case the last visited state is always the same as the state, which has the outgoing transition to the history state. For this reason the semantics of the history state can be described by a set of loop transitions, where a loop transition has the same the source- and target vertex. Due to loop transitions the correct semantics can be preserved with a lesser number of transitions.

This situation is shown in Equation (4.6), where it is required that the source vertex of the incoming transition in the history state is a state $s \in S$, α_x are the concatenated exit- and α_n the concatenated entry actions.

$$\frac{s \xrightarrow{l,\varphi,\alpha} r_h \quad s \in \mathbf{ancestors}\,(r_h)}{\left\{s' \xrightarrow{l,\varphi,\alpha_x\circ\alpha\circ\alpha_n} s' \mid s' \in \mathbf{simple}\,(s)\right\}} \tag{4.6}$$

The concatenated exit- and entry actions α_x and α_n, respectively, are a concatenation of the exit- and entry actions of the composite states between the simple state and the history state parent. In this context between means the states in the state hierarchy, where the set of these composite states $S_{xn} \subseteq S$ is calculated as shown in Equation (4.7).

$$S_{xn} = \mathbf{ancestors}\,(s') \setminus \mathbf{ancestors}\,(r_h) \tag{4.7}$$

If the incoming transition in the history state is an external transition then the set S_{xn} has to be unified with the parent of the history state, because the parent state is left and reentered. In order to create the concatenated exit- and entry actions a list containing the states in S_{xn} is used, which is in ascending order corresponding to the state levels in the state hierarchy. This means that the first state in the list has the lowest and the last state the highest level.

Given the ordered state list $\langle s_1, \dots s_m \rangle$, the concatenated exit actions are given by $\alpha_x = \alpha_x^1 \circ \cdots \circ \alpha_x^m$ and the concatenated entry actions by $\alpha_n = \alpha_n^m \circ \cdots \circ \alpha_n^1$, where α_x^i is state exit action and α_n^i the state entry action in the list at index $1 \leq i \leq m$. The entry actions are in contrast to the exit actions in reverse order, which represents the opposite direction during the iteration through the state hierarchy.

Example 4.6 (History State Removal)
The Keyless Access Controller shown in Figure 4.3 contains two history states H_1 and H_2, where H_1 is contained in the state *CarMoving* and H_2 in *Normal*. The main difference of these two history states is, that the source vertex of the incoming transition of H_1 is one of its ancestor states, while this is not the case for the incoming transitions of H_2.

For this reason different relink and removal strategies are applied to prevent the creation of transitions which guard will never evaluate to true. This means that for the history state H_2 the algorithm described in Equation 4.4 is applied, while for H_1 Equation 4.6 is used.

During the history state removal of H_2 a set of transitions from the simple states contained in the source vertex *WarnLightOn* and *WarnLightOff* to the simple states of the history state parent *AutoUnlocked* and *AutoLocked* are created, as described in Equation 4.4. This means in this case that from both states *WarnLighOn* and *WarnLightOff* two transitions with target vertex *AutoUnlocked* and *AutoLocked*, respectively are created. This means that in sum four transitions for the incoming transitions of the history state are created, but additional transitions for the history state default transition are needed. In this example two additional transitions from the states *WarnLighOn* and *WarnLightOff* to *AutoLocked* are created, which increases the total number of transitions representing the history state behavior from four to six.

The exit- and entry actions have to be added to the transition action corresponding to the left and entered states. This issue is discussed on one of the created transitions mentioned before, which has *WarnLightOn* as source- and *AutoUnlocked* as target vertex and belongs to the transition with signal *evKeyInside* between the vertices *Warning* and H_2 in Figure 4.3.

This transition leaves the states *WarnLighOn* and *Warning* enters *Normal* and *AutoUnlocked*, which reflects the order of the state border crossings of the transition in the graphical representation of the State Machine. Since only the state *AutoUnlocked* has a defined entry action, it is added after the transition action.

For the incoming transition into the history state H_1 with label *extSpeed* contained in the state *CarMoving* four transitions have to be created, because $\textbf{simple}$ (*CarMoving*) contains four states and the transition source vertex *CarMoving* is in $\textbf{ancestors}\,(H_1)$. This means that corresponding to Equation 4.6 transitions forming loops on these simple states have to be created.

Although these created transitions do not leave or enter any other state than the respective simple state, their exit- and entry actions have to be added corresponding to the source vertex of the incoming transition into the history state. This means that all exit- and entry actions from the simple state of a created transition to the source vertex of the transformed transition have to be added.

For the transition creating a loop on the simple state *AutoLocked*, the exit actions of the states *AutoLocked*, *Normal* and *CarMoving* have to be added before and the entry actions of the states *CarMoving*, *Normal* and *AutoLocked* have to be added after the transition action in the given order. Since in this example only *AutoLocked* and *CarMoving* actually have an entry action defined, only these are considered. $\square$

The concatenated exit- and entry actions have to be incorporated into the transition action at this stage of the model transformation, because the transition source vertex of the created loop transitions is the final simple state instead of the initially composite state. This means that the information about the source vertex before the removal of the history state is lost and therefore prevents the incorporation of the exit- and entry actions later on.

4.2.2. State Machine Flattening

Given an UML State Machine where all pseudo states are removed, the SM vertices consists only of composite- and simple states, which represent the state hierarchy. Since an ESTS is flat, meaning that its states must not contain other states, the SM has to be *flatted* by removing the state hierarchy. In general the flatting is performed by relinking transitions and the removal of the composite states, where relinking is the change of the source- or target-vertex of a transition to another vertex. After the flatting the SM consists only of simple states and transitions, which can be transformed in a straightforward task into an ESTS.

Since during the State Machine flattening all composite states are removed, only its simple states remain which will be transformed into an ESTS in the end. For this reason the transitions, which have a composite state as source- or target vertex, have to be relinked to their contained simple states.

According to the UML specification a transition having a composite state as source vertex actually represents an outgoing transition from each of its contained simple states. This means that an outgoing transition of a composite state represents a set of transitions, where each transition has the same target vertex but a source vertex in $\mathbf{simple}\,(s_c)$ of the composite state s_c. Since these transitions are not explicitly modeled in a SM containing a state hierarchy, additional transitions have to be created during the flattening.

In contrast a transition having a composite state as target vertex represents only one transition after the flattening. The reason is that due to the usage of default transitions the default simple state – and therefore the target vertex of the transition after the relink – is uniquely defined.

In the remainder of this section we use s and s' as source- and target-vertex of a transition before and s_f and s'_f as source- and target-vertex after the relink. The rule describing the creation of transitions during their relink is shown in Equation (4.8), where a transition set is created containing the additionally created transitions depending on their source vertex.

$$\frac{s \xrightarrow{l,\varphi,\alpha} s'}{\left\{ s_f \xrightarrow{l,\varphi,\alpha_f} s'_f \mid s_f \in \mathbf{simple}\left(s\right) \wedge s'_f = \mathbf{default}\left(s'\right) \right\}} \tag{4.8}$$

However, the concatenated exit- and entry actions in α_f, which are added to the transition action in Equation (4.8), still have to be defined. In general they are calculated in the same way as shown in the section above, where a history state caused transition self loops. Unfortunately this calculation cannot be used likewise in this case, because the considered entry- and exit actions depend on the transition source- and target vertex and their level in a state hierarchy.

For this reason we discuss in this section the different concatenations of exit- and entry actions corresponding to the transition vertices, where we focus only on external transitions. The actions of external transitions are calculated similarly to local transitions, where the entry- and exit action of a local transition source vertex can be neglected. The reason is that a local transition does not cross the border of its source vertex in the graphical representation and therefore its target vertex can only be one of the simple states contained in the source vertex.

In order to create the concatenated exit- and entry actions the considered states have to be identified first. For this reason the three sets S_x, S_n and S_d are built, where S_x is the set of exited states, S_n is the set of entered states and S_d is the set of states containing a default transition which action has to be taken into account.

Since a transition can connect two arbitrary states, the considered exit-, entry- and default transition actions depend on the relative position of the states to each other in the state hierarchy. By relative position we mean the number of state hierarchy levels, which have to be taken into account in order to connect these states. A considered hierarchy level is similar to a transition crossing a state border in the graphical representation of a State Machine. This means that depending on the transition direction the actions corresponding to the state border crossings have to be taken into account.

Depending on the position of the connected states, it is not always necessary that all states in the state hierarchy from the simple state up to the root state have to be considered. The reason is that for example two simple states are children of the same parent state, which means that a transition connecting these two states does not cross the border of their or any other parent state. Consequently the exit- and entry actions of these parent states can be neglected for such transitions.

The parent state with the highest level in the state hierarchy containing both, the source- and target vertex of a transition, is called the least common ancestor s_{lca}. The calculation of the least common ancestor s_{lca} is shown in Equation (4.9), where $max_L\left(S'\right)$ returns the state with the highest state hierarchy level in $S' \subseteq S$ and the non existing state s_ϵ if $S' = \emptyset$.

$$s_{lca} = max_L\left(\left(\mathbf{ancestors}\left(s_f\right) \cup \{s\}\right) \cap \left(\mathbf{ancestors}\left(s'_f\right) \cup \{s'\}\right)\right) \tag{4.9}$$

In Equation 4.9 the least common ancestor s_{lca} is the state with the highest level in the set of states representing the common ancestors of the transition source- and target vertex after it has been relinked. These sets are unified with their initial source- and target vertex, because in the case the source vertex is an ancestor of the target vertex (or vice versa) the least common ancestor might have been at a lower level before the transition relink. Since this information is lost during the relink it is necessary to consider it explicitly in this calculation.

Given the least common ancestor, the set of states which exit- and entry actions are taken into account are given by Equation (4.10) and (4.11), respectively.

$$S_x = \mathbf{ancestors}\left(s_f\right) \setminus \mathbf{ancestors}\left(s_{lca}\right) \tag{4.10}$$

The set S_x contains in general the states between the least common ancestor and the transition source vertex in the state hierarchy. If no least common ancestor exists then all ancestors of the state s_f are in the result set, because $\mathbf{ancestors}\left(s_\epsilon\right) = \emptyset$ as defined above and therefore does not have any impact in this equation.

$$S_n = \mathbf{ancestors}\left(s'_f\right) \setminus \mathbf{ancestors}\left(s_{lca}\right) \tag{4.11}$$

The same as for the set of states which exit actions are incorporated is true for the states which entry actions have to be taken into account, where the states between the transition target vertex and the least common ancestor are considered.

In contrast to the two equations shown above Equation (4.12) contains the states which contained default transitions define the actions to be considered. In this case the built set of states does contain the states between the transition target vertex s'_f and the transition target vertex s' before it was relinked.

$$S_d = \mathbf{ancestors}\left(s'_f\right) \setminus \mathbf{ancestors}\left(s'\right) \tag{4.12}$$

The reason is that a default transition has only to be considered for incoming transitions of the parent of its initial pseudo state. In this case the default transition defines the target vertex for these incoming transitions and can also be an incoming transition in another parent state containing a default transition of its own, which is one of its children.

This mechanism of defining the parent's default child state causes an alternating sequence of state- and default transition-actions, which has to be reflected in the transition actions after the relink. The action lists added before and after the original transition action are built from the exit actions of the states in S_x, the entry actions of the states in S_n and the actions of the default transition contained in S_d.

These lists are built in the same way as explained in the Section 4.2.1, where the states contained in these sets are ordered by their hierarchy level meaning that S_x is sorted descending and

S_n and S_d are sorted ascending. Given these action lists the transition action α_f is calculated as shown in Equation (4.13), where superscripts denote the list index and the subscript x, n and d indicate an exit-, entry- and default transition action, respectively.

$$\alpha_f = \underbrace{\alpha_x^n \circ \cdots \circ \alpha_x^1}_{\text{exit actions}} \circ \alpha \circ \underbrace{\alpha_n^1 \circ \cdots \circ \alpha_n^k \circ \alpha_d^k \circ \alpha_n^{k+1} \circ \cdots \circ \alpha_d^{m-1} \circ \alpha_n^m}_{\text{entry- and default transition-actions}} \tag{4.13}$$

As defined in Equation (4.13) the exit actions $\alpha_x^n, \ldots, \alpha_x^1$ are inserted before the original transition action α and the entry actions $\alpha_n^1, \ldots, \alpha_n^m$ and default transition actions $\alpha_d^k, \ldots, \alpha_d^m$ are interleaved and added afterwards, where $n, m, k \in \mathbb{N}_1$ are the respective number of actions.

Example 4.7 (Concatenation of Entry- and Exit-Actions)
The main difference to the concatenation of exit- and entry-actions in comparison to the approach presented for history states in Example 4.6 is that also actions defined on default transitions have to be considered. Given the transition between the states *CarStopped* and *CarMoving* with label *extSpeed*, two transitions will be created during the state machine flattening of which we discuss the one with *CarUnlocked* as source vertex in more detail here.

Since the target vertex of this transition is the state *AutoUnlocked*, which is defined by the default transitions in state *CarMoving* and *Normal*, their actions have to be added to the transition action as well. This means that for this transition $S_x = \{\mathit{CarStopped}, \mathit{CarUnlocked}\}$, $S_n = \{\mathit{CarMoving}, \mathit{Normal}, \mathit{AutoUnlocked}\}$ and $S_d = \{\mathit{CarMoving}, \mathit{Normal}\}$. The transition action is built corresponding to Equation 4.13, where we treat the sets S_x, S_n and S_d as lists while maintaining the order given in this example.

This means that α_x^2 and α_x^1 are the exit actions of the states *CarUnLocked* and *CarStopped*, α_n^1, α_n^2 and α_n^2 are the entry actions of the states *CarMoving*, *Normal* and *AutoUnlocked* and α_d^1 and α_d^2 are the default transition actions of the states *CarMoving* and *Normal.* This means that for this example the indices in Equation 4.13 have the values $n = 2$, $k = 1$ and $m = 3$. $\square$

After the transition relink has been performed the original transitions as well as the state hierarchy can be removed. This means that all composite states and their outgoing- and incoming-transitions are deleted in order to finally flatten the State Machine, which then can be transformed into an ESTS.

However, due to the flattening process the information of the state hierarchy as well as the source- and target vertex of the original transitions is lost, but is needed for the creation of timing groups during the ESTS model transformation and to achieve traceability between the hierarchical and the flat UML State Machine.

For this reason a *flattening mapping* of model elements existing before and after the flattening has to be created, which preserves the state hierarchy data and maps the transitions of the hierarchical State Machine to their corresponding transitions in the flat State Machine. We mean by hierarchical State Machine the State Machine before and by flat State Machine the State Machine after the relink of the transitions and the removal of the composite states.

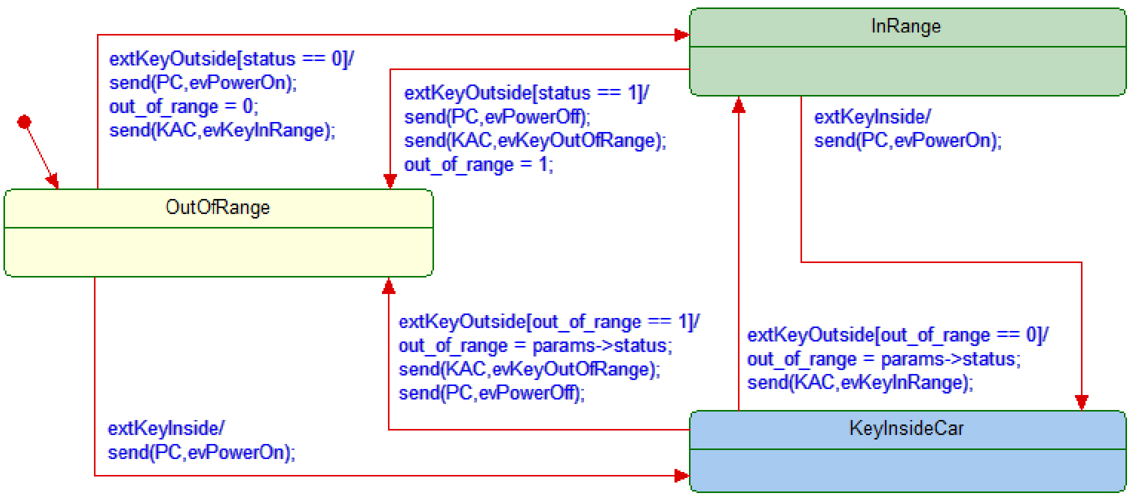

InRange
OutOfRange
KeyInsideCar
extKeyOutside[status == 0]/
send(PC,evPowerOn);
out_of_range = 0;
send(KAC,evKeyInRange);
extKeyOutside[status == 1]/
send(PC,evPowerOff);
send(KAC,evKeyOutOfRange);
out_of_range = 1;
extKeyInside/
send(PC,evPowerOn);
extKeyOutside[out_of_range == 1]/
out_of_range = params->status;
send(KAC,evKeyOutOfRange);
send(PC,evPowerOff);
extKeyOutside[out_of_range == 0]/
out_of_range = params->status;
send(KAC,evKeyInRange);
extKeyInside/
send(PC,evPowerOn);

Figure 4.7.: Flat State Machine of the KLD.

Transitions in the flattening mapping of the hierarchical State Machine can be mapped to none, one or multiple transitions in the flat State Machine. The reason is that during the State Machine flattening transitions have to be removed, relinked or additional ones have to be created.

Example 4.8 (Flattening Mapping)
A default transition which initial pseudo state is a child of a composite state is removed during the flattening, but is mapped to the incoming transitions of this composite state. In the KAC such default transitions are contained in the composite states *CarStopped*, *CarMoving*, *Normal* and *Warning* as shown in Figure 4.3. The default transition in *CarMoving* of the hierarchical State Machine maps for example on the two incoming transitions with label *extSpeed* and their respective source vertices *CarLocked* and *CarUnlocked* of the corresponding flat State Machine. These transitions are created during flattening of the transition with the label *extSpeed* having *CarStopped* and *CarMoving* as source- and target vertex, respectively.

A transition in the hierarchical State Machine maps to exactly one transition in the flat State Machine if for example its source- and target-vertex is a simple state, meaning that it can be directly represented in the flat State Machine. Such a transition is shown in the KAC State Machine and has the label *extAckWarning* and connects the simple states *WarnLightOn* and *WarnLightOff*.

An one-to-many transition mapping can have various reasons like composite states serving as source- and history states as target vertex, because both situations require the creation of additional transitions. Examples for such transitions are the incoming transitions of the history state H_2 contained in the state *Normal*, which require a one-to-many mapping for both reasons stated before and the outgoing transitions of the states *CarStopped* and *Normal*. □

Example 4.9 (Flat State Machine)
The flat State Machine of the KLD is shown in Figure 4.7, which was created from its corresponding hierarchical State Machine depicted in Figure 4.4. The flat State Machine does not contain

the condition state anymore and has the same amount of transitions. The reason is that although the composite state *KeyOutsideCar* has an outgoing transition with label *extKeyInside*, which requires the creation of an additional transition during the flattening, the incoming transition of the condition state is removed. The removal of this transition is performed after its signal, guard and action has been merged with the outgoing transitions of the condition state, which are relinked to the source vertex *KeyInsideCar* of its incoming transition. □

4.2.3. State Machine to ESTS Transformation

In this section we explain the transformation of a flat UML State Machine into an ESTS, where in a flat State Machine no state hierarchy nor pseudo states remain. For this reason the model transformation is a straight forward task, where in general a flat State Machine can be directly mapped on an ESTS.

The main difference between a State Machine and an ESTS is that in an ESTS the inputs and outputs are defined by distinct transitions, whereas in a State Machine outputs are specified in transition actions. In addition the ESTS does not support default transitions and requires timing groups for delay transitions, which are the equivalents of timed transitions in UML State Machines.

These differences are discussed in more detail and a transformation solution is proposed in the following sections.

Transition Transformation

In this section the transition transformation of a flat State Machine to an ESTS is explained, which means that the State Machine does not contain a state hierarchy nor pseudo states. The transition transformation is performed after all states from the flat State Machine have been created in the ESTS, which is a straight forward task and does not require any adoptions during the transformation. This means that for every state in the UML SM a corresponding state in the ESTS is created.

A State Machine transition can be transformed into a list of ESTS transitions as shown in Equation (4.14), where $s_f \xrightarrow{l,\varphi,\alpha} s'_f$ is a flat State Machine transition, $(s_i, l', \varphi, \alpha_i, p, d, s_{i+1})$ is an ESTS transition with list index $i \in \mathbb{N}_1$, l' is the created unique transition label, χ_i is an output signal and α_i is an attribute update function.

$$\frac{s_f \xrightarrow{l,\varphi,\alpha} s'_f}{((s_1, l', \varphi, \alpha_1, p, d, s_2), \ldots, (s_m, \chi_m, \top, \alpha_m, p, 0, s_{m+1}))} \tag{4.14}$$

An UML SM transition might have to be represented by a list of ESTS transitions, because of the distinction between input- and output-signals, which dependencies and creation will be explained in the next section in more detail. The source- s_1 and target- s_{m+1} state of the ESTS transition list are those states corresponding to the State Machine states s_f and s'_f, respectively.

The transitions are connected to each other, which means that the target state of one transition is also the source state of its successor transition.

The label of the SM transition can either be an input, a timeout or the empty label, whereas the transformation of timed transitions is discussed later. Since a send command in a transition action defines the output signal and the receiving object (which behavior is described by a State Machine), a signal and State Machine combination is used in order to create a unique output label for an ESTS. Correspondingly the input signals have to match these unique output signal labels, which means that an input signal label l' is the concatenation of the ESTS object- and the output signal name.

State Machine transitions with an empty label ϵ are represented by completion transitions in an ESTS. For this reason such a transition can be transformed straight forward by creating a new transition in the ESTS using the completion label γ.

The guard φ can be reused without any changes in its semantics, but some syntactical adoptions might be necessary, depending on the syntax used in the State Machine. However, this can be achieved by a direct mapping of the operators and is also straightforward. Since the UML transition guard is only evaluated once it is only necessary to use this guard on the first ESTS transition, where for the others their guard always has to evaluate to true.

The used priority p depends on the level of the transition source vertex s in the state hierarchy of a SM, which defines the execution priority in UML. For this reason the defined state hierarchy level can be directly assigned to ESTS transition priority p.

The UML State Machine as defined in [9] does not provide any information about the execution duration. However, due to the use of annotations as described in Section 4.1.2 the execution duration can be provided in a hierarchical State Machine, where such an annotation belongs to a transition and has to be copied to every corresponding transition created during the flattening. Afterwards the defined values for the execution duration can be assigned to the ESTS transitions. The given duration has to be assigned only to the first ESTS transition, where for the others the execution duration 0 is used.

Action Split

Since an UML SM transition action α consists of a list of attribute update functions and send commands, which cannot be represented by an ESTS directly, the action has to be split in such a way that input- and output signals are separated.

Given the fact that α is an arbitrary sequence of update functions ρ and send actions χ, α has to be split before every contained send action χ. This means that an action $\alpha = \alpha_1 \circ \cdots \circ \alpha_m$ is split into action subsequences α_i such that the first subsequence of the action has the form $\alpha_1 = \rho_1, \ldots, \rho_n$ and the remaining ones the form $\alpha_i = \chi, \rho_1, \ldots, \rho_n$. This means that all subsequences except the first one begin with a send action followed by a number of attribute update functions.

Therefore an additional transition for each action subsequence starting with index $2 \leq i \leq m$ has to be created in the ESTS. This has already been indicated in Equation (4.14), where α_i is an action subsequence for which an additional transition is created. The creation of the transition list as described above ensures that an output- always follow an input- or completion transition, which prevents an input/output conflict causing a non-deterministic behavior in the resulting ESTS.

Default Transitions

After the State Machine flattening only one default transition remains, which defines its initial state and cannot be transformed into an ESTS directly. The reason is that the initial state of an ESTS is defined by the initial configuration q_0 instead of a dedicated transition.

Since a default transition can have an action, a list of transitions might have to be created as discussed above during the transformation. For this reason an additional state is created in the ESTS, which is its initial state and represents the initial pseudo state of the default transition. Since the default transition has an empty label the first transition of the created transition list is labeled with the completion label and has a guard always evaluating to true to maintain the semantics of the State Machine.

Timed Transitions

For every timed transition t of an hierarchical UML State Machine a timing group has to be created in the resulting ESTS. As defined in Section 2.1 a timing group $g \in G$ consists of a set of states S_g, a set of delay- T_g and a set of clock reset transitions T_r.

The states of a timing group S_g depend on the source state s of the timed transition t, where all its simple states $S_g = \mathbf{simple}(s)$ are contained. This means that the contained states are basically defined by the state hierarchy of a SM.

The timing group set of delay transitions T_g contains a delay transitions for each timed transition of the hierarchical State Machine and those created during the flattening. Since for all simple states of the transition source state s a transition is created during the flattening, all states in S_g have an outgoing delay transition in the corresponding ESTS.

In order to reset the timing group clock, a set of clock reset transitions T_r is required. This set consists of all incoming transitions in the source state s of the timed transition t. Similar to the delay transitions, also all transitions in the hierarchical State Machine and their additionally created transitions during the flattening of the State Machine have to be considered.

Transformation Mapping

The transformation of an UML State Machine requires a *transformation mapping* in order to maintain the traceability between the State Machine and ESTS elements. The reason is that a State Machine transition can belong to multiple transitions and states in an ESTS and therefore they do not relate to each other directly.

The transformation mapping is similar to the flattening mapping, which allows a one-to-many transition and state linkage. Due to the creation of these mappings the ESTS elements can be uniquely identified in the UML State Machines, which allows the incorporation of information based on the much more detailed ESTS level. For this reason test engineers only need to interact with the UML State Machines even if ESTS specific data shall be provided, which allows a much more comfortable modeling.

Input Completeness

According to the UML specification a received signal is dropped if it does not have an outgoing transition with a corresponding input label. This means that such a signal reception does not cause a state nor attribute value change. This is similar to the requirement that an ESTS is input complete.

However, due to the model transformation and the additional created states in the ESTS during the action split, only those states which directly correspond to the states of the flat State Machine have to be input enabled.

These input enabled states are also the quiescent states of an ESTS, which do not have any outgoing output transitions. The reason is that during the model transformation of a State Machines into an ESTSs output transitions are only created after an input- or completion transition. This means that in an UML State Machine always an input signal has to be received first, before an output signal can be sent, which is the precondition for quiescence.

Example 4.10 (State Machine Transformation)
The transformed ESTS of the KLD is shown in Figure 4.8, where we use a similar syntax for the transitions as defined for the State Machines. This means a transition description starts with its signal preceded by a question- or exclamation-mark to denote an input (?) or output (!), respectively and for completion transitions no label is shown at all. After the label follows the guard surrounded by squared brackets, which we omit for inputs and outputs in this figure if it always evaluates to true. Finally the attribute update function is shown, which is proceeded by a slash (/) if it is not the identity function `id`.

In Figure 4.8 the initial state of the ESTS has a double line border and is filled red, while the quiescent states are filled blue and the remaining ones in light orange. This means for this example that the states *OutOfRange*, *InRange*, *KeyInsideCar* and *Init* are quiescent corresponding to Definition 2.8.

The state *Init* is the initial state of the ESTS and therefore part of it initial configuration containing also the initial attribute- and clock values. Since a default transition can have an action a corresponding completion transition is created, which connects the states *Init* and *OutOfRange* and has a guard [*true*] always evaluating to true in this example.

The action split explained in Section 4.2.3 causes the creation of additional states in the ESTS. These states are, because of the used State Machine transformation into an ESTS, never quiescent and are depicted light orange filled in Figure 4.8. Their names start with an *O* followed by an unique identifier, which means that the states *O1* to *O9* are the non-quiescent states of this ESTS.

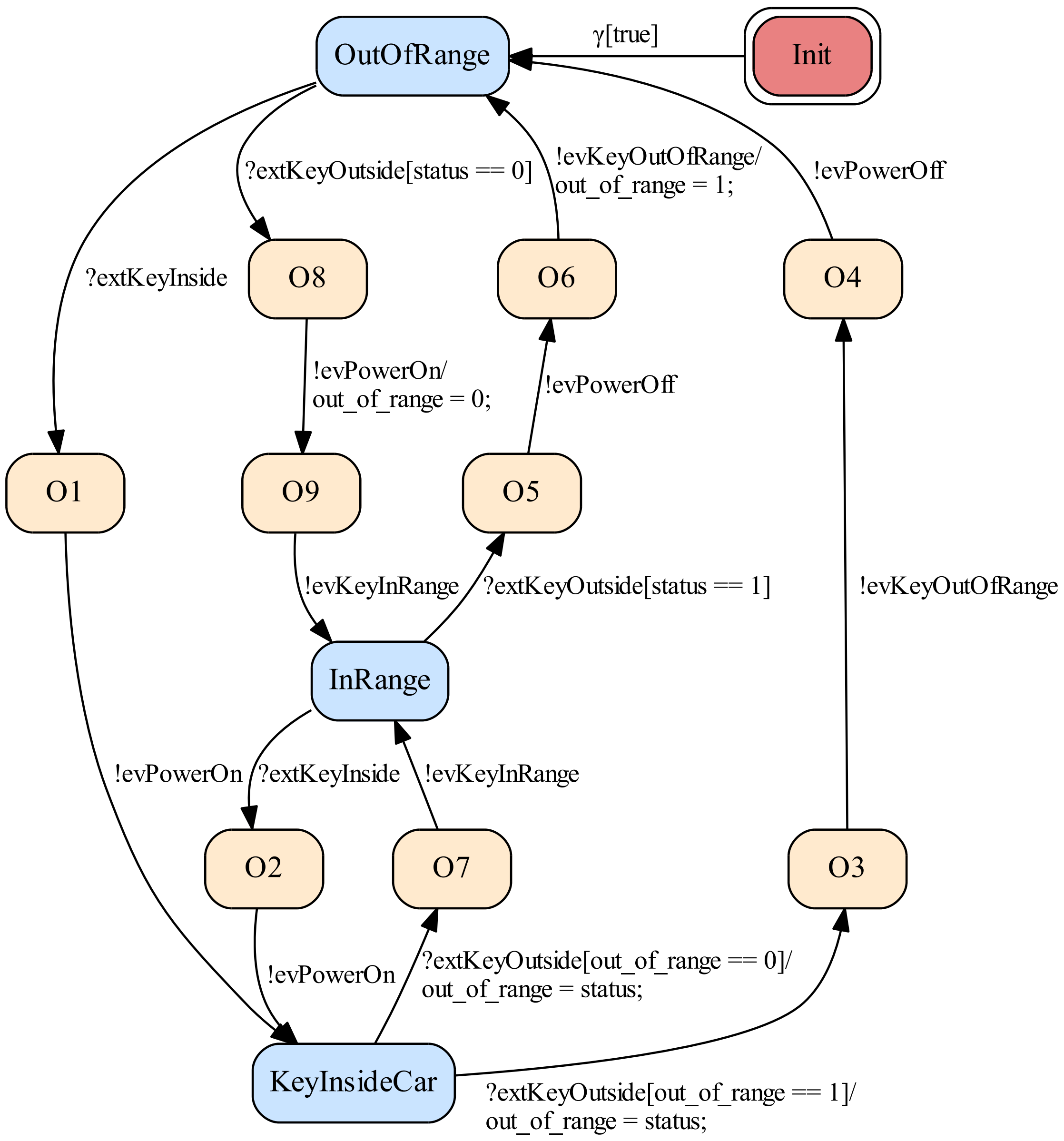


Figure 4.8.: Transformed ESTS of the KLD State Machine.

For example the transition having the label *extKeyOutside* and which lies between the states *OutOfRange* and *InRange* in the flat State Machine in Figure 4.7 causes the creation of two additional states *O8* and *O9* in the ESTS, because it contains the two send actions *send(PC,evPowerOn)* and *send(KAC, evKeyInRange).* Each of this send actions is represented by a corresponding output transition in the ESTS, where the first send action is leads to the transition with source state *O8* and the second to the transition with source state *O9.* The contained update action of the State Machine transition, which shall be executed between the send actions, is realized by an attribute update function of the first ESTS transitions. □

Chapter 5

Industrial Application

Parts of this chapter are taken from ´Test Case Generation for Integration Testing of Electronic Control Unit Networks´ [30] which is joint work with Stefan Widder.

In this chapter the industrial setting and its imposed requirements on the presented approach are discussed. The goal of this work was the creation of an integrated tool chain allowing the test case generation for integration testing of an ECU network. The integration testing of such networks, where implemented functionalities are distributed over multiple ECUs, is a difficult and laborious task. The reason is that integration testing is performed as black box test, where the controllability of the SUT as well as of the Test Automation (TA) are limited. Especially in the case of time dependent behavior these limitations cause difficulties, because e.g. time advancement cannot be controlled at all but would be needed to guarantee that a certain test scenario is executed.

5.1. Integrated Tool Chain

Since this approach is an extension to a well established testing process, the developed test generation tool STATE BASED SYSTEM TEST AND SIMULATION (STATION) presented in Section 5.3 had to be incorporated in the existing tool chain. In this development process RHAPSODY was already used for modeling UML State Machines and shall also be the only interface to the test engineer in the presented approach. This means that for the test case generation the test engineer needs to focus on one tool and one representation of the created models.

The test execution and the verification of the obtained results in order to determine a test verdict is realized in the TA framework EXTENDED AUTOMATION METHOD (EXAM) providing a set the *test automation functions*. A test automation function is basically a software function executed on the TA, which controls the SUT and the test environment. These test automation function are called one after another in order to perform a test.

The used test automation framework EXAM allows the execution of test cases described as UML Sequence Charts and UML Activity Diagrams, where we focus on Sequence Charts in this

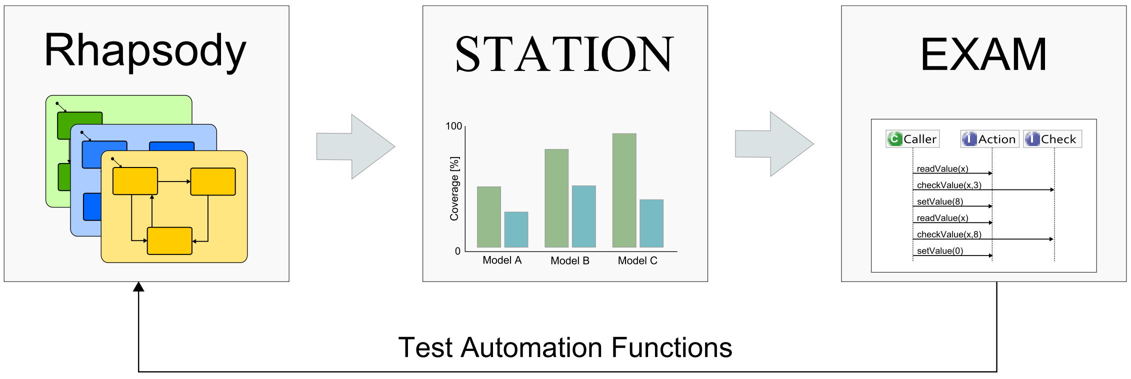


Figure 5.1.: Industrial Tool Chain

work. The reason is that due to the usage of deterministic models and a deterministic communication scheme the generated test cases are also sequences and therefore the usage of Sequence Charts is sufficient.

A schematic of the tool chain is shown in Figure 5.1, where the annotated UML State Machines are exported from RHAPSODY and imported and transformed into an ESTS in STATION. An annotated State Machine does not only describe the behavior of a system component, but also contains test definitions and test automation functions.

The presented approach does not use the test cases generated on the basis of the ESTSs itself, but uses the test automation functions mapped to the UML states and transitions in order to generate test sequences executable by the TA. Since the test automation functions are incorporated in the UML State Machines, it is crucial that a generated test case of an ESTS can be uniquely mapped back on the corresponding UML State Machines. This is achieved by the model transformation explained in Section 4.2.

5.2. Rhapsody

RHAPSODY is an UML modeling tool, which allows code generation from State Machines, Sequence Charts and Activity Diagrams. This code can be executed and for example be used to animate the created State Machines. An animated State Machine highlights its current state and the last executed transition, which allows the model developers to comprehend the modeled behavior.

In order to generate test cases from UML State Machines which are executable by a given test automation system, the TA and the SMs have to be linked to each other. This means that the behavior described in the State Machines have to be represented by the test automation functions in the generated test case. This is achieved by annotated State Machines in this approach, which are State Machines where the test automation functions and test definitions have been incorporated. The annotations are provided using tags, which consist of a key/value pair. Since in RHAPSODY tags can only have strings as keys and values a syntax and naming convention has been introduced and used for the annotations.

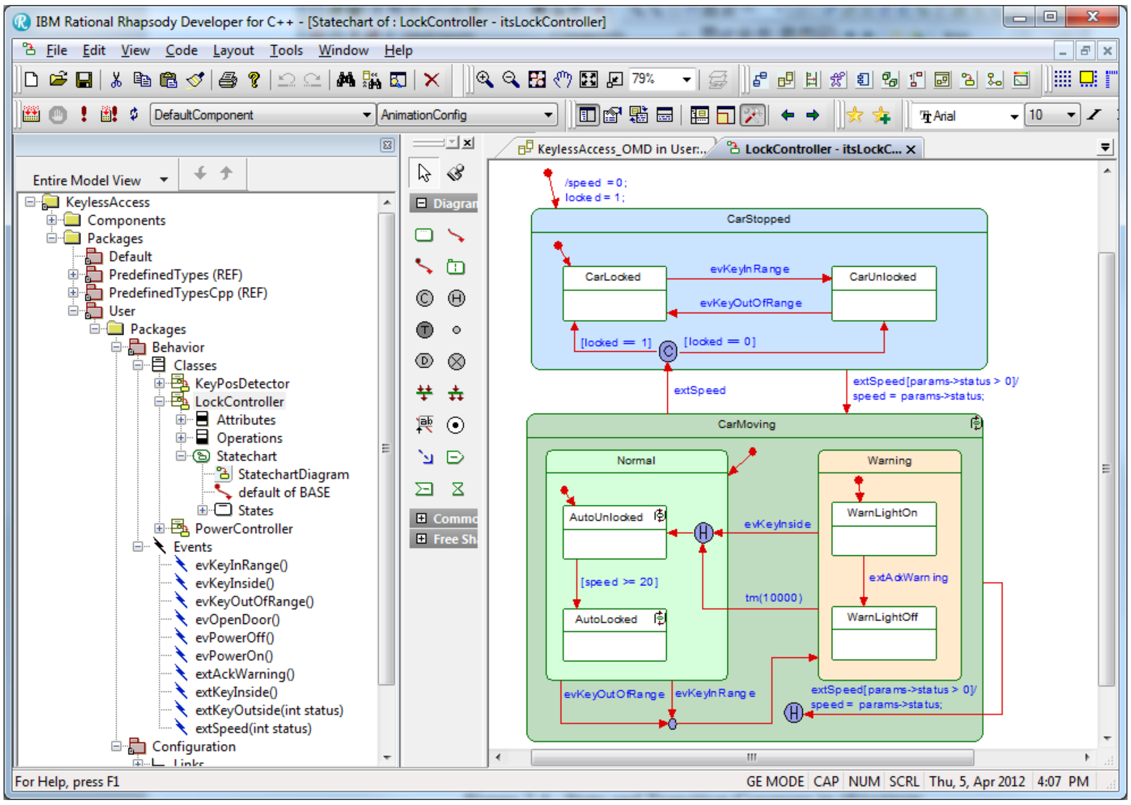


Figure 5.2.: UML State Machine in RHAPSODY.

This mechanism allows to link test automation functions and test definitions on states and transitions, where the annotations can be interpreted by the test generation tool STATION. Since the annotations are added to states and transitions, which correspond to an ESTS path due to the created mappings during the State Machine flattening and transformation, the generated test cases represent paths through the UML State Machines. These paths are used to create sequences of the annotated test automation functions, which can be executed by the corresponding TA.

The same mechanism is used to define a test purposes in the State Machines, where the transitions and states which shall be covered by a generated test case are defined using annotations. These annotations provided in tag values have to contain a property specifying the order in which the states and transitions have to be covered. A test definition provided in this way is similar to the test purpose defined in Section 3.1 and can be easily transformed into this structure, where in general each annotation corresponds to a generation step.

Example 5.1 (Rhapsody)
Figure 5.2 shows a screenshot of RHAPSODY showing the three main parts of the tool, namely the model browser, model element tool bar and the modeling pane. The model browser contains a tree view of the model and all its involved elements and signals and is depicted on the leftmost side in the screenshot.

The model element tool bar is shown right to the model browser and contains the elements available for State Machine modeling. Therefore it contains states, transitions and pseudo states, whereat the pseudo states supported by RHAPSODY do not follow the UML specification [9] precisely. The entries of the model element bar are used to create the desired model in the modeling pane shown on the leftmost side of the screenshot. □

In order to allow a further processing of the UML model, it has to be exported in a machine readable format. This export is realized by the tool REPORTERPLUS, which is shipped with RHAPSODY and allows the creation of model reports based on templates, where a template defines the content of the report.

For this reason a corresponding template has been created, which produces a Extensible Markup Language (XML) file containing the model elements. The exported model elements can have properties, which can be accessed during the export in order to provide all information needed for the test case generation.

5.3. STATION

The tool STATION is a prototypic implementation of the test generation strategies introduced in this work, where a detailed description of the approach can be found in Chapter 2. STATION is entirely written in the programming language JAVA and allows the generation of test cases based on deterministic ESTSs.

Since STATION was integrated into an existing tool chain the interfaces to the other tools had to be defined and implemented. This means that for the import of the UML State Machines a file parser for the XML export created by REPORTERPLUS was developed. On the other side the generated test cases are exported from STATION into an XML file, which can be read and processed by an EXAM import plugin.

This means that after the file parsing of the REPORTERPLUS export the read UML model is flattened and transformed into an ESTS as described in Section 4.2. During the transformation the test definitions annotated in the State Machines are mapped on the ESTS for the test case generation.

During this mapping the parameters defined in the annotations for the state space limitation techniques presented in Section 3.5 are interpreted. This allows a test engineer to control the test generation for each generation step via these test definitions. In these annotations several parameters for each limitation technique can be provided.

Example 5.2
Figure 5.3 depicts a STATION screenshot showing the state- and transition coverage of a generated test case on a flattened UML State Machine, where the covered elements are presented green.

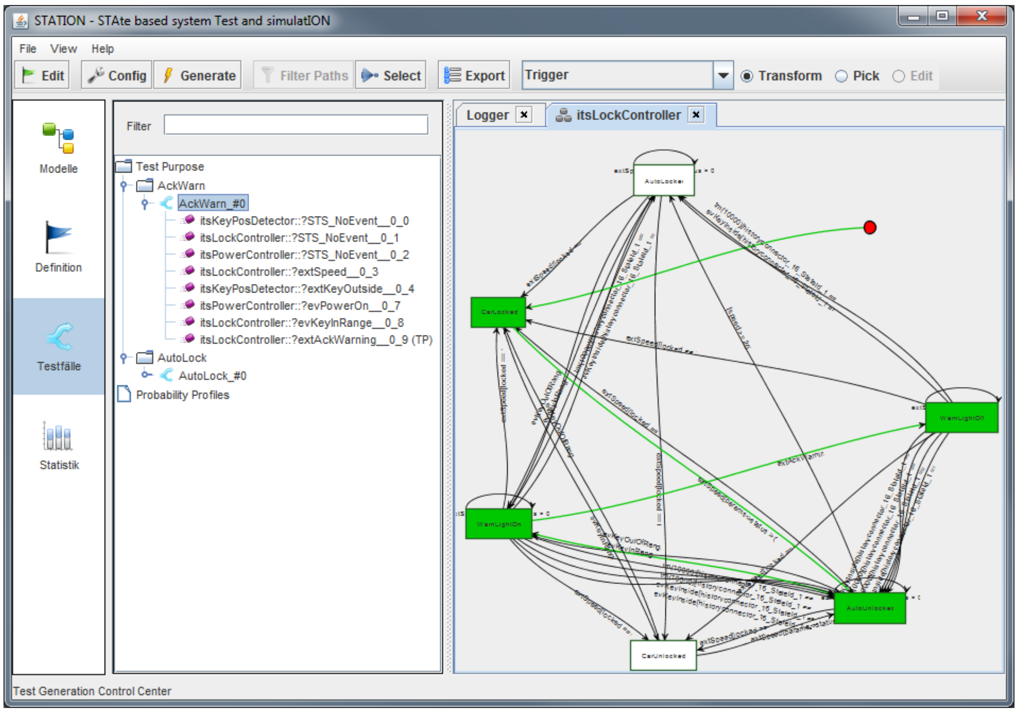

STATION - STAte based system Test and simulatION
File View Help
Edit
Config
Generate
Filter Paths
Select
Export
Trigger
Transform
Pick
Edit
Modelle
Definition
Testfälle
Statistik
Filter
Test Purpose
AckWarn
AckWarn_#0
itsKeyPosDetector::?STS_NoEvent__0_0
itsLockController::?STS_NoEvent__0_1
itsPowerController::?STS_NoEvent__0_2
itsLockController::?extSpeed__0_3
itsKeyPosDetector::?extKeyOutside__0_4
itsPowerController::?evPowerOn__0_7
itsLockController::?evKeyInRange__0_8
itsLockController::?extAckWarning__0_9 (TP)
AutoLock
AutoLock_#0
Probability Profiles
Logger
itsLockController
AutoLocked
CarLocked
WarnLightOff
WarnLightOn
AutoUnlocked
CarUnlocked
Test Generation Control Center

Figure 5.3.: Test coverage on a flat UML state machine in STATION.

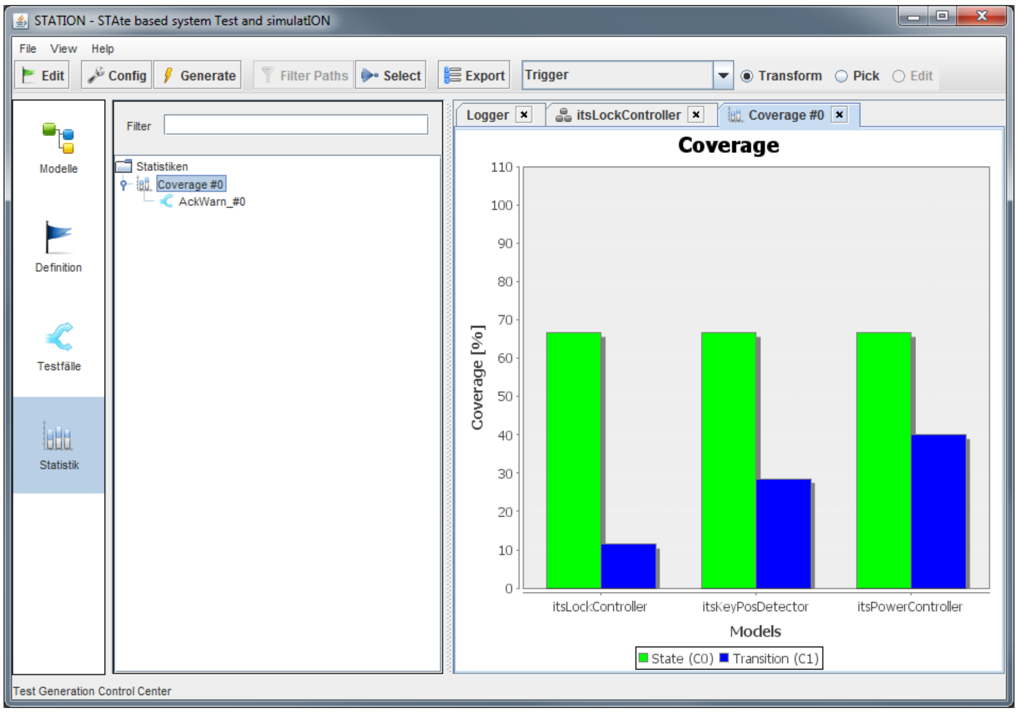


Figure 5.4.: State and transition coverage statistics shown in STATION.

The reason for the display of the flattened UML SM is that it is the most compact representation of all available transitions. This means that no split of inputs and outputs has to be shown on the one hand and on the other hand no transition is hidden due to pseudo states or state hierarchy. For this reason it shows the full complexity of the model.

The screenshot in Figure 5.4 shows the coverage of the same test case mentioned above, but from a different point of view. In this image the state- and transition coverage of each State Machine is presented in form of statistics, where the relative number of covered states and transitions is shown. The statistics are based on the single test case used before, but can also consist of multiple different test cases in order to allow the test engineer to get an overview of those test cases needed for a specific task. □

The incorporation of the test automation functions into the UML State Machines and the mapping of its states and transitions on the corresponding ESTS elements allows the creation of test cases, which are directly executable by a TA like EXAM.

5.4. EXAM

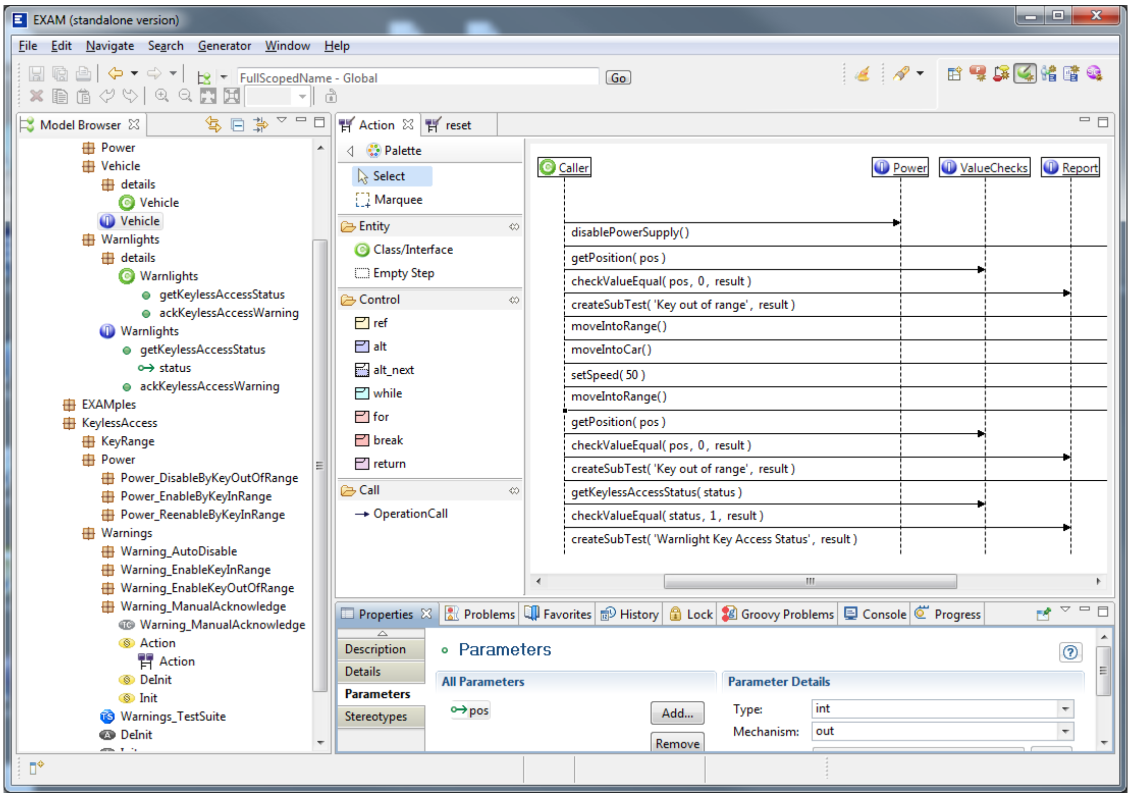
The TA used in this industrial setting is EXAM, which provides test automation functions allowing to interact with the SUT by controlling the test environment. In this approach the test environment is a HiL system, which allows e.g. measurement and verification of electrical signals, the creation of bus messages and the capturing of trace files, where a trace file contains all bus messages with parameter values and time stamps.

The TA provides an implementation of the test automation functions for the used HiL systems, which allows a fully automated test execution. This automated test execution approach has the advantage that the test cases can be implemented similar to software functions and therefore ensure the reproducibility of the test execution. This means that the test automation system functions as adapter between the test cases being a piece of software and the SUT, which in this case consists of a set of hardware components.

In EXAM the test cases are modeled in UML Sequence Charts, which consists of a list of test automation functions. These Sequence Charts can be combined with parameter sets, which are used to parameterize the test case in order to allow a reuse of the same test case with different parameter values. In addition EXAM follows a standard object oriented programming language paradigm by allowing the separation of interfaces and implementations. This is very beneficial in case different HiL systems are used, because the specific underlying implementations can be exchanged without influencing the already existing test cases.

Example 5.3
Figure 5.5 shows a screenshot of EXAM, where a simple test sequence consisting of test automation functions is depicted. In this example the test automation functions are provided by multiple interfaces like *Power*, *ValueChecks* and *Report*, which are called by the class *Caller*.

The HiL specific implementations of the provided interfaces by the TA have to be selected before test cases can be executed. These specific implementations are grouped in a so called



```
Power
Vehicle
  details
    Vehicle
  Vehicle
Warnlights
  details
    Warnlights
      getKeylessAccessStatus
      ackKeylessAccessWarning
  Warnlights
    getKeylessAccessStatus
      status
    ackKeylessAccessWarning
EXAMples
KeylessAccess
  KeyRange
  Power
    Power_DisableByKeyOutOfRange
    Power_EnableByKeyInRange
    Power_ReenableByKeyInRange
  Warnings
    Warning_AutoDisable
    Warning_EnableKeyInRange
    Warning_EnableKeyOutOfRange
    Warning_ManualAcknowledge
      Warning_ManualAcknowledge
        Action
          Action
        DeInit
        Init
    Warnings_TestSuite
    DeInit
```

```
disablePowerSupply()
getPosition( pos )
checkValueEqual( pos, 0, result )
createSubTest( 'Key out of range', result )
moveIntoRange()
moveIntoCar()
setSpeed( 50 )
moveIntoRange()
getPosition( pos )
checkValueEqual( pos, 0, result )
createSubTest( 'Key out of range', result )
getKeylessAccessStatus( status )
checkValueEqual( status, 1, result )
createSubTest( 'Warnlight Key Access Status', result )
```

Parameters

All Parameters: pos

Type: int

Mechanism: out

Figure 5.5.: Test Sequence in EXAM.

System Configuration, which has to be predefined and tailored to the specific requirements of the underlying system. □

Chapter 6

Experimental Results

Parts of this chapter are taken from ´An Extended Symbolic Framework for Systematic Test Case Generation´ which is an ongoing work with Bernhard K. Aichernig and Franz Wotawa and ´Test case generation in practice for communicating embedded systems´ [22] which is joint work with Franz Wotawa.

The applicability of the presented approach is shown based on two illustrative examples and on two industrial use cases from the automotive industry, which are described in Section 6.1 and Section 6.2, respectively.

The illustrative examples are used to show the influence of basic State Machine structures on the number of generated test cases. The structure of a State Machine is given by the used state hierarchy and pseudo states and their connection to each other using transitions. In order to show the applicability of the approach in an industrial setting, the State Machines were created in such a way that they contain mainly those structures which were often used in the industrial examples.

The state space limitation techniques used for the illustrative examples are then applied to the industrial use cases namely the Flash Light Controller (FLC) and the Park Assistant (PA). In these scenarios certain test goals like a high model coverage or a specific test case had to be achieved.

6.1. Illustrative Examples

In this section common State Machine structures, vastly increasing the number of potential test cases, are described and the impact of corresponding search limitation techniques are presented. In particular transition loops, history states and completion transitions are discussed. For this reason two examples are provided, where the first example focuses on the history states and transition loops and the second example on completion transitions.

The results were obtained using different structural search limitations. The used limitation techniques are those having the most impact on the generated test case number and restrict the structural search depth and the number of allowed loops in a symbolic path. In addition we show the influence of the distance between two test purpose steps and the impact of the Non-Target Exclusion (NTE) described in Section 3.5. The distance between two test purpose steps is the number of transitions on the shortest path between their included transitions.

6.1.1. Keyless Access Controller

The results presented in Table 6.1 are obtained for the test purpose *AckWarn*. It contains only one included transition of the KAC State Machine, which is described in detail in Example 4.4. The included transition has the label *extAckWarning* and *WarnLightOn* and *WarnLightOff* as source- and target vertex respectively.

The test goal described by the test purpose is to ensure that the warning light can be manually turned off after the key got out of range of the KLD while the vehicle is moving. The shortest path fulfilling the test purpose covers the transitions with the labels *extSpeed*, *evKeyInRange* and *extAckWarning* in the given order. For this path only the KAC State Machine is considered and the path is assumed to start from the initial state *CarLocked*.

During the test case generation a value for the *extSpeed* signal parameter *status* has to be calculated, where *status* has to be less than 20 in order to prevent the execution of the completion transition between the states *AutoUnlocked* and *AutoLocked*.

The main challenge for the test case generation is the infinite number of possible test cases, which is caused by loops in the State Machine. These loops represent correct behavior and therefore cannot be neglected, but do not necessarily contribute to a given test goal. For the test purpose described above mainly two loops in the KAC have an impact on the number of generated test cases. The first loop consists of the two transitions between the states *CarLocked* and *CarUnlocked* with the labels *evKeyInRange* and *evKeyOutOfRange*. The second loop consists of one transition between the state *CarMoving* and H_1 having the label *extSpeed*.

Constructs including a history state as for the second loop are often used in industrial applications, because they provide a simple way to update attributes without being dependent on a certain state. This is often necessary in order to keep track of the state of other State Machines or to temporarily store signal parameter values.

The loops shown above have a direct impact on the number of test cases found during the test case generation. The reason is that these loops can be infinitely often executed, but do not contribute to the achievement of the test goal. This means for example that the signals *evKeyInRange* and *evKeyOutOfRange* can be received arbitrarily often before the signal *extSpeed* is received, which leaves the state *CarStopped* and is needed in order to fulfill the test purpose. The same is true for the signal *extSpeed* after the state *CarMoving* has been entered.

For this reason it is necessary to limit the considered state space and correspondingly the number of found test cases. The results in Table 6.1 show the number of generated test cases for varying structural search depths d and the maximum number of loops. The structural search depth defines the maximum number of consecutive transitions and the maximum loop number restricts the overall number of loops allowed during the structural search.

The results in Table 6.1 show that the number of found test cases increases with an increasing search depth and a higher number of allowed loops. The restriction of the overall loop number during the structural search prevents the creation of more test cases and heavily reduces the number of the created symbolic paths. The number of generated test cases also depends on the ratio between the search depth and the loop number, because the search depth has a direct impact

No.	Configuration	0	1	2	4	9
1	$d = 9$	3	7	10	33	63
2	$d = 7$	3	7	10	28	51
3	$d = 5$	3	7	10	23	36
4	$d = 3$	1	1	1	1	1

Table 6.1.: Number of generated test cases for the test purpose *AckWarn*.

on the loops which can possibly be covered. The reason is that loops are formed by transitions which considered number is restricted by the search depth.

6.1.2. Completion Example

In this example the focus lies on the influence of completion transitions on the test case generation result. The discussion is based on an illustrative example consisting of two UML State Machines depicted in Figure 6.1 and 6.2. Both State Machines contain one integer attribute named x and p indicates the signal parameter of the signals b and f, which is also of type integer. The State Machines contain several loops like the self looping transitions with the labels a, f and g and the loop formed by the transition with the labels c and d and c, *after(100)* and d, respectively.

In this example two signals are used for communication between the State Machines, where e is sent from model A to B and b is sent vice versa. The model A contains one delay transition with source vertex *A1*, which requires a timing group in the corresponding ESTS. The state *A1* is the only element in the states and the transition with signal c is the only clock reset transition of this timing group.

These State Machine contain several deadlocks, where deadlock means that no transition can be executed anymore. This is for example the case if the transition with signal b in model A or the transition with signal e in model B is executed.

In this example the focus lies particularly on the influence of the completion transition on the number of generated test cases. For this reason the three completion transitions are used in *Model A* having either *A3* or *A4* as source vertex. The vertex finally reached, after the execution of the transition labeled with b, is defined by its signal parameter value, which enables the completion transitions to be executed afterwards. In this case the final state can either be *A3*, *A4* or *A5*.

In this example a single test goal in form of a test purpose is used, where the transition with label b, source vertex *A0* and target vertex *A3* is the only included transition. Since x has a value in the range $[1, 4]$ after the execution of this transition, always all four possible completion transition executions have to be considered during the test case generation.

The results in Table 6.2 and 6.3 were obtained for the test purpose described above using various configurations of the state space reduction methods described in Section 3.5. In these tables d is the maximum search depth, NTE indicates the usage of the non-target exclusion and E

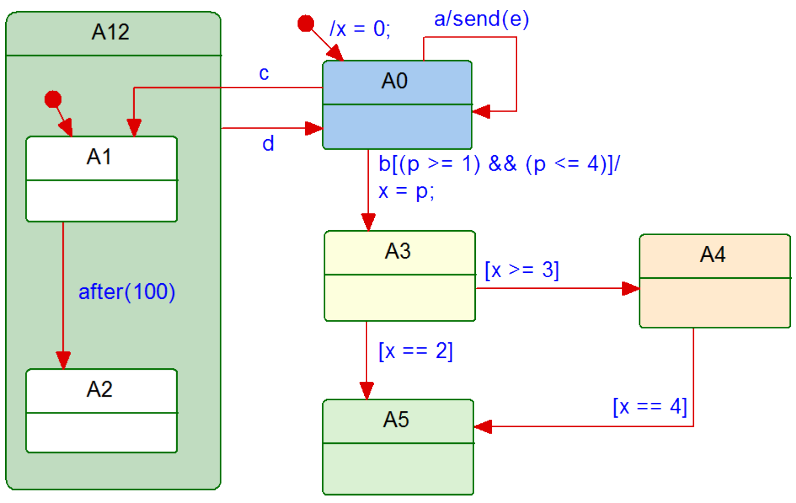


Figure 6.1.: Model A

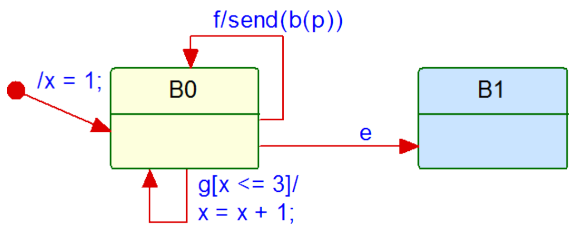


Figure 6.2.: Model B

No.	**Configuration**	**0**	**1**	**5**	**10**
1	$d = 10$	16	28	76	136
2	$d = 5$	16	28	52	60
3	$d = 3$	12	16	16	16
4	$d = 1$	4	4	4	4
5	$d = 10$, *NTE*	4	4	4	4
6	$d = 10, E = \{a\}$	12	20	52	80
7	$d = 10, E = \{c\}$	8	12	28	40
8	$d = 10, E = \{a, c\}$	4	4	4	4

Table 6.2.: Number of generated test cases for model *A*.

No.	**Configuration**	**0**	**1**	**5**	**10**
9	$d = 10$	12	24	140	220
10	$d = 1$	4	4	4	4
11	$d = 10$, *NTE*	4	4	4	4

Table 6.3.: Number of generated test cases for model *A* and *B*.

is the set of excluded transitions in the test purpose, where only their signal names are presented. Furthermore *No.* is the number of the experiment and l the maximum number of allowed loops during the structural search.

Table 6.2 contains the number of feasible paths obtained only form model *A*, whereas model *B* was not taken into account during this analysis. For this reason the data reflects only the paths without any communication dependency resolution and the according generation of sender paths. The results show that even for simple models such as *A* a high number of test cases can be obtained if weak state space limitations are applied. This situation is intensified with a growing system size as shown in Table 6.3, where in addition to model *A* also model *B* was considered.

The usage of different strategies to reduce the state space allows to find a minimum number of paths fulfilling the test goal. However, the usage of a particular strategy depends on the underlying State Machine and a general solution cannot be provided.

Morover the restriction of the structural search depth d is problematic, because it is also used in during the structural search in the sender models. Therefore a conflict arises between the search capabilities on the model containing transitions included in a test purpose and the sender models, where the search should be less restricted. The reason is that the included transitions of the test purpose steps can be set close to each other and therefore do not require an extended structural search, where in this context close means that only a small number of transitions lies between the included transitions of two test purpose steps.

However, it is possible to specify a depth separately for every sender model, but this can easily turn out to be a tedious task. A compromise is the Non-Target Exclusion (NTE), which allows a higher flexibility during the structural search for sender paths while maintaining a focused search for close test purpose steps. The same result can be achieved with transition exclusions, which can be used for fine grained interventions allowing a detailed tailoring of the test case generation process.

6.2. Industrial Use Cases

In this section the state space limitation techniques are applied on two industrial use cases, where a fast generation of test cases fulfilling a given test purpose was desired. The main difference between these use cases is the usage of data, where the Flash Light Controller (FLC) model contains only a few attributes and signal parameters and therefore describes a control flow for the most part. In contrast the Park Assistant (PA) uses several attributes and signal parameters, which increases the considered state space significantly.

6.2.1. Flashlight Controller

The goal of this industrial use case was to support test engineers during the test case development, where a Flash Light Controller was used to assess the approach. The core model of the FLC is shown in Figure 6.3 and supports four different flash modes. These modes contain the absence of flashing, the left and right turn indication flashing and the warning flashing, where both directions flashlights are active.

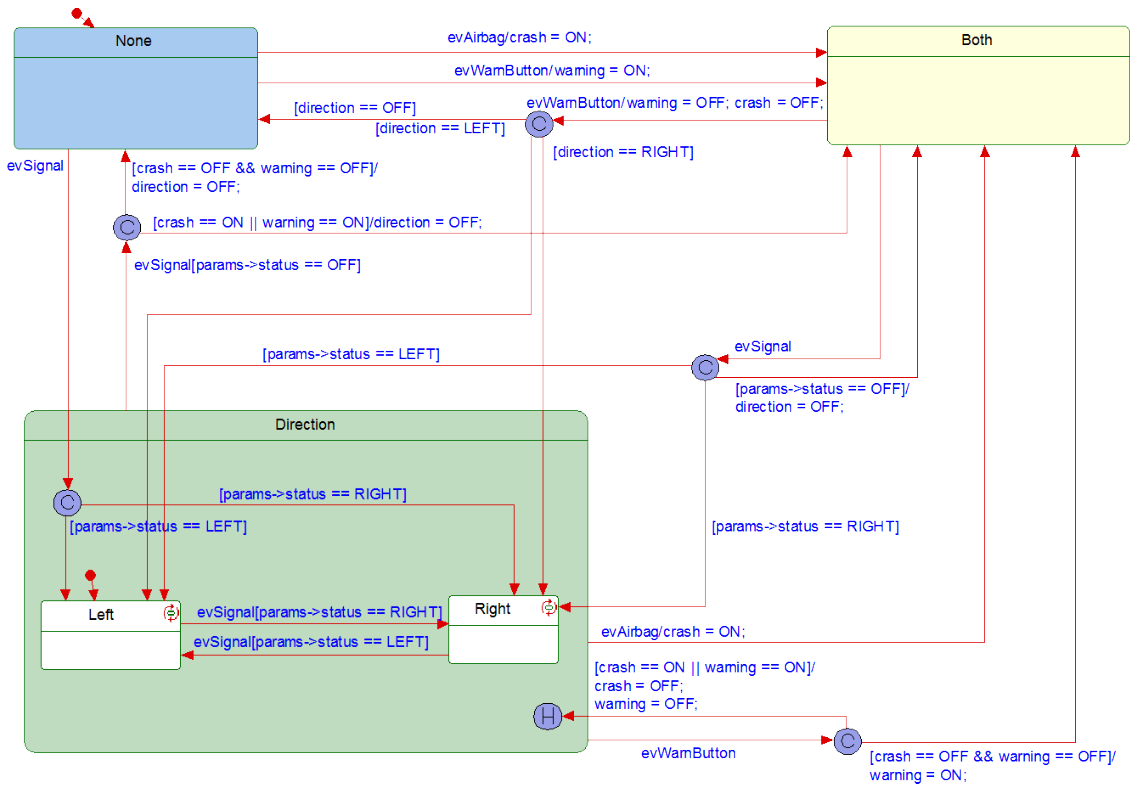
For the flashlight controller it was required that the original flash mode will be restored if it was changed while it was active. This means for example that if warning flashing is activated and a left turn indication flashing is requested by the driver then warning flashing has to become active again after the turn indication flashing has been disabled.

This behavior is modeled in the FLC using attributes storing the last activated flashing mode. These attributes are named *direction*, *crash* and *warning*, where *direction* can have the values *OFF*, *LEFT* and *RIGHT* and *crash* and *warning* can either be *ON* or *OFF*. This means that the direction flashing mode is stored in the attribute *direction* and an activated crash and warning mode are stored in the attributes *crash* and *warning*, respectively.

The warning flashing can be activated either by the reception of the signal *evWarnButton* or *evAirbag*, where the former signal is sent if the driver presses the hazard button in the cockpit and the latter is sent if a crash is detected and the Airbag becomes activated. These signals as well as all other signals shown in Figure 6.3 are sent from other State Machines and therefore they need to be included as sender paths during the test case generation.

In sum the system consists of 10 states, 37 transitions and 3 attributes. Due to the absence of attributes and signal parameters holding data the state space only amounts for 120 states neglecting the impact of time.

In this scenario four test purposes were used, which are defined in such a way that their corresponding generated test cases achieve a high state- and transition-coverage. In order to minimize the generation time the structural search depth was limited to $d = 5$, the NTE was activated and no loops were allowed.



None
Both
Direction
Left
Right
evAirbag/crash = ON;
evWarnButton/warning = ON;
[direction == OFF]
evWarnButton/warning = OFF; crash = OFF;
[direction == LEFT]
[direction == RIGHT]
evSignal
[crash == OFF && warning == OFF]/ direction = OFF;
[crash == ON || warning == ON]/direction = OFF;
evSignal[params->status == OFF]
[params->status == LEFT]
evSignal
[params->status == OFF]/ direction = OFF;
[params->status == RIGHT]
[params->status == LEFT]
[params->status == RIGHT]
evSignal[params->status == RIGHT]
evSignal[params->status == LEFT]
evAirbag/crash = ON;
[crash == ON || warning == ON]/ crash = OFF; warning = OFF;
evWarnButton
[crash == OFF && warning == OFF]/ warning = ON;

Figure 6.3.: Flashlight Controller

Test Purpose	TC#	C0	C1	Length	Gen. Time
AirbagDir	1	75.0%	39.1%	32	0.516s
DirWarn	1	100.0%	39.1%	32	0.391s
SwitchDirBoth	1	100.0%	43.5%	47	0.551s
WarnDir	1	75.0%	26.1%	23	0.210s

Table 6.4.: Coverage and generation time for the given Test Purposes.

The obtained generation results are presented in Table 6.4, where **TC#** is the number of generated test cases, **C0** is the state coverage, **C1** the transition coverage, **Length** the number of transitions of the test case and **Gen. time** the needed generation time. The results show that these four test purposes were sufficient for the set coverage goals, where the corresponding generated test cases together achieved a 100% state- and transition-coverage. For this reason it was sufficient in this scenario to generate one test case per test purpose. This allowed the usage of a strong limitation of the considered state space and reduced the needed test generation time.

6.2.2. Park Assistant

In this use case the application of the presented approach is shown for systems intensively using data, where test cases for the integration test of a vehicle Park Assistant (PA) were created. The functionality of the PA is distributed over multiple components, where the behavior of each component was described by an ESTS. In sum this system consists of 15 components containing 188 states and 365 transitions, which were used to describe its behavior. In addition it contains 17 integer attributes with a value range $[0, 255]$. This leads to about $8.36 * 10^{42}$ states, not considering time or the corresponding timing group clock values.

The PA is used to support the driver during the parking maneuver by the output of sounds, which depends on the distance to obstacles in reverse driving direction. This means for example that the sound is pulsed if an obstacle is within a given range, continuous if it is closer and absent otherwise. The volume and frequency of the tone can be adjusted by the driver. In addition the system supports a rear view camera and a graphical representation of the distances. The image of the rear view camera can be overlaid by driving guidelines, which show the foreseeable driving direction of the car corresponding to the steering wheel angle.

The behavior of the PA depends on the status of the ignition, a possibly hitched trailer, driving direction, vehicle speed and parking break. These informations are provided by separate components, which are also part of the system.

In order to generate test cases and to show the impact of various generation configurations three test purposes are used, namely *Off*, *On/Off* and *Tone*. The test purpose *Off* defines that a corresponding test case has to deactivate the PA, which means that in a test case the PA has to be activated first. The activation is made explicit in the test purpose *On/Off*, where an additional test purpose step is used. The last test purpose *Tone* represents a test case of our industrial partner used during the integration test. It consists of 15 test purpose steps, where the PA is activated, deactivated and reactivated again. After the second activation a driving maneuver is simulated causing a pulsed tone, its volume lowering, its reactivation and disabling due to increasing distance to an obstacle. This is followed by moving towards an obstacle again causing a continuous tone, which shall be muted by engaging the hand brake. During the muting the frequency of the PA tone is adjusted and the distance to an obstacle is increased causing a pulsed tone before the PA is deactivated by the driver.

Table 6.5 shows the generation result for the test purpose *Off*, where **Loop No.** is the number of loops allowed during the structural search, **TC#** is the number of generated test cases, **C0** is

Loop No.	**TC#**	**C0**	**C1**	**Length**	**Gen. Time**
0	10	51.0%	27.9%	58 ± 7	5s
1	152	52.6%	30.6%	66 ± 10	17s
2	810	53.7%	32.0%	72 ± 15	61s

Table 6.5.: Number of generated test cases for Test Purpose *Off*.

Loop No.	**TC#**	**C0**	**C1**	**Length**	**Gen. Time**
0	3	38.3%	19.7%	57 ± 11	3s
1	24	40.9%	21.9%	63 ± 10	6s
2	81	42.5%	23.2%	69 ± 12	12s

Table 6.6.: Number of generated test cases for Test Purpose *On/Off* with $d = 6$

the state coverage, **C1** is the transition coverage, **Length** the mean and standard deviation of the generated test case lengths and **Gen. Time** is the overall time needed for the generation.

The loop number defines how often a transition can be reused during the structural search, which means for example that for a loop number of 0 the transition can only occur once in a resulting path. In this example we used a structural search depth of 6, which means that the paths corresponding to one test purpose step contains at maximum 6 input- or delay-transitions. The shown length is defined by the number of transitions on its corresponding path.

The data in Table 6.5 shows that the loop number has a direct influence on the length and number of found test cases. The reason is that transitions forming a loop on a state are incorporated in various orders if loops are allowed. This effect becomes stronger the higher the allowed loop count during the structural search becomes, which leads to more and longer test cases. However, short test cases are still created if the allowed loop number rises, which can be seen by the standard deviation of the test case length which is ± 7 for zero loops and ± 15 for two loops.

The generation time corresponds the number of created test cases, because in this case the component does only depend on other components described by small ESTSs only consisting of up to four states. For this reason the resolution cannot cause many possible paths given the used state space limitations explained above. These state space limitations reduce the amount of possible paths already quite a lot, but even a small increase of allowed loops increases the number of found test cases heavily, which can be seen in Table 6.5 in the lines with loop number zero and two.

In the case of the Park Assistant such self loop transition can be caused by transitions used to keep track of the status of other components. This means for example the volume or frequency used for the warning tone or if a trailer has been hitched or unhitched. These interactions with the system are allowed and have to be processed, but are not necessarily desired for a given test goal. This means that too much freedom during the structural search test cases are created where the trailer is hitched and unhitched multiple times, which is not required to fulfill the test purposes described above.

In order to further reduce the state space and therefore limit the resulting test case number the

Loop No.	**TC#**	**C0**	**C1**	**Length**	**Gen. Time**
0	3	38.2%	19.7%	57 ± 11	3s
1	8	40.9%	21.6%	58 ± 9	4s
2	10	42.0%	22.4%	59 ± 8	5s

Table 6.7.: Number of generated test cases for Test Purpose *On/Off* with $d = 3$

Loop No.	**TC#**	**C0**	**C1**	**Length**	**Gen. Time**
0	1	43.6%	24.1%	121 ± 0	6s

Table 6.8.: Number of generated test cases for Test Purpose *Tone*.

parameters used for the structural search can be made more restrictive or the test definition can be refined. The lower bound for the restriction of the structural search parameters is defined by the distance between two test purpose steps, where distance means the minimum number of input- and delay-transitions needed in order to create a feasible path between them. Table 6.6 shows the results of the test purpose *On/Off*, where we use the same structural search parameters as before, meaning the structural search depth is limited to six.

In comparison to the results obtained for the test purpose *Off* the number of found test cases and the corresponding generation time could be reduced significantly for a loop number of two. The reason is that a smaller number of transitions is needed to reach a transition of a test purpose step, which prevents a breadth search and therefore limits the number of considered paths. This effect can also be observed by the obtained coverage, which is reduced correspondingly.

The results shown in Table 6.7 are achieved for the test purpose *On/Off*, where a structural search depth of three is used instead of six. The limitation of the search depth also reduces the impact of the allowed loop number, because only a few loops and their combinations are possible for this search depth. This generation configuration reduces again the considered state space and correspondingly the needed generation time and number of found test cases. The results presented in Table 6.5 to 6.7 show that the state space can be limited efficiently using the presented state space limitation techniques. This approach allows to reduce the amount of paths created for a test purpose step significantly and therefore reduces the overall generation time. This approach allows the fast generation of test cases for longer test purpose like *Tone*.

Table 6.8 shows the results obtained for a test case generation for the test purpose *Tone*, where the maximum distance between the transitions of two test purpose steps was at most two input- or delay-transitions. Correspondingly we used a maximum structural search depth of three, activated the non-target exclusion and continued only with the shortest found path after each test purpose step. This was sufficient in order to generate a corresponding test case within six seconds. This generation time is similar to the generation time for the test purpose *Off*, which is covered by the first two test purpose steps of *Tune*.

The results show that the presented approach allows to focus the test case generation on a small part of the state space in order to avoid the state space explosion and to reduce the needed generation time correspondingly. The focused search allows to neglect parts of the behavior described by the ESTSs, which does not contribute to the test goal defined in the test purpose. The behavior not contributing to the test goal can be calculated because of the usage of separated component models. This allows the presented algorithm to inserts paths of other components, which correspond to an input- or output-transition. Since the separation of the components is maintained, during the search only the possible behaviors of one component has to be considered at a time. In contrast the behavior contributed by each component cannot be distinguished anymore for a system model containing all possible behaviors in one ESTS. For this reason all possible behaviors have to be considered at a time limiting the scalability of such approaches.

Chapter 7

Final Remarks

Parts of this chapter are taken from ´An Extended Symbolic Framework for Systematic Test Case Generation´ which is an ongoing work with Bernhard K. Aichernig and Franz Wotawa.

7.1. Related Work

The automatic generation of test cases has been a research topic for decades now and has gained a lot of attention from the scientific community in recent years. The ever increasing complexity of software and software enabled systems requires sophisticated techniques to enhance the system quality. For this reason many scientific and industrial tools are available covering a broad range of verification methods.

The test generation tools TGV [18] and TORX [31] are both based on a LTS [32] and were applied successfully to various scientific and industrial applications as well. TGV allows the off-line generation of test cases with respect to a given test purpose, which is a systematic test definition and is used during the state space exploration to find feasible test cases. TORX implements an on-the-fly random test case generation algorithm. On-the-fly means that the generated inputs are executed on the SUT and its produced outputs are checked immediately. For this reason the state space is iteratively explored which allows this approach to be used on complex specifications, but due to the randomness it is hard to find tests according to a given test definition. The main drawback of an LTS is the treatment of data, which needs to be enumerated and therefore limits the scalability of such approaches.

Also model checkers like SPIN [2] have been used to generate test cases. This is done by checking the specification against a given proposition formulated in e.g. Linear Temporal Logic (LTL) or Computational Tree Logic (CTL). The proposition has to be formulated in such a way, that it is not fulfilled by the specification and a counterexample can be calculated. This counterexample describes a path through the model and therefore represents the test case.

Usually the specification has to be provided in a process algebra language like Promela or LOTOS [17], whose creation is often a laborious task and limits the usage of such approaches on a daily work basis.

In [33] a mutation testing approach for UML state transition diagrams is presented, where faults are injected into the UML models and then test cases are generated that would detect this faults. This fault-based approach is orthogonal to the presented test purpose driven test case generation, although one could view the injected faults as test purposes, as has been shown in [26]. The rational is to test whether a faulty model, i.e. a mutant, has been implemented. The mutation approach is implemented by first translating the UML models to action systems, and then, by running an **ioco**-check between the original and mutated labeled transition systems of the two models. That approach allows the usage of non-deterministic models.

In order to enhance the scalability of these techniques, which are limited by the explicit treatment of data and therefore by the state space explosion, a symbolic handling has been used in the following approaches. Symbolic means that the data is stored in variables, whose values have to be determined during the test case generation. Several scientific tools based on a STS like the STSIMULATOR[13] and STG [34] already exist. The STSIMULATOR provides a framework used for on-the-fly random testing comparable to TORX and is used in the Jambition Project [35] to automatically derive test cases for web applications. It is an implementation of **sioco**, which is the symbolic variant of **ioco** used to determine the correctness of the SUT.

STG is an extension of TGV and allows the generation of test cases with respect to a given test purpose. Approaches using symbolic execution as presented above rely on constraint solvers to find feasible variable values satisfying the conditions in the transition guards. The capabilities of such approaches are usually limited to the constraint solver capabilities, which are often restricted to integer variables and simple data-structures. The impact of constraint solver limitations on the scalability is also discussed in [36] in more detail. Another issue is the calculability of input values for a given result, which is not always possible like for the computation of hash values.

A method to circumvent the calculability issue is used in CONCOLIC execution [37], where a combination of CONCrete values and symbOLIC execution is used. In such approaches the input values are generated randomly at the initial stage and are refined by using a constraint solver. This allows the execution of arbitrary functions, but due to the lack of an invertibility this technique may miss given test scenarios with low probability.

PEX [38] is a tool based on dynamic symbolic execution, which extends the approach of pure symbolic execution by the incorporation of values calculated at runtime. The feasibility check during the symbolic execution is performed by a Satisfiability Modulo Theories (SMT) solver, which allows the handling of different domains like integer, bit-vectors, heaps and data-types.

SPEC EXPLORER [39] uses alternating simulation to define the conformance between the SUT and the model. It was recently extended to work with UML sequence diagrams used for testing and program slicing. In comparison to this work SPEC EXPLORER does not support model composition, which is one of the key features of the presented approach, and does no full symbolic state space exploration. However, this allows a wider range of supported data types in contrast to this work, where we are limited to integer and boolean values.

Although the approaches described above were used successfully in various applications, they do not incorporate time as part of the specification. For this reason several extensions were introduced to lift the well understood approaches to timed models. This led in the case of an LTS to its timed version and the according implementation relations like **tioco** and **rtioco**. A detailed discussion is given in [14], where a survey about the similarities and differences between these approaches and their variants is provided. However, these techniques still rely on an enumerative treatment of data limiting the scalability in data intense applications.

Also model checkers based on timed automata like UPPAAL [40, 41] were used for the generation of test cases. The timing constraints in a timed automata are given as time invariants on states and clock guards on transitions. UPPAAL also allows the interaction of data and time, meaning that attribute values can be used in the timing constraints. Their approach still relies on an explicit modeling of data and therefore faces the same scalability problems as methods based on an LTS.

A symbolic variant based on timed automata is defined in [42], where the Symbolic Timed Automaton (STA) is introduced. It is a combination of a STS with the timing handling of a timed automata and also allows the usage of attribute values as bounds for timing constraints. On the basis of the STA the testing conformance relation **stioco** being a symbolic extension of **tioco** is described. Although a STA allows similar semantics, it does not include a formal description of a composition and neglects unobservable events at the moment.

As mentioned before the main limitation in symbolic execution is the capability of the used constraint solver and in enumerative approaches the scalability. These issues are for example circumvented by the random input generation [43] approaches, where the SUT is executed randomly and the obtained paths are used to determine various coverage metrics. However, the achieved coverage is often poor, due to the random nature of this approach, which often creates the same paths having a high probability.

For this reason the approach was extended in search based testing [44], which uses a metaheuristic approach to guide the input data creation. This allows a better control of the input value selection and therefore a steering of the executed paths, but the detection of paths with low probability is still present. Low probability paths are often caused by conditions satisfied only by a small range of parameter values.

In addition to the scientific tools discussed above also commercial tools are already available, which use different approaches and test goals.

CONFORMIQ DESIGNER [45] from Conformiq allows the test generation based on QML, which is a subset of UML state machines. It supports the usage of multiple and nested state machines, and generates test cases by finding paths between defined start and end points.

CERTIFYIT [46] developed by Smartesting (former Leirios) is based on a model specified by a subset of UML to automatically derive test cases. The used test model consists of class-, state chart- and object-diagrams, which are used in combination with OCL to generate test cases.

TESTWEAVER [47] from QTronic is an on-the-fly testing tool, which allows the usage of reactive and continuous models for the test process. It supports a wide range of development tools for various purposes but cannot be used for offline test case generation.

7.2. Summary

In this work the challenges in testing and the automated generation of test cases and execution in an industrial setting has been elaborated. It has been shown that a model-based test generation approach can support a further automation of the test development, where in this work the Extended Symbolic Transition System (ESTS) was introduced to allow the formalization of a system component behavior. The ESTS extends the Symbolic Transition System defined by Frantzen et. al [13] by timed behavior and contains additional transition types allowing a straight forward model transformation of Unified Modeling Language State Machines.

In the context of this work a system consist of a set of communicating components, which are interconnected to each other by a network. The system behavior is defined by the composition of these components, where a global queue was used for signal passing between them. The usage of the global queue ensures a deterministic communication, which prevents a further increase of the state space.

On the basis of the communicating ESTSs a test case generation algorithm was introduced, which creates test cases corresponding to a given test purpose. The presented algorithm maintains the separation of the components, meaning that no single system model has to be created, which contains all possible behaviors of the composed components. For this reason the contribution of each component to the system behavior is kept and can be used to focus the test case generation on those parts of the state space which are necessary in order to create a feasible test case. This approach is a trade-off between scalability and generality, where a high scalability is necessary for industrial sized problems.

In general the test case generation algorithm creates symbolic paths for a test purpose step. These symbolic paths can contain dependencies of other components named senders and have to be resolved. The dependency resolution is done by the creation of symbolic paths in the sender components and their insertion in the original symbolic path. Since at this stage only states and transitions are considered, their feasibility has to be ensured by using a constraint solver. The constraint solver tries to calculate feasible input parameter values for a corresponding path constraint of a symbolic path, which contains all variable updates and conditions of the path transitions. If the constraint solver can find feasible input parameter values for the path constraint also the feasibility of the path is ensured.

Since an easy to use modeling language is required in an industrial process, the Unified Modeling Language and in particular State Machines were used to create component models. These component models describe the behavior of the components and were transformed into ESTSs. The transformation can be performed after the pseudo-states and state hierarchy has been removed, which is done during the State Machine flattening.

The presented approach was implemented in the prototype STATION, which was part of an integrated tool chain. This tool chain was used for the development and execution of test cases in the automotive industry. It consisted in addition to STATION of the UML modeling tool RHAPSODY and the test automation framework EXAM.

The implemented prototype STATION was used to validate the approach and to create experimental results, which show the applicability of the introduced state space limitation techniques for industrial sized settings.

7.3. Future Work

The presented approach allows the test case generation from deterministic ESTSs, which can be created from UML State Machines using a deterministic communication based on a global queue. However, the restriction to deterministic systems can cause false test cases if the underlaying system is executed in parallel. In addition in this approach not all capabilities of UML State Machines are utilized, which hinders an efficient model development and therefore the applicability of the presented approach. Moreover the used State Machines and ESTSs are assumed to be correct, which is crucial for a test case generation approach and for which the model developers do not have a sufficient tool support yet.

Therefore the following extensions to the presented approach should be considered:

- **Non-determinism**: The behavior of non-deterministic ESTSs as well as the impact of a non-deterministic communication of their composition has to be elaborated.
- **Modeling elements**: The ESTS introduced in this work does not support any parallelism within an ESTS itself, like it is provided by parallel regions in UML State Machines. The parallel regions can influence the behavior of each other by changing the attribute values, which means that they are used as shared variables. This kind of interaction is also used by MATLAB within the Simulink/Stateflow toolbox, which is a widespread application in the automotive industry. For this reason a further investigation of parallel regions and shared variables incorporating various scheduling possibilities should be undertaken.
- **Model analysis**: Although first results for the automated analysis of ESTSs and therefore for their corresponding State Machines could be achieved [23], a further investigation of these techniques is necessary. The reason is that the correctness of the models is crucial for a model based test case generation approach, but would require an exhaustive test of these models. For this reason an automated approach is necessary which supports the model developer by static and dynamic analysis methods.

The extension of the presented approach according the stated list above could strengthen model-based development and testing approaches and could foster their widespread application in the industry.

7.4. Conclusion

Model-based test case generation techniques have been studied for decades and have been gaining increasing popularity in the industry in recent years. The reason is the availability of high computational power at low cost and steadily advances in constraint solving allowing the usage of symbolic test generation strategies.

In this work a symbolic test case generation algorithm as well as its prototypical implementation has been introduced. The algorithm relies on a purely deterministic system, which means that its components as well as their communication scheme has to be deterministic. This limitation is a necessary trade-off in order to achieve a high scalability, which is needed for industrial sized systems.

Although the state space can be vastly reduced by the restriction to deterministic systems it is still too large for test generation approaches based on a composite model containing all possible system behaviors. The reason is that due to the model composition the information about the contribution of each component is lost, which increases the effort needed during the test case generation.

In order to overcome the stated challenges above, the presented approach maintains the separation of the components and therefore allows a fine grained focusing of the test case generation. The focusing is achieved by multiple state space limitation techniques introduced in this work and allows to reduce the computational effort during the generation significantly as shown in the experimental results in Chapter 6.

The integration of the prototype STATION into an existing industrial tool chain allows the automated generation and execution of test cases. The applicability of the integrated tool chain was verified in an industrial setting, where it was used for integration testing of several ECUs using a Hardware in the Loop system.

List of Acronyms

ABS	Anti Blocking System
AC	Air Conditioner
CAN	Controller Area Network
CTL	Computational Tree Logic
DVB	Digital Video Broadcast
DVD	Digital Versatile Disc
ECU	Electronic Control Unit
EPB	Electronic Parking Brake
ESP	Electronic Stability Program
EFSM	Extended Finite State Machine
ESTS	Extended Symbolic Transition System
EXAM	EXtended Automation Method
FLC	Flash Light Controller
GW	Gateway
HiL	Hardware in the Loop
HMI	Human Machine Interface
HU	Head Unit
KAC	Keyless Access Controller
KLD	Key Location Detector
PC	Power Controller
LIN	Local Interconnect Network
LOTOS	Language of Temporal Ordering Specification
LTL	Linear Temporal Logic
LTS	Labeled Transition System
MBT	Model-based Testing
MOST	Media Oriented Systems Transport
NSM	Nested State Machine
NTE	Non-Target Exclusion
OMD	Object Model Diagram
OMG	Object Management Group
PA	Park Assistant

RVC	Rear View Camera
SA	Shock Absorber
SAS	Steering Angle Sensor
SMT	Satisfiability Modulo Theories
STA	Symbolic Timed Automaton
STS	Symbolic Transition System
STATION	STAte based system Test and simulatION
SUT	System Under Test
TA	Test Automation
TLTS	Timed Labeled Transition System
UML	Unified Modeling Language
SM	State Machine
XMI	Extensible Markup Language Metadata Exchange
XML	Extensible Markup Language

Bibliography

[1] E. M. Clarke and B.-H. Schlingloff, *Model checking*. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 2001.

[2] D. Latella, I. Majzik, and M. Massink, "Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker," *Formal Aspects of Computing*, vol. 11, no. 6, pp. 637–664, 1999.

[3] S. Siegl, K.-S. Hielscher, R. German, and C. Berger, "Formal specification and systematic model-driven testing of embedded automotive systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, march 2011, pp. 1 –6.

[4] S. Siegl, K.-S. Hielscher, and R. German, "Introduction of time dependencies in usage model based testing of complex systems," in *Systems Conference, 2010 4th Annual IEEE*, april 2010, pp. 622 –627.

[5] "Can in automation." [Online]. Available: http://www.can-cia.org

[6] "Lin consortium." [Online]. Available: http://www.lin-subbus.org

[7] "Flexray consortium." [Online]. Available: http://www.flexray.com/

[8] "Most cooperation most." [Online]. Available: http://www.mostcooperation.com

[9] OMG, "UML superstructure reference, http://www.omg.org/spec/uml/2.1.2/superstructure/pdf/," (last visited Jan. 09).

[10] "Ibm rational rhapsody." [Online]. Available: http://www.ibm.com/software/awdtools/rhapsody/

[11] "SPARX systems enterprise architect." [Online]. Available: http://www.sparxsystems.com.au/

[12] "Visual paradigm international visual paradigm." [Online]. Available: http://www.visual-paradigm.com/

[13] L. Frantzen, J. Tretmans, and T. A. C. Willemse, "Test generation based on symbolic specifications," in *FATES 2004, number 3395 in LNCS*. Springer-Verlag, 2005, pp. 1–15.

[14] J. Schmaltz and J. Tretmans, "On conformance testing for timed systems," in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, F. Cassez and C. Jard, Eds. Springer Berlin / Heidelberg, 2008, vol. 5215, pp. 250–264. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85778-5_18

[15] V. Chimisliu, C. Schwarzl, and B. Peischl, "From UML statecharts to LOTOS: A semantics preserving model transformation," in *Proceedings of the 2009 Ninth*

International Conference on Quality Software, ser. QSIC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 173–178. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2009.31

[16] ——, "Test case generation for embedded automotive systems: A semantics preserving model transformation," in *Proceedings of the 2009 Second Workshop on Model-based Testing in Practice*, ser. MOTIP '09. Entschede, The Netherlands: IEEE Computer Society, 2009, pp. 43–52. [Online]. Available: http://www.fokus.fraunhofer.de/en/fokus_events/motion/motip_2009/_docs/Motip2009_proceeding_final.pdf

[17] ISO, "ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour," 1989.

[18] C. Jard and T. Jéron, "TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 4, pp. 297–315, 2005.

[19] C. Schwarzl and B. Peischl, "Generation of executable test cases based on behavioral UML system models," in *Proceedings of the 5th Workshop on Automation of Software Test*, ser. AST '10. New York, NY, USA: ACM, 2010, pp. 31–34. [Online]. Available: http://doi.acm.org/10.1145/1808266.1808271

[20] ——, "Test sequence generation from communicating UML state charts: An industrial application of symbolic transition systems," in *Proceedings of the 2010 10th International Conference on Quality Software*, ser. QSIC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 122–131. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2010.22

[21] C. Schwarzl, B. Aichernig, and F. Wotawa, "Compositional random testing using extended symbolic transition systems," in *Testing Software and Systems*, ser. Lecture Notes in Computer Science, B. Wolff and F. Zaidi, Eds. Springer Berlin / Heidelberg, 2011, vol. 7019, pp. 179–194. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24580-0_13

[22] C. Schwarzl and F. Wotawa, "Test case generation in practice for communicating embedded systems," *e & i Elektrotechnik und Informationstechnik*, vol. 128, pp. 240–244, 2011, 10.1007/s00502-011-0009-5. [Online]. Available: http://dx.doi.org/10.1007/s00502-011-0009-5

[23] C. Schwarzl and B. Peischl, "Static- and dynamic consistency analysis of uml state chart models," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, D. Petriu, N. Rouquette, and y. Haugen, Eds. Springer Berlin / Heidelberg, 2010, vol. 6394, pp. 151–165, 10.1007/978-3-642-16145-2_11. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16145-2_11

[24] A. Petrenko, S. Boroday, and R. Groz, "Confirming configurations in efsm testing," *Software Engineering, IEEE Transactions on*, vol. 30, no. 1, pp. 29 – 42, jan. 2004.

[25] H. van der Bijl, A. Rensink, and G. Tretmans, "Compositional testing with ioco," in *Formal Approaches to Software Testing (FATES)*, ser. Lecture Notes in Computer Science, A. Petrenko and A. Ulrich, Eds., vol. 2931. Berlin: Springer Verlag, 2004, pp. 86–100. [Online]. Available: http://doc.utwente.nl/66359/

[26] B. K. Aichernig and C. C. Delgado, "From faults via test purposes to test cases: on the fault-based testing of concurrent systems," in *Proceedings of FASE'06, Fundamental*

Approaches to Software Engineering, Vienna, Austria, March 27–29, 2006, ser. Lecture Notes in Computer Science, L. Baresi and R. Heckel, Eds., vol. 3922. Springer-Verlag, 2006, pp. 324–338.

[27] Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M.-L. Potet, "Test purposes: adapting the notion of specification to testing," in *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, nov. 2001, pp. 127 – 134.

[28] T. Jéron, "Symbolic model-based test selection," *Electron. Notes Theor. Comput. Sci.*, vol. 240, pp. 167–184, 2009.

[29] David and Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231 – 274, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167642387900359

[30] C. Schwarzl and W. Stefan, "Test case generation for integration testing of electronic control unit networks," in *Simulation und Test für die Automobilelektronik*. Expert Verlag, 2012, pp. 280–291.

[31] J. Tretmans, E. Brinksma, and C. D. Resyste, "TorX: Automated model based testing - côte de Resyste," 2003.

[32] J. Tretmans, "Test generation with inputs, outputs, and quiescence," in *Lecture Notes in Computer Science*. Springer, 1996, pp. 127–146.

[33] B. K. Aichernig, H. Brandl, E. Jöbstl, and W. Krenn, "Efficient mutation killers in action," in *IEEE Fourth International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, 21-25 March 2011*. IEEE Computer Society, 2011, pp. 120–129.

[34] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva, "STG: A symbolic test generation tool," in *Lecture Notes in Computer Science*. Springer, 2002, pp. 151–173.

[35] L. Frantzen, M. Las Nieves Huerta, Z. G. Kiss, and T. Wallet, "On-the-fly model-based testing of web services with Jambition," in *Web Services and Formal Methods*, R. Bruni and K. Wolf, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 143–157. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01364-5_9

[36] E. J"obstl, M. Weiglhofer, B. Aichernig, and F. Wotawa, "When bdds fail: Conformance testing with symbolic execution and smt solving," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, april 2010, pp. 479 –488.

[37] K. Sen, "Concolic testing," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007, pp. 571–572.

[38] D. Vanoverberghe, N. Bjørner, J. Halleux, W. Schulte, and N. Tillmann, "Using dynamic symbolic execution to improve deductive verification," in *Proceedings of the 15th international workshop on Model Checking Software*, ser. SPIN '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 9–25. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85114-1_4

[39] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with Spec Explorer," in *Formal Methods and Testing*, ser. Lecture Notes in Computer Science, R. Hierons, J. Bowen, and M. Harman, Eds. Springer Berlin / Heidelberg, 2008, vol. 4949, pp. 39–76, 10.1007/978-3-540-78917-8_2. [Online]. Available:

http://dx.doi.org/10.1007/978-3-540-78917-8_2

[40] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using UPPAAL," in *Formal methods and testing*, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 77–117. [Online]. Available: http://portal.acm.org/citation.cfm?id=1806209.1806212

[41] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, ser. LNCS, M. Bernardo and F. Corradini, Eds., no. 3185. Springer, September 2004, pp. 200–236.

[42] S. Von Styp, H. Bohnenkamp, and J. Schmaltz, "A conformance testing relation for symbolic timed automata," in *Proceedings of the 8th international conference on Formal modeling and analysis of timed systems*, ser. FORMATS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 243–255. [Online]. Available: http://portal.acm.org/citation.cfm?id=1885174.1885193

[43] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, vol. 40, no. 6. New York, NY, USA: ACM Press, June 2005, pp. 213–223. [Online]. Available: http://dx.doi.org/10.1145/1065010.1065036

[44] P. McMinn, "Search-based software test data generation: a survey: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, pp. 105–156, June 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1077276.1077279

[45] "Conformiq designer." [Online]. Available: http://www.conformiq.com

[46] "Smartesting certifyit." [Online]. Available: http://www.smartesting.com

[47] "Qtronic testweaver." [Online]. Available: http://www.qtronic.de