Thomas Gödl, BSc

Static Analysis of Extended Symbolic Transition Systems

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Institute for Software Technology

Dipl.-Ing. Dr.techn. Christian Schwarzl
Virtual Vehicle Research Center

Graz, April 2015

This document is set in Palatino, compiled with pdfLaTeX2e and BibTeX.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

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.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

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.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Models are widely used in the field of software engineering at the present day to model cyber-physical systems, embedded systems and software systems. They provide different abstract views and support a common understanding among various stakeholders. In Model-based Testing a system is verified in terms of its correctness between a behavioral model, which describes the behavior of the system, and the System Under Test. Since models are used in the Model-based Testing approach to ensure the correctness of the System Under Test, these models are required to be correct.

In this thesis a static analysis technique approach is proposed which is applied to an Extended Symbolic Transitions System, which is a concrete model utilized in Model-based Testing, in order to ensure and measure the quality of such a model. The static analysis addresses the quality of an Extended Symbolic Transition System in terms of two aspects: (1) validating the model structure for contradictions by means of *checks* and (2) measuring the model quality quantitatively by means of *metrics*. Thus, this thesis introduces a collection of checks and metrics which are applicable to an Extended Symbolic Transition System. The checks and metrics are implemented in a prototype static analysis tool and are applied to two illustrative examples. The results obtained can be used to infer the quality of an Extended Symbolic Transition System.

Kurzfassung

Die Nutzung von Modellen zur Modellierung Cyber-physischer Systeme, eingebetteter Systeme und Software Systeme ist heutzutage weit verbreitet. Sie bieten unterschiedliche abstrakte Sichten und fördern das Verständis zwischen verschiedenen Interessengruppen. Im modellbasierten Testen werden Systeme verifiziert, indem die Korrektheit zwischen einem Verhaltensmodell, welches das Verhalten des Systems abbildet, und dem zu testenden System sichergestellt wird. Da im modellbasierten Testen Modelle verwendet werden, um die Korrektheit des zu testendenden System zu gewährleisten, ist es erforderlich, dass die Modelle selbst korrekt sind.

In dieser Arbeit wird ein statischer Analyseansatz vorgestellt, der auf ein Extended Symbolic Transition System, welches ein konkretes Modell für das modellbasierte Testen darstellt, angewendet wird, um die Qualität eines solchen Modells zu garantieren und zu messen. Die statische Analyse befasst sich dabei mit der Qualität von Extended Symbolic Transition Systemen unter folgenden Aspekten: (1) Validierung der Modellstruktur, um Widersprüche unter Anwendung von sogenannten *Checks* zu eruieren, und (2) die quantitative Messung der Modellqualität mit Hilfe von Metriken. Die Arbeit stellt eine Sammlung von *Checks* und *Metriken* dar, die auf ein Extended Symbolic Transition System ihre Anwendung finden. Diese *Checks* und *Metriken* wurden in einem Prototyp eines statischen Analyseprogramms realisiert, welches auf zwei Beispielmodellen angewandt wurde. Die Ergebnisse zeigen, dass Schlussfolgerungen über die Modellqualität getroffen werden können.

Acknowledgment

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no 295311 (VETESS), from the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economy, Family and Youth (BMWFJ) and the Austrian Research Promotion Agency (FFG).

I would like to express my gratitude to my supervisors Franz Wotawa and Christian Schwarzl for the useful comments, remarks and engagements through the entire duration of this master thesis. I am extremely thankful and indebted to them for sharing expertise and their sincere and valuable guidance. I also thank my parents for the unceasing encouragement, support and attention. Furthermore, I am also grateful to my girlfriend who supported me throughout this venture.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

Models, such as Unified Modeling Language (UML)[1], Business Process Model (BPM)[2] and Finite State Machine (FSM)[3] models, are widely used in the field of software engineering at the present day to model cyber-physical systems, embedded systems and software systems. The models aid the engineering process by providing different abstract views of the system and therefore supports a common understanding between various stakeholders. Thus, software engineering disciplines like Model Driven Engineering (MDE)[4], Model Driven Architecture (MDA)[5] and Model-based Testing (MBT)[6, 7] rely on such models, which are used to express structural and behavioral aspects of a system.

The quality of software systems is an important issue. Especially in safety critical domains, like the air traffic, air safety, aeronautics, rail, and health care domain, it is crucial for these systems to meet highest quality standards. As software systems are becoming more and more complex, automated techniques are required to aid the software engineering process in order to ensure and measure the quality of the system under development. The *ISO/IEC 25010:2011(E)*[8] standard defines *software quality* in terms of a quality model consisting of eight main quality characteristics, namely *Functionality Suitability*, *Reliability*, *Operability*, *Performance Efficiency*, *Security*, *Compatibility*, *Maintainability* and *Transferability*, and a set of sub-characteristics. These quality characteristics can be automatically measured or assessed by qualified persons, in order to aid the development process in the detection of bad coding style, security and performance issues, in finding and fixing errors or in indicating issues in terms of usability and understandability[9]. The prior mentioned quality properties were mainly established with a focus on the source code of the software. On that account Lindeland *et al.* proposed a quality model in [10], which is more suitable for models and consists of three main quality characteristics, namely *Syntactic Quality*, *Semantic Quality* and *Pragmatic Quality*. Figure 1.1 illustrates the proposed quality characteristics and sub-characteristics with respect to models as discussed in [9].

Models are utilized nowadays to generate source code, to generate test cases in order to verify a system or even for simulation purposes. Thus, the quality of models has to be ensured and measured at an early stage in order to guarantee the quality for its target application. As a consequence, approaches like *static and dynamic analysis* are required. Static analysis is a widely used approach in terms of quality assurance of source code. The advantage lies in the fact that the source code must not be executed, in contrast to dynamic analysis. Thus, static analysis is carried out before compile time and can detect flaws in source code, like null pointers. Examples of tools which automatize the static analysis of source code are *FindBugs*[11], *Checkstyle*[12] or *PMD*[13]. Another technique of static analysis is model checking[14], which computes all possible states and executions

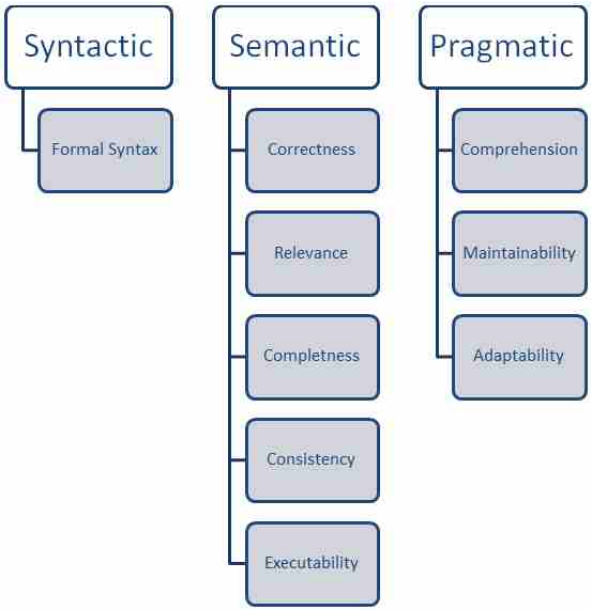


Figure 1.1.: Model quality characteristics and sub-characteristics by Lindeland *et al.* [9].

paths of the program without actually running the program[15]. The static analysis of models has already been addressed in scientific works like [16], [17], [18], [19], [20] or [21]. On the other hand computer-aided software engineering (CASE) tools, like *IBM Rational Rose*[22], already provide built-in basic checking in order to verify the syntactic correctness based on constraints.

Software metrics are used to measure quality properties of a software system in a quantitative manner and are mainly related to measure the complexity of software. These metrics act as indicators to describe the extent to which a system is understandable, maintainable or testable. With respect to the proposed quality model for models in Figure 1.1, the complexity metrics are related to *Pragmatic Quality*. Thus, complexity metrics in terms of models measure the degree to which a model can be comprehended, maintained and adapted.

1.1. Motivation

In Model-based Testing (MBT)[6, 7] a system is verified in terms of its correctness between a behavioral model, which describes the intended behavior of the system, and the System under Test (SUT). The range of models that can be used in MBT is wide, like UML State Charts, FSM, BPM models, Labeled Transition System (LTS)[23], Symbolic Transition System (STS)[24] or Action Systems[25, 26].

In [27] another MBT approach is discussed which employs an Extended Symbolic Transition System (ESTS)[28, 29, 27] as the underlying model. An ESTS model is based on an STS and enhances it by introducing the following additional properties: *(1) delay transitions*, *(2) completion transitions*,

(3) transition priorities, *(4) transition execution duration* and *(5) timing groups*. As models in the MBT context are used to prove the correctness of the SUT, it is desirable that these models are correct.

1.2. Contribution

The aim of this thesis is to provide a static analysis suite for ESTS[28, 29, 27] models in order to ensure and measure the quality of such models. This thesis is based on a proposal by Schwarzl[27], who suggested the application of static analysis techniques to an ESTS model. In contrast to Schwarzl's and Peischl's approach in [16], where an ESTS model is utilized as an intermediate model in order to apply static and dynamic analysis to UML State Charts, this thesis only focus on an ESTS model. The static analysis suite consists of two main components, namely *checks* and *metrics*.

Checks address the quality aspect of ESTS models by ensuring syntactically and semantically valid models and by extracting particular structural constructs from an ESTS. The following syntactic and semantic checks are introduced:

1. Communication resolution in terms of not receivable or not sent messages.
2. Non-determinism check in terms of overlapping transition guards resulting in multiple enabled outgoing transitions.
3. A check with respect to the detection of ambiguously defined attributes and parameters.
4. Ensuring valid guard and action expressions.
5. Detection of hidden transitions, which are never executable transitions since their guard parameter range are entirely covered by higher prioritized transitions.

Beyond that, this thesis proposes checks to extract particular constructs from an ESTS model, namely loops and cascades. The common property of both constructs is that they solely consist of transitions of type unobservable, completion and output. In terms of MBT such constructs can cause a huge test generation effort for approaches like the one stated in [27].

Metrics are introduced in this thesis in order to determine the complexity of an ESTS model. Complexity is addressed in terms of the structure, data flow, control flow and the size of a model. Some of the metrics have already been introduced for other models, like UML models. Therefore, these metrics have been adapted in order to meet the ESTS model structure. On the one hand, the metrics are a measure of the complexity of a model and thus address the quality properties of *Comprehension*, *Maintainability* and *Adaptability* with respect to Figure 1.1. On the other hand, these metrics can be utilized to draw a conclusion on the test generation effort, since they can be interpreted in terms of attribute, transition, state and transition-pair coverage.

The checks and metrics presented in this thesis find their application in a prototype implementation of a tool called *STSStaticAnalyzer*, which automates the static analysis process. The checks and

metrics are applied to two illustrative examples by means of the prototype tool. The results obtained show the outcome of the checks and metrics application.

1.3. Outline

The remainder of this thesis is organized as follows: In Chapter 2 the related work is discussed. In Chapter 3 an ESTS is defined in terms of its structure and semantics. Furthermore, some additional thesis specific definitions are introduced.

In Chapter 4 the analysis of an ESTS model is discussed. This chapter contains the description of three types of checks: (1) checks in terms of syntactic and semantic violation detection, (2) checks in order to extract certain structural constructs and (3) checks to calculate model metrics. Furthermore, some common data structures as well as convenient tools are introduced which are utilized throughout the chapter.

In Chapter 5 the experimental results are presented. The results are obtained by applying a prototype analyzer tool, which contains implementations of the checks defined in Chapter 4, to three ESTS models. The thesis concludes with Chapter 6, where a summary is provided and potential future work is discussed.

2. Related Work

Model analysis in terms of quality, to ensure *consistency* and *correctness* as well as *measuring complexity*, is a popular research topic as models are an important artifact in the fields of Object-oriented design, MBT[6, 7], MDE[4], MDA[5] and BPM[2]. This chapter gives an overview of existing scientific work concerning model quality.

2.1. Analysis of Models

UML[1] is the most utilized modeling language in the software engineering field. A fact that is reflected by the huge amount of papers addressing models created in UML as shown in [30], where an overview of 907 papers are given dealing with UML consistency management.

The work of Schwarzl and Peischl in [16] depicts the closest related work to this thesis. In their work they proposed static and dynamic analysis on UML state charts. The static analysis part covered the detection of *syntactical errors*, *undefined variables*, *not receivable messages*, *non-determinism in terms of overlapping parameter ranges* and *hidden transitions*. Whereas the dynamic analysis dealt with the detection of *deadlocks*, *infeasible transitions* and *inter-model loops*. The dynamic analysis applied the random walk method which traverses the model and resolve the model communication dependencies in order to create valid input sequences. Their approach transformed an UML state chart model to an STS model due to the ambiguous semantic of the former. The static and dynamic analysis techniques are applied to the STS model. In addition for dynamic analysis errors a failure trace through the model is created in order to assist the debugging task.

In [17] the tool vUML is introduced for the verification of UML class, collaboration and state charts utilizing the model checking[14, 31] method. UML models are transformed into PROMELA language in order to verify them by means of the model checker SPIN[32]. In this way the models are checked for the existence of *deadlocks*, *livelocks*, *reaching an invalid marked state*, *a violation of an object constraint for instance*. The tool provides for a detected error a counterexample in the form of a sequence diagram. A drawback of the tool is that additional stereotypes for states have to be introduced in order to assist the model checking routine. The model checker approach is often used in the context of UML model verification as shown in [30]. Grumberg *et al.* verified in [33] UML state charts in terms of *livelocks* and *Linear Temporal Logic (LTL) safety properties* by combining static analysis and *bounded model checking*. They also identified a subclass of livelocks, namely *cycle-livelocks*. Their approach transformed a state chart model into C code. The verification is carried out by means of the model checker CBMC[34]. Fernandes *et al.* [35] proposed an approach to verify UML

use case diagrams, state chart diagrams and activity diagrams by means of the a tool called UML Checker. The tool expects as input the respective model and validation properties which are defined in temporal logic notation. The model is transformed to the NuSMV[36] input language[37] in order to be validated by the NuSMV model checker[36] by considering the given validation properties.

In [18] the consistency between UML sequence and state chart diagrams are considered. This approach transformed both diagrams to Communicating Sequential Processes (CSP)[38] process algebra. The used consistency rules are defined by means of CSP assertions. The transformed diagrams are validated by means of the model checker FDR[39] with respect to the defined consistency rules.

Malgoures and Motet[40] formalized UML models as Constraint Logic Programming (CLP)[41]. They defined the consistency rules in CLP as well. The consistency check is carried out by means of a *Constraint solver*[42] based on the formalized models and the consistency rules. The main limitation of their approach lies in the fact that they are limited to a specific set of UML features which arise from the circumstance that they used CLP as formal language.

The authors of [43] chose as the intermediate formal language *Web Ontology Language OWL2*[44]. UML models consisting of multiple class, object and state chart diagrams are transformed into the OWL2 specification which is analyzed with a logic reasoner like HermiT[45]. This approach is able to validate modeling concepts like classes, object, associations, links, labeled links, domain, range, multiplicity, composition, unique and non-unique associations, ordering, class generalization and association generalization. In additions to that the conformance of object diagrams against class diagrams, consistency of class diagrams and state chart diagrams and the consistency of multiple merged models can be analyzed by their approach.

Approaches like [19] and [20] used Object Constraint Language (OCL) to define consistency rules directly on the meta-data level. Thus, no intermediate language and model, respectively, are required. In [20] the authors demonstrated the application of OCL in terms of validating the static semantics of UML models. In [19] the concepts of Queries, View and Checks are introduced based on OCL. Queries are functions over meta-model elements returning a value. A check is basically a query but only with a boolean return value and is utilized for assessing a model with respect to constraints. Views support the modeler with collected information about the models.

Egyed proposed a tool called UML Analyzer in [46, 47] which has the ability to perform *instant consistency checking*. As stated in the paper the instant approach is an adaptation of the incremental approach. The tool is built on top of IBM Rational Rose in order to access Rational Rose models as well as receive notifications when a model has changed. He showed an approach for quickly, correctly and automatically deciding when to evaluate consistency rules. Together with Reder, Egyed addressed the problem of resolving inconsistencies in design models in[48]. They stated that resolving inconsistencies is a much harder work than to find one. Their approach utilized a *Repair Tree* which organizes repair actions in a hierarchical manner. The *Repair Tree* basically reflects the structure of the consistency rules. A repair action is considered as a potential change to the model.

In order to model BPM, languages like Business Process Model and Notation (BPMN)[2], UML activity diagrams[1], YAWL[49] or Event-driven Process Chain (EPC)[50] are utilized. Gruhn and Laue showed in [51] the application of the logic programming paradigm to validate BPM in terms of *syntactical requirements*. Furthermore they showed that logic programming can be used for reasoning about complex properties like finding patterns in a model or checking consistency across models.

In [52] an approach is presented which used the model checking method to verify BPMN models with respect to *deadlocks*, *livelocks* and *multiple termination issues*. First, the approach translates a BPMN model to Kripke structure[53], a structure that is closely related to a FSM and which shows the characteristics that it does not differentiate between inputs, outputs, program locations and local variables[54]. Second, the desired properties, such as *deadlocks*, *livelocks* and *multiple termination* are expressed in LTL formulae. The LTL properties and the Kripke structure are the input for a model checker.

Awad and Puhlmann proposed another approach in [21] to detect *deadlocks* in BPMN models. They utilized BPMN-Q, a BPMN based graphical query language, by translating existing deadlock patterns, which have been identified in [55], into BPMN-Q. Dijkman *et al.* proposed in [56, 57] a translation of BPMN models into Petri nets[58] in order to utilize Petri net-based verification tools to check issues like *dead tasks* and *improper completions*.

2.2. Complexity of Models

Another quality aspect of models is *understandability* and therefore the *complexity* of models which is measured by model metrics. Genero *et al.* proposed in [59, 60] a metric suite for UML state charts. The suite consists of the two metric types *size metrics* and *structural complexity metrics*. The former metric type consists of the following metrics: *Number of Activities*, *Number of Simple States*, *Number of Events*, *Number of Guards*, *Number of Entry Actions*, *Number of Exit Actions* and *Number of Composite States*. The latter metric type consists of the metrics *McCabe Cyclomatic Complexity*[61] and *Number of Transitions*. Moreover, the authors empirically validated these metrics in order to prove them as good understandability indicators.

Lankford applied in [62] the *McCabe Cyclomatic Complexity* and *Halstead Difficulty* metric to UML class, sequence and state chart diagrams. Soliman *et al.* applied in [63] the *Chidamber and Kemerer metrics suite* to UML class, activity and sequence diagrams. In [64] Hall introduced two new metrics, namely *Top-Level Cyclomatic Complexity* and *Hierarchical Cyclomatic Complexity*, by adapting the McCabe Cyclomatic Complexity for non-hierarchical to hierarchical state charts. Hoe Bae *et al.* proposed in [65] the State Machine Understandability Metric (SUM) to measure the understandability of state machines. The metric is based on state cohesion and state coupling within a state machine.

In [66] the *Number of Deactivating Transitions*, *Number of Activating Transitions*, *Number of Deactivated States and Number of Activated States* for state charts are proposed. These metrics depicts the complexity of ingoing and outgoing transitions, respectively, of composites states.

Lassen *et al.* proposed in [67] three complexity metrics for Workflow nets - a subclass of Petri nets[58]. The calculated metrics are the *Extended Cardoso Metric*, *Extended Cyclomatic Metric* and *Structuredness Metric*. The first metric is an adaptation of Cardoso's Control Flow Complexity (CFC)[68] metric which measures the control flow complexity in business processes, workflows and Web processes. The Cardoso metric is an generalization of McCabe's Cyclomatic Complexity[61]. Extended Cyclomatic Complexity is an adaptation of the McCabe Cyclomatic Complexity[61] to Workflow nets. The last metric measures complexity with respect to the structure of the Workflow net. This is carried out by identifying certain structural patterns in the Workflow net, and by associating cognitive weights to each pattern representing its complexity.

In [69] Genero *et al.* introduced a metric suite to measure structural complexity of Entity-Relationship (ER) diagrams. Guo *et al.* proposed a complexity number for concurrent FSM based embedded software in [70]. The complexity number is defined on the decision diagram representation of the system functionality and indicates the upper bound on the number of required test cases in order to accomplish Condition/Decision coverage.

In [71] the authors discussed how existing software complexity metrics can be applied on BPM. The author discussed the application of metrics like *Line of Code*, *McCabe Cyclomatic Complexity*, *Fan in* and *Fan out* in context of BPM. They state that their presented results are modeling language independent. The adaptation of the McCabe Cyclomatic Complexity[61] has already been discussed by Cardoso in [72] where he proposed the Control Flow Complexity (CFC) metric, which is based on McCabe's metric, for BPM. In [73] two other software metrics are discussed in order to be adapted for BPM. These are the *Halstead Complexity metric*[74] and the *Information Flow metric* by Henry and Kafura[75]. Piattini *et al.* introduced in [76] a metric suite for BPMN models which is based on the measurement framework FMESP[77]. Other metrics known from the software domain have been adapted by Khlif *et al.* in [78]. They adapted coupling metrics for BPMN models.

3. Extended Symbolic Transition System

An ESTS[28, 29, 27], which is tightly related to an UML State Machine (SM)[1], is used to define the behavior of systems. The ESTS is based on the STS defined in [24] and introduces the following additional elements: (1) *delay transition*, (2) *completion transition*, (3) *transition priorities*, (4) *transition execution duration* and (5) *timing groups*. The following definitions are taken from [27] and have been slightly extended to comply with the content of this work.

***Definition* 3.1** (Extended Symbolic Transition System)
An Extended Symbolic Transition System e 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 message parameters, T is the set of transitions, G is the set of timing groups and* q_0 *is the initial configuration.*

The set of labels is defined as $L = L_i \cup L_o \cup L_* \cup \mathbb{N}_1$ with $L_i \cap L_o = \emptyset$. Hence, the set of all input and output messages is $L_{io} = L_i \cup L_o$. $L_* = \{\tau, \gamma\}$ is a set of labels representing the unobservable and completion transitions, whereas $\mathbb{N}_1$ represents the delayed transitions.

Attributes A and message parameters P are both variable sets for the purpose of symbolic treatment of data. The attributes are properties of an ESTS and parameters are used in the context of input and output messages in order to accomplish data transmission. A message parameter of a transition is defined by the function $\mathbf{par}(l)$, where $\mathbf{par}(l) \subseteq P$ if $l \in L_{io}$, otherwise $\mathbf{par}(l) = \emptyset$. The parameter and attribute sets are disjoint, resulting in $A \cap P = \emptyset$, whereas $V = A \cup P$ denotes the set of all variables of an ESTS.

***Definition* 3.2** (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 its value domain containing all possible values of v.* □

The type of a variable is defined due to its corresponding value domain set $\mathfrak{U}^v$. The set $\mathbb{B} = \{true, false\}$ denotes values of the Boolean domain, $\mathbb{R}$ values of the real number domain and $\mathbb{Z}$ values of the integer domain. Thus, a real number variable is defined by $\mathfrak{U}^v = \mathbb{R}$, an integer variable by $\mathfrak{U}^v = \mathbb{Z}$ and a Boolean variable by $\mathfrak{U}^v = \mathbb{B}$.

For a given subset of variables $X \subseteq V$, $\mathfrak{U}^X$ defines the set of all variable valuations for X. In addition the parameter valuation is denoted as $\varsigma \in \mathfrak{U}^{\mathbf{par}(l)}$, whereas the attribute valuation is denoted as $\iota \in \mathfrak{U}^A$.

***Definition* 3.3** (Literal)
A literal is a single value $\mathfrak{c}$ *which corresponds to a certain value domain. Where* $\mathfrak{c}$ *is used for different value domains and therefore* $\mathfrak{c}$ *is denoted as* $\mathfrak{c} \in \mathbb{B}$ *for a Boolean literal,* $\mathfrak{c} \in \mathbb{R}$ *for a real number literal and* $\mathfrak{c} \in \mathbb{Z}$ *for an integer literal, when necessary.* □

***Definition* 3.4** (Transition)
A transition $t \in T$ *is defined as the tuple* $(s, l, \varphi, \rho, p, d, s')$*, where* $s, s' \in S$ *are the source- and target state,* $l \in L$ *defines its label,* $\varphi \in \mathfrak{F}(V)$ *defines its guard,* $\rho \in \mathfrak{A}(V)^A$ *defines the attribute update function,* $p \in \mathbb{N}_0$ *its priority and* $d \in \mathbb{N}_0$ *a lapse of time.* □

The guards are first order logic predicates over the variables in V, with $\mathfrak{F}(V)$ denoting the set of first order logic predicates. The update of an attribute valuation ι is carried out by means of the attribute update function $\rho \in \mathfrak{A}(V)^A$, where $\mathfrak{A}(V)^A$ is a set of update attribute functions, updating the attributes of set A, which depends on the variable set V. The update is denoted as $\iota' = \rho(\varsigma \cup \iota)$, which states that the attribute update function produces a new attribute valuation ι'. The new attribute valuation is calculated based on the current attribute valuation ι and the parameter valuation ς by means of the attribute update function ρ. The parameter valuation ς is an empty set if the transition is neither an input nor an output transition, or the input and the output transition has no message parameter assigned. Furthermore, the identity function $id \in \mathfrak{A}(V)^A$ is defined, stating that an attribute valuation remains unchanged. The priority $p \in \mathbb{N}_0$ of a transition is a natural number which uniquely defines an *execution order* of transitions for states with multiple outgoing transitions. The execution duration $d \in \mathbb{N}_0$ states a lapse of time for a transition and indicates the time needed to execute a transition.

Moreover, the set of outgoing transitions of a particular state s is defined as function **stateOutTrans**(s) which is stated in Equation 3.1. The function returns a set of transitions where the state s is the source state of.

$$\textbf{stateOutTrans}(s) = \{t \in T \mid t.s = s\} \tag{3.1}$$

Equation 3.1 states a specific syntax in order to reference the elements of a tuple. An element of a tuple is referenced by the '.' operator. For instance, referencing either the source or the target state of a transition t is accomplished by the statements $t.s$ and $t.s'$, respectively. This syntax will be used throughout the remainder of this work.

***Definition* 3.5** (Input Transition)
A transition $t \in T$ *is an input transition when its label* l *is in the set of input labels such that* $l \in L_i$*, where* L_i *is the set of input labels. The set* I *of all input transitions is defined as* $T_i = \{t \in T \mid t.l \in L_i\}$ *with* $T_i \subseteq T$*.* □

The function **inputMessages**(s), shown in Equation 3.2, returns a set of labels corresponding to input transitions of the outgoing transition set **stateOutTrans**(s) of state s. On the contrary, Equation

3.3 denotes the function **outputMessages**(s) which returns a set of labels corresponding to output transitions of the outgoing transition set **stateOutTrans**(s) of state s.

$$\textbf{inputMessages}(s) = \{l \in L_i \mid t \in \textbf{stateOutTrans}(s) \wedge t.l = l\} \tag{3.2}$$

$$\textbf{outputMessages}(s) = \{l \in L_o \mid t \in \textbf{stateOutTrans}(s) \wedge t.l = l\} \tag{3.3}$$

The equation stated in Equation 3.4 returns a set of labels. These labels represent the timeout terms of all delay transitions of the outgoing transition set **stateOutTrans**(s) for a given state s.

$$\textbf{timeoutTerm}(s) = \{l \in \mathbb{N}_1 \cup A \mid t \in \textbf{stateOutTrans}(s) \wedge t.l = l\} \tag{3.4}$$

***Definition* 3.6** (Output Transition)
A transition $t \in T$ is an output transition if its label l is related to the set of output labels such that $l \in L_o$, where L_o is the set of output labels. The set T_o of all output transitions is defined as $T_o = \{t \in T \mid t.l \in L_o\}$ with $T_o \subseteq T$. □

Let $E = \{e_1, e_2, ..., e_n\}$ be a set of ESTS with L_i^x is the set of input labels of ESTS $e_x \in E$ with $x \in \mathbb{N}_1$. The set of input labels of the ESTS in set E are disjoint so that $L_i^x \cap L_i^y = \emptyset$ for $x \neq y$.

***Definition* 3.7** (Unobservable Transition)
A transition $t \in T$ is an unobservable transition if its label $l = \tau$. The set T_τ of all unobservable transitions is defined as $T_\tau = \{t \in T \mid t.l = \tau\}$ with $T_\tau \subseteq T$. □

***Definition* 3.8** (Completion Transition)
A transition $t \in T$ is a completion transition if its label $l = \gamma$. The set T_γ of all completion transitions is defined as $T_\gamma = \{t \in T \mid t.l = \gamma\}$ with with $T_\gamma \subseteq T$. □

***Definition* 3.9** (Delay Transition)
A transition $t \in T$ is a delay transition if its label l, which is also called timeout term, is either a numeric value $n \in \mathbb{N}_1$, with $\mathbb{N}_1 = \mathbb{N}_0 \setminus \{0\}$, or an attribute value $a \in A$. The set T_d of all delay transitions is defined as $T_d = \{t \in T \mid l \in \mathbb{N}_1 \vee l \in A\}$ with $T_d \subseteq T$. □

The set $T_{o\gamma\tau} = T_o \cup T_\gamma \cup T_\tau$ denotes the set of all output, completion and unobservable transitions, with $T_{o\gamma\tau} \subseteq T$. These transitions are considered as *instantly executable*. *Instantly executable* means that these transitions are executed as soon as the respective guard evaluates to *true* without any interaction with another ESTS or the *environment*. Whereas interaction with another ESTS or *environment* means that a message is sent or that a specific time has to elapse.

In order to refer to a certain transition in the remainder of this work the notation $(s \xrightarrow{label} s')$ is used. s denotes the source state, s' the destination state and *label* the label of the transition and therefore

its type, such that $label \in L_{io} \cup L_* \cup \mathbb{N}_1$. In case of labels for output and input transitions the label is prefixed by an "!" and "?", respectively.

***Definition* 3.10** (Timing Group)
A timing group g is a tuple $\langle c, S_g, T_d^g, T_r \rangle$, where $c \in C$ is its clock, with C the set of clocks, $S_g \subseteq S$ is a set of states which correspond to the timing group, $T_d^g \subseteq T_d$ 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. $\zeta \in \mathfrak{U}_C$ is the variable valuation of a clock $c \in C$ of a timing group, C is the set of all clocks of the timing groups in G including the ESTS global clock c^*, such that $C = C \cup c^*$. $\mathfrak{U}_C$ is the set of all possible clock values.

The clock c^* keeps tracks of the elapsed time in an ESTS and behaves similar to a timing group with $T_d^g = T$, $S_g = S$ and $T_r = \emptyset$, which implies that the clock is used globally for all states and transitions of the ESTS.

The clock update function ξ is denoted as $\zeta' = \xi(\zeta, n)$, where n specifies the time difference used to update the clock valuation of all clocks in C.

***Definition* 3.11** (Configuration)
A configuration of an ESTS is a triple (s, ι, ζ) of a state $s \in S$, an attribute valuation $\iota \in \mathfrak{U}^A$ and a clock valuation $\zeta \in \mathfrak{U}^C$. □

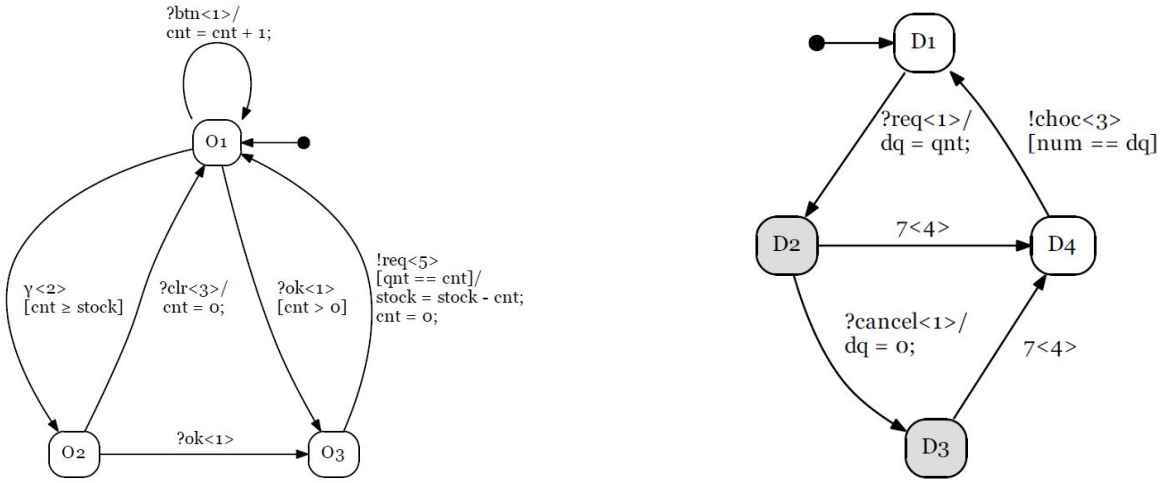
The set of all configurations is defined as $Q = (S \times \mathfrak{U}^A \times \mathfrak{U}^C)$. A single configuration is denoted as $q \in Q$ and the initial configuration q_0 is given as $q_0 = (s_0, \iota_0, \zeta_0)$ with $q_0 \in Q$.

As stated in [27] the behavior of an ESTS is described by the *executions* of its transitions. The term *execution* means that a state change from the source to the destination state of a transition occurs. In addition to that, the execution of a transition involves the update of the attribute valuation according to its attribute update function ρ. A transition can only be executed when it is enabled, which is the case if the corresponding guard evaluates to *true*.

***Definition* 3.12** (Enabled Transition)
A transition is enabled if for a given configuration, parameter and attribute valuation, the corresponding guard evaluates to true. Besides the satisfiability of the guard, a delay transition $t \in T_d^g$ must satisfy $c >= n$ as well, where c is the clock of timing group g and n is the delay of the delay transition, with $n \in \mathbb{N}_1$. □

An input transition is enabled as soon as the corresponding message with its possibly empty parameter valuation is received and the guard is satisfied. Completion, unobservable and output transitions are enabled as soon as the source state of the transition is reached and the corresponding guard evaluates to *true*. Additionally a delay transition has to fulfill a time constraint. Thus, it might be necessary that the timing group clock of its respective timing group must be increased by $n - c$ in order to enable the transition.

The transition priority p denotes an *execution order* for enabled transitions corresponding to the same label. Where the execution order states a ranking according to the priority such that the enabled transition with the highest priority will be executed and all other enabled transitions with lower priorities will be ignored.



(a)Order Component (b)Delivery Component

Figure 3.1.: Chocolate vending machine depicted as ESTSs. (Images adapted from [27])

Example 3.1 (Chocolate Vending Machine)
Figure 3.1 depicts the Chocolate Vending Machine, which consist of two components. The first component is the Order Component, which is illustrated by the ESTS in Figure 3.1a. Figure 3.1b depicts the second component, namely the Delivery Component.

The Order Component consists of three states, namely $O1$, $O2$ and $O3$. Transition $O3 \xrightarrow{!req} O1$ denotes an output transition which triggers the message *req*. The value in angle brackets $< 5 >$ is related to the transition execution time, whereas the expression enclosed in square brackets corresponds to the guard of the transition. Expressions $stock = stock - cnt$ and $cnt = 0$ denote the actions which are associated to the transition. The states of the Delivery Component which are filled gray belong to a timing group.

The Order Component is used in order to define the amount of chocolates which shall be requested from the Delivery Component. By an external input message *?btn*, which states the fact that the environment detected that a button has been pressed, the order counter *cnt* will be incremented. The reception of the input message *?ok* triggers the transmission of the output message *!req*. This message is received from the Delivery Component by means of the input message *?req*. The component either waits a configured timeout, depicted by delay transition $D2 \xrightarrow{7} D4$, or for an input message *?cancel* which indicates that the delivery process shall be canceled. Finally, the output message *!choc* is sent with the amount of chocolates to be dispensed. □

4. Static Analysis

This chapter describes the approach in order to analyze an ESTS model in a static manner. Therefore, this chapter provides a description of the check and metric suite for an ESTS model. First of all, in Section 4.1 preliminaries in order to apply static analysis to an ESTS are provided. In Sections 4.2 and 4.3 checks are introduced addressing consistency and structural issues, respectively. This chapter concludes with the introduction of metrics to measure data flow, control flow, structural complexity as well as complexity in terms of size.

4.1. Preliminaries

In this section, preliminary data structures are introduced which are utilized by checks and metrics discussed later on. In Section 4.1.1 the Binary Expression Tree (BET) data structure is introduced, which represents guard and action expressions as a binary tree. Section 4.1.2 presents a tree representation of the possible paths through an ESTS, namely an Independent Path Tree (IPT). In Section 4.1.3 the data structure *Shortest Distance Map* is discussed. The map contains the shortest distance of all transition pairs of an ESTS. In Section 4.1.4 the *Def-Use* data structure is introduced, which illustrates the definition and use of attributes.

Furthermore, constraint solving is discussed in Section 4.1.5, which is utilized by some checks. At last in Section 4.1.6 two statistical instruments, namely *mean value* and *standard deviation*, are discussed.

4.1.1. Guards and Actions as Binary Expression Tree

This section introduces the BET[79] data structure, which depicts a different way to represent guard and action expressions.

***Definition* 4.1** (Expression)
An expression exp *is a tuple* $\langle op, opd_l, opd_r \rangle$*, where* op *denotes the operator,* opd_l *the left operand and* opd_r *the right operand. A not used operand is denoted with NIL. An expression which corresponds to a unary operator only has the left operand set, such that* $opd_l \neq NIL$ *and* $opd_r = NIL$*. By contrast, for a binary operator either operands are set. Moreover, the operands of an expression can be an expression again, a literal or a variable valuation, such that* $opd_l, opd_r \in \{exp, \mathfrak{c}, \mathfrak{U}^V\}$*. The function* $\boldsymbol{bet}()$ *denotes the transformation of a guard or action to a BET, such that* $\boldsymbol{bet}(\varphi) : \varphi \longmapsto exp$ *and* $\boldsymbol{bet}(\rho) : \rho \longmapsto exp$ □

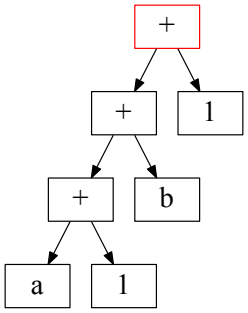


Figure 4.1.: Binary Expression Tree of expression $a + 1 + b + 1$.

A BET depicts a tree with expression exp_{root} denoting the root expression, whose operator corresponds to the root node of the tree. An expression is divided into the following types: *(1) arithmetic*, *(2) assignment*, *(3) comparison* and *(4) logical*.

***Definition* 4.2** (Arithmetic Expression)
An expression exp is considered an arithmetic expression if $\text{exp}.op \in ArithmOp$ *with* $ArithmOp = \{+, -, *, /, \%, ++, --\}$. *The set of all arithmetic expressions is $ArithmExp$.* □

***Definition* 4.3** (Assignment Expression)
An expression exp is considered an assignment expression if $exp.op \in AssignOp$ with $AssignOp = \{=\}$. The set of all assignment expressions is $AssignExp$. □

***Definition* 4.4** (Comparison Expression)
An expression exp is considered a comparison expression if $exp.op \in ComparisionOp$ with $ComparisionOp = \{<, \leq, >, \geq, ==, != \}$. The set of all comparison expressions is $ComparisionExp$. □

***Definition* 4.5** (Logical Expression)
An expression exp is considered a logical expression if $exp.op \in LogicalOp$ with $LogicalOp = \{\&\&, \|, !\}$. The set of all logical expressions is $LogicalExp$. □

Example 4.1 (BET)
Consider the example BET in Figure 4.8, which depicts the arithmetic expression $x + 1 + b + 1$. Taking the definitions of this section into account, the following expressions are defined.

- $exp_0 = \langle +, a, 1 \rangle$
- $exp_1 = \langle +, exp_0, b \rangle$
- $exp_2 = \langle +, exp_1, 1 \rangle$

The expression exp_2 depicts the root expression exp_{root} of the BET. Thus, $exp_{root} = exp_2$ and the red plus operator in Figure 4.8 corresponds to the root expression operator $exp_{root}.op$.

□

4.1.2. Independent Path Tree

This section introduces a different structure to represent an ESTS namely an IPT. An IPT consists of all possible independent paths within an ESTS so that each path in the tree is unique. The structure depicts a convenient intermediate view of an ESTS, on which some of the upcoming checks rely in order to ease their application.

***Definition* 4.6** (Independent Path Tree)
An independent path tree $\mathfrak{T}$ *is a tuple* $\langle \mathfrak{N}, \mathfrak{E} \rangle$*, where* $\mathfrak{N}$ *denotes the set of nodes and the set* $\mathfrak{E}$ *referring to all edges of the tree. The function* ***estsToPathTree***$(e) : e \longmapsto \mathfrak{T}$ *denotes the conversion of an ESTS* e *to an IPT* $\mathfrak{T}$*.* □

***Definition* 4.7** (Node)
A node $\mathfrak{n} \in \mathfrak{N}$ *is a tuple* $\langle s \rangle$*, where* s *denotes the corresponding ESTS state.* $\mathfrak{N}$ *denotes the set of nodes. The node* $\mathfrak{n}_0$ *represents the root node of the tree.* □

The nodes of an IPT represent the states of an ESTS. Various nodes can refer to the same state, which means that the number of nodes is greater than or equal to the number of states: $|\mathfrak{N}| \geq |S|$.

***Definition* 4.8** (Edge)
An edge $\mathfrak{e} \in \mathfrak{E}$ *is defined as the tuple* $\langle \mathfrak{n}, t, \mathfrak{n}' \rangle$ *with* $\mathfrak{n}, \mathfrak{n}' \in \mathfrak{N}$*.* $\mathfrak{n}$ *refers to the source node and* $\mathfrak{n}'$ *to the destination node. Hence,* $\mathfrak{n}$ *is considered to be the parent node of* $\mathfrak{n}$ *and* $\mathfrak{n}'$ *is the child node of* $\mathfrak{n}$ *by implication.* $t \in T$ *denotes the corresponding transition of the ESTS, which is represented by edge* $\mathfrak{e}$*.* □

In order to use the tree structure, functions are defined to retrieve the following information:

- The parent node of a given node as depicted in Equation 4.1.
- The child nodes of a given node as shown in Equation 4.2.
- The outgoing transitions of a node to its children as illustrated in Equation 4.3.
- The incoming transition by which the node is connected to its parent node as depicted in Equation 4.4.

As shown in Equation 4.1, for a given node $\hat{\mathfrak{n}}$ the parent node $\mathfrak{n}_p$ belongs to an edge $\mathfrak{e}$, whose source node is equal to the parent node and its destination node is equal to the given node. Thus, in the set $\mathfrak{E}$ exists an edge $\mathfrak{e}$ such that $\mathfrak{e}.\mathfrak{n} = \mathfrak{n}_p \wedge \mathfrak{e}.\mathfrak{n}' = \hat{\mathfrak{n}}$. If a node has a parent node, precisely one edge exists where this definition applies. The root node $\mathfrak{n}_0$ is an exception as this node has no parent node. Thus, a node cannot have more than one parent node. The function **parentNode($\hat{\mathfrak{n}}$)**, which is depicted in Equation 4.1, returns only a single value which is either the determined parent node or *NIL*. By contrast, *NIL* indicates that no parent node is present.

$$\textbf{parentNode}(\hat{\mathfrak{n}}) = \begin{cases} \mathfrak{n}_p \in \mathfrak{N} & \mathfrak{e} \in \mathfrak{E} \wedge \mathfrak{n} = \mathfrak{n}_p \wedge \mathfrak{n}' = \hat{\mathfrak{n}} \\ \mathit{NIL} & \text{otherwise} \end{cases} \tag{4.1}$$

Equation 4.2 states the formula in order to retrieve a set of nodes which are denoted as child nodes of a particular node. A child node $\mathfrak{n}_c$ is defined as the node which functions as destination node $\mathfrak{n}'$ of an edge, whereas the source node n corresponds to the given node $\hat{\mathfrak{n}}$. The function **childNodes**($\hat{\mathfrak{n}}$) returns a set of nodes or an empty set, whereas the empty set denotes that a node has no child nodes at all.

$$\textbf{childNodes}(\hat{\mathfrak{n}}) = \{\mathfrak{n}_c \in \mathfrak{N} \mid \mathfrak{e} \in \mathfrak{E} \wedge \mathfrak{e}.\mathfrak{n} = \hat{\mathfrak{n}} \wedge \mathfrak{e}.\mathfrak{n}' = \mathfrak{n}_c\} \tag{4.2}$$

Equation 4.3 depicts the function **nodeOutTrans**($\hat{\mathfrak{n}}$), which returns a set of transitions related to those edges where the given node $\hat{\mathfrak{n}}$ is the source node, such that $\mathfrak{n} = \hat{\mathfrak{n}}$.

$$\textbf{nodeOutTrans}(\hat{\mathfrak{n}}) = \{\mathfrak{e}.t \in T \mid \mathfrak{e} \in \mathfrak{E} \wedge \mathfrak{e}.\mathfrak{n} = \hat{\mathfrak{n}}\} \tag{4.3}$$

Equation 4.4 states the function **nodeInTrans**($\hat{\mathfrak{n}}$). The function returns the incoming transition which corresponds to the edge which connects the given node $\hat{\mathfrak{n}}$ to its parent node. Thus, the desired transition belongs to edge $\mathfrak{e}$ such that $\mathfrak{e}.\mathfrak{n}' = \hat{\mathfrak{n}}$. Since a node can either have one parent node or no parent node at all, the function returns either a single transition or *NIL*.

$$\textbf{nodeInTrans}(\hat{\mathfrak{n}}) = \begin{cases} \mathfrak{e}.t \in T & \mathfrak{e} \in \mathfrak{E} \wedge \mathfrak{e}.\mathfrak{n} = \hat{\mathfrak{n}} \\ \mathit{NIL} & \text{otherwise} \end{cases} \tag{4.4}$$

***Definition* 4.9** (Path)
*A path $\mathfrak{p} \in \mathfrak{P}^{\mathfrak{T}}$ is an ordered list $(\mathfrak{e}_x, \mathfrak{e}_{x+1}, ..., \mathfrak{e}_m)$, with $x \geq 1$ and $\mathfrak{p} \subseteq \mathfrak{E}$, of edges. $\mathfrak{P}^{\mathfrak{T}}$ denotes the set of all paths of tree $\mathfrak{T}$. The function **treeToPaths**($\mathfrak{T}$): $\mathfrak{T} \longmapsto \mathfrak{P}^{\mathfrak{T}}$ denotes the computation of the independent path set $\mathfrak{P}^{\mathfrak{T}}$.* □

The tree $\mathfrak{T}$ is considered an IPT when all paths of the set $\mathfrak{P}^{\mathfrak{T}}$ are mutually distinct. Let $\mathfrak{P}^{\mathfrak{T}} = \{\mathfrak{p}_0, \mathfrak{p}_1, ..., \mathfrak{p}_m\}$ be a set of all paths of $\mathfrak{T}$. Then $\mathfrak{T}$ is considered as IPT if for any arbitrary pair of paths $\mathfrak{p}_i, \mathfrak{p}_j \in \mathfrak{P}$ holds that $\mathfrak{p}_i \setminus \mathfrak{p}_j \neq \emptyset$ with $i, j \in \mathbb{N}_0$ and $i \neq j$. Thus, mutually distinct in terms of paths means that every arbitrary pair of paths of set $\mathfrak{P}^{\mathfrak{T}}$ differs in at least one edge.

$$\textbf{hasOutgoingTransitions}(\hat{s}) = \begin{cases} \mathit{false} & \nexists\, t \in T : t.s = \hat{s} \\ \mathit{true} & \text{otherwise} \end{cases} \tag{4.5}$$

In addition the destination node $\mathfrak{n}'$ of the last edge $\mathfrak{e}_m$ of the path, with $\mathfrak{e}_m \in \mathfrak{p}$, precisely shows one of the following characteristics:

1. The state corresponding to node $\mathfrak{e}_m.\mathfrak{n}'$ has no outgoing transitions at all, which is stated in Equation 4.5.

2. The state belonging to node $\mathfrak{e}_m.\mathfrak{n}'$ corresponds to another source or destination node of an edge which is unequal to the last edge of the path, which is stated in Equation 4.6. Furthermore, this case denotes the presence of a loop in the path.

Moreover node $\mathfrak{e}_m.\mathfrak{n}'$ always belongs to a leaf node of tree $\mathfrak{T}$, such that $\textbf{nodeOutTrans}(\mathfrak{e}_m.\mathfrak{n}') = \emptyset$.

$$\textbf{containsLoop}(\mathfrak{p}) = \begin{cases} \textit{true} & \exists\ \mathfrak{e} \in \mathfrak{p} : (\mathfrak{e}.\mathfrak{n}.s = \mathfrak{e}_m.\mathfrak{n}'.s) \vee (\mathfrak{e}.\mathfrak{n}'.s = \mathfrak{e}_m.\mathfrak{n}'.s) \\ \textit{false} & \text{otherwise} \end{cases} \tag{4.6}$$

Example 4.2 (Independent Path Tree)
In Figure 4.2a an ESTS is depicted and in Figure 4.2b its corresponding path tree. In respect to the tree in Figure 4.2b the node set is $\mathfrak{N} = \{\mathfrak{n}_0, \mathfrak{n}_1, \mathfrak{n}_2, \mathfrak{n}_3, \mathfrak{n}_4, \mathfrak{n}_5\}$ with:

- $\mathfrak{n}_0 = A1$
- $\mathfrak{n}_1 = A2$
- $\mathfrak{n}_2 = A3$
- $\mathfrak{n}_3 = A1$
- $\mathfrak{n}_4 = A4$
- $\mathfrak{n}_5 = A4$

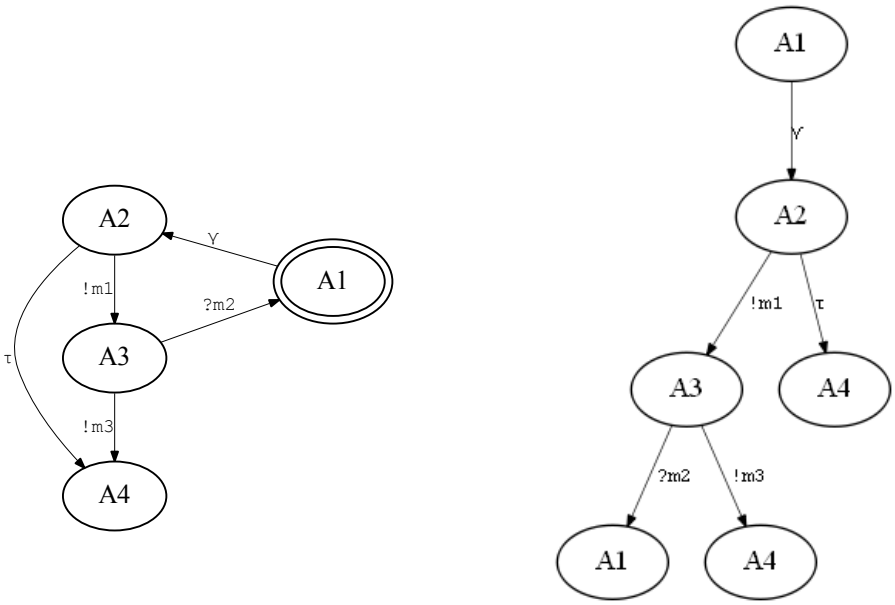
The corresponding edge set is $\mathfrak{E} = \{\mathfrak{e}_0, \mathfrak{e}_1, \mathfrak{e}_2, \mathfrak{e}_3, \mathfrak{e}_4\}$ with:

- $\mathfrak{e}_0 = \{\mathfrak{n}_0, (A1 \xrightarrow{\gamma} A2), \mathfrak{n}_1\}$
- $\mathfrak{e}_1 = \{\mathfrak{n}_1, (A2 \xrightarrow{!m1} A3), \mathfrak{n}_2\}$
- $\mathfrak{e}_2 = \{\mathfrak{n}_1, (A2 \xrightarrow{\tau} A4), \mathfrak{n}_5\}$
- $\mathfrak{e}_3 = \{\mathfrak{n}_2, (A2 \xrightarrow{?m2} A1), \mathfrak{n}_3\}$
- $\mathfrak{e}_4 = \{\mathfrak{n}_2, (A2 \xrightarrow{!m3} A4), \mathfrak{n}_4\}$

By applying function **treeToPaths**() to the tree of Figure 4.2b three paths can be retrieved such that $\mathfrak{P} = \{\mathfrak{p}_1, \mathfrak{p}_2, \mathfrak{p}_3\}$ with:

- $\mathfrak{p}_1 = \{\mathfrak{e}_0, \mathfrak{e}_1, \mathfrak{e}_3\}$
- $\mathfrak{p}_2 = \{\mathfrak{e}_0, \mathfrak{e}_1, \mathfrak{e}_4\}$
- $\mathfrak{p}_3 = \{\mathfrak{e}_0, \mathfrak{e}_2\}$

The path tree in Figure 4.2b is being considered as an IPT since all paths are independent due to the fact that $\mathfrak{p}_1 \setminus \mathfrak{p}_2 = \{\mathfrak{e}_3\}$, $\mathfrak{p}_1 \setminus \mathfrak{p}_3 = \{\mathfrak{e}_1, \mathfrak{e}_3\}$ and $\mathfrak{p}_2 \setminus \mathfrak{p}_3 = \{\mathfrak{e}_1, \mathfrak{e}_4\}$. □



(a)Example ESTS.

(b)Corresponding path tree of the example ESTS.

Figure 4.2.: An example of an ESTS and its depiction as path tree.

Algorithm 1 represents the concrete algorithm to transform an ESTS into an IPT. The computation starts at the start state s_0 of an ESTS. For this reason a node $\mathfrak{n}$ corresponding to the start state is created as shown by the statement $\mathfrak{n} \leftarrow \langle s_0 \rangle$. Afterwards the node is added to the variable *element* which is a tuple $\langle \mathfrak{n}, t_{next} \rangle$, with $\mathfrak{n}$ corresponding to a node of the tree which is added to the node set $\mathfrak{N}$ later on, and t_{next} denotes the outgoing transition of $\mathfrak{n}.s$ which should be traversed next. In addition, t_{next} can also be *NIL*, which indicates that no specific outgoing transition is set. The variable *element* basically denotes the current state of an ESTS and its outgoing transition, which are currently being considered during path tree creation.

The algorithm uses a while loop in order to traverse the ESTS. The loop is executed as long as the Boolean variable *treeCompleted* is $false$, which is set to *true* when no further element to process is present, such that $element = NIL$. This fact indicates that all states and transitions of the ESTS have been processed.

Algorithm 1 Conversion ESTS to IPT

```
1:  function CREATE_INDEPENDENT_PATH_TREE(E) → 𝔗
2:      𝔗 ← ⟨𝔑, 𝔈⟩;
3:      stackBranches ← ∅;
4:      𝔭 ← ∅;
5:
6:      𝔫 ← ⟨s_0⟩;
7:      element ← ⟨𝔫, NIL⟩;
8:
9:      treeCompleted ← false;
10:     while treeCompleted == false do
11:         𝔑 ← 𝔑 ∪ element.𝔫;
12:         t_next ← DETERMINE_NEXT_TRANSITION(element, 𝔭, stackBranches);   ▷ Defined in Algorithm 2
13:         if t_next ≠ NIL then
14:             𝔫_s' ← ⟨t_next.s'⟩;
15:             𝔢 ← ⟨element.𝔫, t_next, 𝔫_s'⟩;
16:             𝔈 ← 𝔈 ∪ 𝔢;
17:             𝔭 ← 𝔭 ∪ 𝔢;
18:             element ← ⟨𝔫_s', NIL⟩;
19:         else
20:             element ← NIL;
21:         end if
22:
23:         if element == NIL ∧ stackBranches.ISEMPTY() == false then
24:             branch ← stackBranches.POP();
25:             element ← branch.element;
26:             𝔭 ← branch.𝔭;
27:         end if
28:
29:         if element == NIL then
30:             treeCompleted ← true;
31:         end if
32:     end while
33:
34:     return 𝔗;
35: end function
```

Each iteration through the loop consists of the following steps:

1. Add the corresponding node of the element *element* to the set of nodes $\mathfrak{N}$, such that $\mathfrak{N} = \mathfrak{N} \cup element.\mathfrak{n}$.
2. Determine the next transition t_{next} by calling procedure *DETERMINE_NEXT_TRANSITION*, which is defined in Algorithm 2, based on the current *element*. The next transition t_{next} is either a transition of the outgoing transition set of state $element.\mathfrak{n}.s$ or *NIL* when no next transition is returned. Thus, $t_{next} \in$ **stateOutTrans**$(element.\mathfrak{n}.s) \vee t_{next} = NIL$. Procedure *DETERMINE_NEXT_TRANSITION* expects as parameter the element *element*, the current path under construction $\mathfrak{p}$ and the stack *stackBranches*. The given path $\mathfrak{p}$ denotes the current path and is used for loop detection. The parameter *stackBranches* depicts a stack which holds all branches which are being detected and not yet being processed during tree construction.

Algorithm 2 consists of the following 3 cases:

a) If the corresponding transition t_{next} of the given *element* is not *NIL*, the transition $element.t_{next}$ is returned as the next transition to be processed.
b) If the constructed path $\mathfrak{p}$ contains a loop, such that **containsLoop**$(\mathfrak{p}) = true$, *NIL* is returned, which indicates that the end of this path has been reached.
c) When none of the previous cases arises the outgoing transitions of the current state $element.\mathfrak{n}.s$ are considered. On that account these transitions are converted to the list *transitionList*. If the list contains more than one transition, branches are present which must be saved in order to handle them later. The reason is that the first transition in the list is considered to be part of the currently constructed path, whereas the remaining transitions depict branches which have to be processed after the current path is finished. Thus, for all transitions of the list, except the first one, a *branch* object has to be created. A *branch* object is a tuple $\langle element, \mathfrak{p} \rangle$. While *element* defines at which state and consequently which transition the branch has been created, $\mathfrak{p}$ denotes the path which has been constructed so far at the time the branch is created. Next, the branches are pushed onto stack *stackBranches*. Finally, the first transition is returned or if the state has no outgoing transitions at all, such that **stateOutTrans**$(element.\mathfrak{n}.s) = \emptyset$ applies, *NIL* is returned.

3. In the event that the next transition determined t_{next} is not *NIL*, the following steps are applied:
   a) Create a new node $\mathfrak{n}_{s'}$ for the target state s' of t_{next}.
   b) Create a new edge $\mathfrak{e}$ whose source node is the node of the current element, the destination node is node $\mathfrak{n}_{s'}$ and the corresponding transition is t_{next}.
   c) Add the edge created to the edge set $\mathfrak{E}$.
   d) Extend the current path $\mathfrak{p}$ with the edge created, such that $\mathfrak{p} = \mathfrak{p} \cup \mathfrak{e}$.
   e) Set the current element *element* to node $\mathfrak{n}_{s'}$ in order to traverse transition t_{next}. Thus, $\mathfrak{n}_{s'}$ will be handled in the next loop iteration.
4. If the current element *element* is *NIL*, the algorithm attempts to continue path construction at the last saved branch. Thus, the last saved branch is retrieved from stack *stackBranches* as long as *stackBranches* comprises branches. In order to continue at a branch the *element* variable has to be set to the corresponding branch element *branch.element* and the current path $\mathfrak{p}$ has to be set to $branch.\mathfrak{p}$. In the next iteration the algorithm therefore continues the tree construction from state $branch.element.\mathfrak{n}.s$ and the next transition to be processed is already predefined by $branch.element.t_{next}$, which in case of a branch is never *NIL*.
5. If the *element* is still *NIL* at the end of a loop iteration, the path tree construction is considered finished.

Algorithm 2 Determination of next transition to be traversed.

```
1:  function DETERMINE_NEXT_TRANSITION(element, p, stackBranches) → t_next
2:      if element.t_next ≠ NIL then
3:          return element.t_next;
4:      end if
5:
6:      if containsLoop(p) ≠ true then                ▷ Application of Equation 4.6
7:          return NIL;
8:      end if
9:
10:     transitionList ← TO_LIST(stateOutTrans(element.n.s));
11:     if transitionList.SIZE() > 1 then
12:         for i ← 2, transitionList.SIZE() do
13:             element_branch ← ⟨element.n, transitionList.GET(i)⟩;
14:             branch ← ⟨element_branch, p⟩;
15:             branchStack.PUSH(branch);
16:         end for
17:     end if
18:
19:     if transitionList.SIZE() > 0 then
20:         return transitionList.GET(1);
21:     end if
22:     return NIL;
23: end function
```

4.1.3. Shortest Transition Distance Map

In the following a data structure will be introduced which is called *Shortest Distance Map*. The map contains the shortest distances between all pairs of transitions within an ESTS. Distance is measured in terms of the number of states which are located between a pair of transition. The data structure will be used by several checks which are discussed in this chapter.

***Definition* 4.10** (Shortest Distance Map)
A shortest distance map $\mathfrak{D}$ *consists of a finite number of distance map entries* $\mathfrak{d}$*, where a single entry corresponds to the shortest distance between a single pair of transitions. A distance map entry* $\mathfrak{d}$ *is a tuple* $\langle t,\ t',\ dist\rangle$*,* t *denotes the start transition of the shortest path,* t' *the end transition and dist denotes a natural value, such that* $dist \in \mathbb{N}_0$*, which represents the minimum number of states which are located between the transition pair.* □

The map only comprises values for transition pairs where a path is possible. Otherwise *NIL* is returned as shown in Equation 4.7. The equation states the function **shortestDistance**(t, t') which returns the computed distance for a given pair of transitions. In the event that $t = t'$, a distance value not equal to *NIL* is returned as long as a path exists, which implies the presence of a loop.

$$\textbf{shortestDistance}(t, t') = \begin{cases} \mathfrak{d}.dist & \mathfrak{d} \in \mathfrak{D} \wedge \mathfrak{d}.t = t \wedge \mathfrak{d}.t' = t' \\ NIL & \text{otherwise} \end{cases} \tag{4.7}$$

The algorithm applied in order to create the *Shortest Distance Map* is based on the *Dijkstra's Algorithm*[80]. But in contrast to the *Dijkstra Algorithm* the paths are not computed in terms of edges, but rather with respect to nodes. In the case of an ESTS, edges refer to transitions and nodes to states. Thus, for a given start transition transition $t_0 \in T$, the algorithm computes the shortest paths and distances to all transitions of the set T. In order to compute the distances for the entire ESTS the algorithm has to be called for each transition of the set, such that $\forall_{t_0 \in T}$ **computeShortestDistance**(t_0), where **computeShortestDistance**(t) denotes the application of the algorithm.

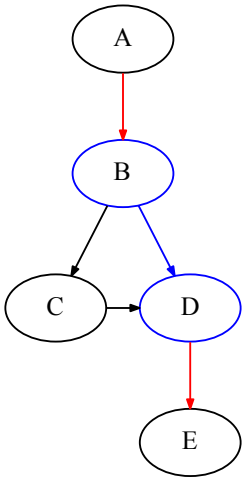


Figure 4.3.: Example of an ESTS to demonstrate shortest path computation.

Example 4.3 (ESTS)
Based on the ESTS in Figure 4.3 the shortest distances are computed. Applying function **computeShortestDistance**(t) to each transition of set T produces the following results:

1. The iteration of the algorithm with $t_0 = (A \to B)$ produces the following map entries:
   a) $\mathfrak{d}_0 = \{(A \to B), (B \to C), 1\}$
   b) $\mathfrak{d}_1 = \{(A \to B), (C \to D), 2\}$
   c) $\mathfrak{d}_2 = \{(A \to B), (D \to E), 2\}$
   d) $\mathfrak{d}_3 = \{(A \to B), (B \to D), 1\}$
2. The iteration of the algorithm with $t_0 = (B \to C)$ produces the following map entries:
   a) $\mathfrak{d}_4 = \{(B \to C), (C \to D), 1\}$
   b) $\mathfrak{d}_5 = \{(B \to C), (D \to E), 2\}$
3. The iteration of the algorithm with $t_0 = (C \to D)$ produces the following map entries:
   a) $\mathfrak{d}_6 = \{(C \to D), (D \to E), 1\}$

4. The iteration of the algorithm with $t_0 = (B \rightarrow D)$ produces the following map entries:
   a) $\mathfrak{d}_7 = \{(B \rightarrow D), (D \rightarrow E), 1\}$
5. The iteration of the algorithm with $t_0 = (D \rightarrow E)$ produces no entries as no path to any other transition exists.

Thus, eight distance map entries are created, such that $\mathfrak{D} = \{\mathfrak{d}_0, \mathfrak{d}_1, \mathfrak{d}_2, \mathfrak{d}_3, \mathfrak{d}_4, \mathfrak{d}_5, \mathfrak{d}_6, \mathfrak{d}_7\}$. When computing the distance for the transition pair $(A \rightarrow B)$, $(D \rightarrow E)$, which are highlighted in red in the figure, the shortest path principle is applied. This example shows that transition $(D \rightarrow E)$ is reachable over two paths, with the blue path being shorter as only two states, B and D, have to be visited in comparison to three states, B, C and D, for the other path. □

4.1.4. Attribute Def-Use Data Structure

This section introduces the *Def-Use* data structures for attributes. The *Def-Use* pattern [81, 82] is used in the context of data flow analysis in order to collect information regarding the modification and use of variables in a program. A *definition (def)* of a variable occurs when a new value is assigned, whereas a *use* occurs when the variable is referenced. The *Def-Use* data structure for attributes is realized by means of the sets *Defs* and *Uses*.

***Definition* 4.11** (Defs)
The definition set Defs consists of definition entries $def \in Defs$. An entry def is a tuple $\langle a, t, exp, RefUses \rangle$. $a \in A$ denotes the attribute which has been defined at transition $t \in T$ and by means of expression $exp \in \{\boldsymbol{bet}(\varphi), \boldsymbol{bet}(\rho)\}$. The exp element represents a guard or action, respectively, in the BET structure. Furthermore, an entry contains all use statements, represented by the set RefUses, which depend on the definition statement, with $RefUses \subseteq Uses$ □

***Definition* 4.12** (Uses)
The use set Uses consists of use entries $use \in Uses$, which is a tuple $\langle a, t, exp, RefDefs \rangle$. $a \in A$ denotes the attribute which has been referenced at transition $t \in T$ and in expression $exp \in \{\boldsymbol{bet}(\varphi), \boldsymbol{bet}(\rho)\}$. The exp element represents a guard or action, respectively, in the BET structure. Furthermore, an entry contains all def statements, represented by the set RefDefs, on which the use statement depends, with $RefDefs \subseteq Defs$. □

In Equation 4.8 the function **getUseEntry**$(\hat{a}, \hat{t}, \hat{exp})$, which returns a set of all *use* entries that correspond to the given attribute $\hat{a}$, transition $\hat{t}$ and expression $\hat{exp}$.

$$\textbf{getUseEntry}(\hat{a}, \hat{t}, \hat{exp}) = \begin{cases} use & use \in Uses \wedge \hat{a} = use.a \wedge \hat{t} = use.t \wedge \hat{exp} = use.exp \\ NIL & \text{otherwise} \end{cases} \tag{4.8}$$

Furthermore, the function **extractAttributeReferences**(exp), with $exp \in \{\textbf{bet}(\varphi), \textbf{bet}(\rho)\}$, is introduced, which returns the set $A_{refs} = \{a_0, ..., a_n\}$, with $a_0, ..., a_n \in A$. A_{refs} contains all attributes which are referenced by the given expression exp. Let $exp = \textbf{bet}(\rho)$ be an action with $\rho \in \mathfrak{A}'(V')^{A'}$,

where $\mathfrak{A}'(V')^{A'}$ is a subset of $\mathfrak{A}(V)^{A}$, such that $\mathfrak{A}'(V')^{A'} \subseteq \mathfrak{A}(V)^{A}$. $\mathfrak{A}'(V')^{A'}$ denotes the set of actions which actually only contains the action ρ. Thus, the set $A' \subseteq A$ contains the attributes which are updated by the action and the set $V' \subseteq V$ denotes the variables which are referenced by the action in order to update the attributes of set A'. Therefore, the resulting set $A_{refs} = V' \cap A$.

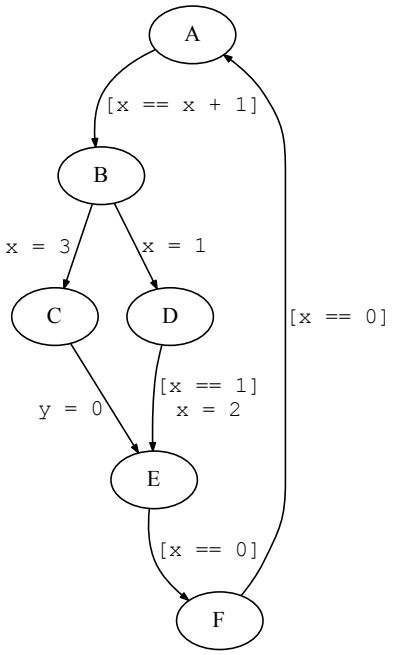


Figure 4.4.: Example of an ESTS to demonstrate *def-use* data structure.

Example 4.4 (ESTS)
Consider the example of the ESTS in Figure 4.4, which consists of six states, A, B, C, D, E, F. The transition types have been neglected and only guards which are enclosed by square brackets and actions are shown as transition-based information. Based on the figure the following *def* and *use* entries are created, whereas no def-use relations have been established yet:

- $def_0 = \{x,\ (B \to C),\ x = 3,\ \emptyset\}$
- $def_1 = \{x,\ (B \to D),\ x = 1,\ \emptyset\}$
- $def_2 = \{y,\ (C \to E),\ y = 0,\ \emptyset\}$
- $def_3 = \{x,\ (D \to E),\ x = 2,\ \emptyset\}$

- $use_0 = \{x,\ (A \to B),\ x = x + 1,\ \emptyset\}$
- $use_1 = \{x,\ (D \to E),\ x == 1,\ \emptyset\}$
- $use_2 = \{x,\ (E \to F),\ x == 0,\ \emptyset\}$
- $use_3 = \{x,\ (F \to A),\ x == 0,\ \emptyset\}$

Taking the structure of the ESTS into account the following reference sets *RefUses* and *RefDefs* for each def and use entry are created.

- $RefUses^{def_0} = \{use_0,\ use_2,\ use_3\}$. The set contains use_0 due to the loop in the ESTS.
- $RefUses^{def_1} = \{use_1\}$
- $RefUses^{def_2} = \emptyset$
- $RefUses^{def_3} = \{use_0,\ use_2,\ use_3\}$

- $RefDefs^{use_0} = \{def_0,\ def_3\}$
- $RefDefs^{use_1} = \{def_1\}$
- $RefDefs^{use_2} = \{def_0,\ def_3\}$
- $RefDefs^{use_3} = \{def_0,\ def_3\}$

As the reference set $RefUses^{def_2}$ is the empty set, this definition of the variable y has no effect of any other statement in the ESTS. Thus, this definition is a *dead definition*.

Applying function **getUseEntry**$(x, D \rightarrow E, \text{x == 1})$ results in the set $\{use_1\}$. Finally, applying function **extractAttributeReferences**$(x == 1)$ results in the set $AttributeRefs = \{x\}$.

□

4.1.5. Constraint Solving

Several checks introduced in this thesis utilize the technique of *Constraint Solving*. Basically a constraint is defined as a restriction of a set of possibilities [83]. In terms of an ESTS a guard represents a constraint which restricts the possible values of variables which are used in the guard. The goal of *constraint solving* is to find for each variable of the constraint a value so that the constraint evaluates to *true*, otherwise the constraint is considered *not solvable*. The solving of constraints is carried out by means of a *Constraint Solver*. An example of such solvers is the *Choco Constraint Solver*[42].

$$\textbf{solveConstraint}(\mathfrak{c}) = \begin{cases} \phi \subset \mathfrak{U}^{V'} & \top(\mathfrak{c}) = true \\ \bot & \top(\mathfrak{c}) = false \end{cases} \tag{4.9}$$

Equation 4.9 introduces the function **solveConstraint(c)**, which depicts the *constraint solving* task and its possible results. The function expects as input a constraint $\mathfrak{c}$ and produces as output either the set ϕ or $\bot$. The set ϕ means that the constraint is solvable, which is denoted by $\top(\mathfrak{c})$ in the Equation. The set ϕ can either be empty or non-empty and contains the variable valuations satisfying the constraint, with $V' \subseteq V$ denoting the set of variables used in the constraint. The set ϕ is empty if variables are restricted in the constraint, but the constraint is solvable. In contrast to that the result $\bot$ denotes the absence of a result, which means that the constraint is not solvable, *i.e.* $\top(\mathfrak{c}) = false$.

In the remainder of this work the *constraint solving* task is denoted by the function **solveConstraint**$(\mathfrak{c})$. The given constraint $\mathfrak{c}$ is a first order logic predicate, where quantifiers are not considered.

Example 4.5 (Constraint Solving)
Consider the constraint $x > 0 \wedge x < 0$. Applying *constraint solving* to the constraint, such that **solveConstraint**$(x > 0 \wedge x < 0)$, results in $\perp$ as no value for variable x can be found in order to satisfy the given constraint. By contrast, for the constraint $x > 0 \wedge x < 2 \wedge y > 1 \wedge y < 3$, such that **solveConstraint**$(x > 0 \wedge x < 2 \wedge y > 1 \wedge y < 3)$, results for example[1] in the solver result $\phi = \{(x, 1), (y, 2)\}$. Hence, the second constraint is solvable. □

4.1.6. Statistical Methods

Some metrics discussed in this thesis use the statistical tools *arithmetic mean* and *standard deviation*. In the remainder of this thesis the *arithmetic mean* will be referred to by the term *mean*. Hence, some functions will be introduced which act as representatives of the corresponding mathematical formulas.

In Equation 4.10 the arithmetic mean formula is stated. The equation utilizes the set $X = \{x_0, x_1, ..., x_n\}$, which consists of n values, with $n = |X|$. The values of the set can either be an integer or a real number, such that $x_0, x_1, ..., x_n \in \mathbb{R}$. The formula is referred to in the remainder of this work by the function **mean**(X).

$$\mu = \frac{1}{|X|} \sum_{i=1}^{|X|} x_i \tag{4.10}$$

Equation 4.11 states the formula for the standard deviation which utilizes the set X and the arithmetic mean μ. In the remainder of this work the standard deviation is referred to by the function **standardDeviation**(X,μ).

$$\sigma = \sqrt{\frac{1}{|X| - 1} \sum_{i=1}^{|X|} (x_i - \mu)} \tag{4.11}$$

Moreover, in Equation 4.12 the function **cardinality**(Y) is stated, which is used in the remainder to calculate the set X, such that $X =$ **cardinality**(Y), wgich is based on the content of a given set Y. Set Y shows the characteristic that it consists of subsets, such that $Y = \{Y^0, Y^1, ..., Y^n\}$. The function counts the comprised elements for each subset Y^i, with $0 \leq i \leq n$.

$$\textbf{cardinality}(Y) = \bigcup_{i=1}^{n} |Y^i| \tag{4.12}$$

[1] other solutions are possible as well

Example 4.6 (Mean and Standard Deviation)
Let $Y = \{\{a,b,c\},\ \{d,e,f,g\},\ \{x,y\}\}$ be a set consisting of 3 subsets containing arbitrary elements. Then **cardinality**(Y) results in $X = \{3,4,2\}$. Applying Equations 4.10 and 4.11 on set X result in $\mu = 3$ and $\sigma = 0,816$. □

In the remainder of this work the mean value and its corresponding standard deviation are stated in the notion $\mu \pm \sigma$.

4.2. Syntactic and Semantic Checks

In the following, checks are introduced with respect to the syntactic and semantic quality of an ESTS model. The goal of these checks is to ensure that a model does not exhibit contradictions. The checks addressing the syntactic and semantic quality issue in terms of *inter-model communication*, *ambiguous variable definitions*, *valid guard and attribute update functions*, *detection of hidden transitions* and *non-determinism in terms of overlapping guards*.

4.2.1. Input and Output Message Consistency

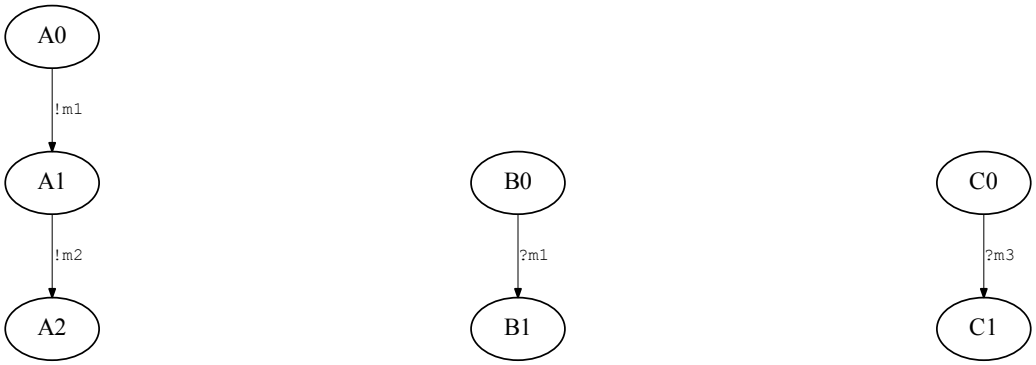
ESTSsare able to communicate with each other by means of input and output messages. The objective of the input and output messages consistency check is to detect messages which are either not sent but received or sent but not received by another ESTS. The presence of such messages is an indicator of an inconsistent system whose behavior cannot be guaranteed to be sound.

Messages which are sent but not received by another ESTS are detected by means of the function **notReceivable()** as shown in Equation 4.13. e' indicates the ESTS for which the output messages are being checked. E represents the set of all ESTS of the system including the ESTS e'. The formula removes from the output label set of ESTS e' all input labels of the united input label set of the other ESTS as far as the label exists. Thus, if an output label of e' is referenced as an input label in another ESTS, this label is removed from the output label set. Therefore, the result of the function is either the empty set if every output message of e' is receivable. Otherwise the result is a non-empty set, comprising all non-receivable messages, which indicates an inconsistency.

$$\textbf{notReceiveable}(e', E) = L_o^{e'} \setminus \bigcup_{e \in E \setminus \{e'\}} L_i^e \tag{4.13}$$

Example 4.7 (Message never received)
Consider a system E consisting of three ESTS instances e_1, e_2 and e_3 as shown in Figure 4.5, such that $E = \{e_1, e_2, e_3\}$. The output messages are decorated with an ! and input messages with a ?. Parameters have been neglected to simplify the example. This notation will also be used for all examples in this sub chapter. In this example the output messages of instance e_1 are checked for

whether they are receivable or not by calling **notReceivable**(e_1, E). ESTS e_1 only sends the messages m_1 and m_2 so that $L_o^{e'} = L_o^{e_1} = \{m_1, m_2\}$. The ESTS e_2 and e_3 receive one message each so that the input label sets are defined as $L_i^{e_2} = \{m_1\}$ and $L_i^{e_3} = \{m_3\}$, respectively. Applying the function in Equation 4.13, such that $\{m_1, m_2\} \setminus (\{m_1\} \cup \{m_3\})$, results in the set $\{m_2\}$. The resulting set indicates that message m_2 is not received by ESTS e_2 or e_3. □



(a)ESTS a (b)ESTS b (c)ESTS c

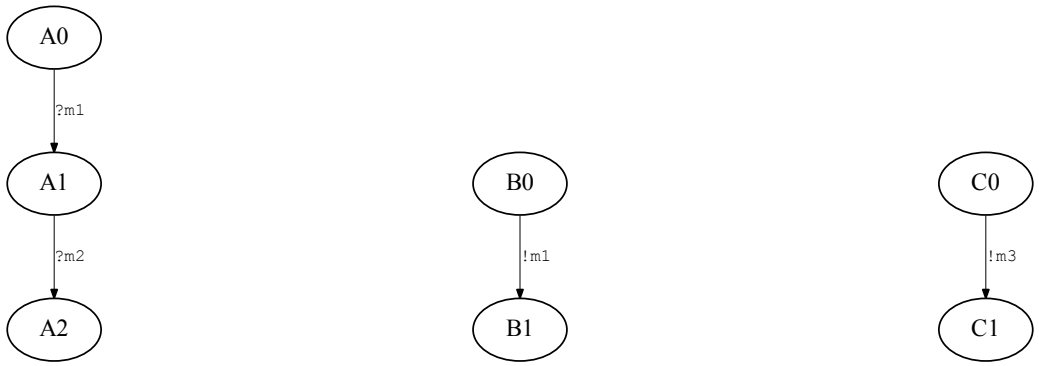
Figure 4.5.: System consisting of three ESTS models a, b and c, in which the output message m_2 of ESTS a is not received.

The detection of input messages which are not sent by another ESTS is realized in a similar way by means of the function **notSent()** as shown in Equation 4.14. e' represents the ESTS for which all input messages are checked. The difference to Equation 4.13 is that in this case the united set of output labels of all other ESTSs, except e', are removed from the input label set of e'. As in the detection of non-receivable output messages, the empty set indicates that no inconsistency has been exhibited, which means every expected input message of e' is sent by another ESTS of set $E \setminus \{e'\}$. The non-empty set indicates that an inconsistency regarding messaging exists or that the message is produced by the environment. It comprises the input message of e' which are not sent by the other ones.

$$\textbf{notSent}(e', E) = L_i^{e'} \setminus \bigcup_{e \in E \setminus \{e'\}} L_o^e \tag{4.14}$$

Example 4.8 (Message not sent)
Consider a system E consisting of three ESTS instances e_1, e_2 and e_3 as shown in Figure 4.7 such that $E = \{e_1, e_2, e_3\}$. In this example the input messages of instance e_1 are checked for whether they are sent by any other ESTS. ESTS e_1 expects to receive two input messages such that the set of input labels of e_1 is $L_i^{e_1} = \{m_1, m_2\}$. The ESTS e_2 and e_3 send one message each so that the output label sets are defined as $L_o^{e_2} = \{m_1\}$ and $L_o^{e_3} = \{m_3\}$, respectively. Applying the function **notSent(**e_1, E**)**,

such that $\{m_1, m_2\} \setminus (\{m_1\} \cup \{m_3\})$, results in the set $\{m_2\}$, which indicates that message m_2 is not sent by ESTS e_2 and e_3. □



(a)ESTS a (b)ESTS b (c)ESTS c

Figure 4.6.: System consisting of three ESTS models a, b and c, in which the input message m_2 of ESTS a is not sent by any of the other ESTSs.

4.2.2. Ambiguous Variable Definition

Ambiguous variable definition addresses the issue that a variable, either an attribute or a parameter, is defined with the same name several times in the corresponding variable scope. The scope of an attribute refers to an ESTS and the scope of a parameter to the corresponding input or output transition. Ambiguous variable definitions result in an inconsistent model and, as a further consequence, affects the readability and understandability of a model.

$$\textbf{ambiguousAttribs}(\iota) = \{(a_i, u_i), (a_j, u_j) \in \iota_0 \mid a_i = a_j \wedge i \neq j\} \tag{4.15}$$

The existence of a ambiguous attribute definitions is checked by considering the initial attribute valuation $\iota_0 \in \mathfrak{U}^A$. An attribute is assumed not to be uniquely defined if ι_0 contains more than one corresponding entry for an attribute. On that account, the function **ambiguousAttribs**(ι_0), which is defined in Equation 4.15, is applied to ι_0 in order to filter multiple attribute valuations. The equation shows that only the name of the attribute is considered, while the corresponding values and therefore the value domains are neglected. The function returns as a result the set $\iota_{multiple}$. The set is either a subset of ι_0, such that $\iota_{multiple} \subseteq \iota_0$ applies, or the empty set if no multiple definitions are present.

Example 4.9 (Ambiguous Attribute Definition)
Consider the initial attribute valuation $\iota_0 = \{(a1, 1.0), (a2, true), (a3, 2), (a1, 2.0)\}$. Applying Equation 4.15 results in $\iota_{multiple} = \{(a1, 1.0), (a1, 2.0)\}$, which comprises two entries for attribute $a1$. The result shows that the attribute definition and, as a further consequence, the initial values are ambiguous. □

$$\forall_{\varsigma \in \mathfrak{U}^{\mathbf{par}(l)}, l \in L_{io}} \mathbf{ambiguousParams}(\varsigma) \tag{4.16}$$

$$\mathbf{ambiguousParams}(\varsigma) = \{(p_i, u_i), (p_j, u_j) \in \varsigma \mid p_i = p_j \wedge i \neq j\} \tag{4.17}$$

In contrast to attributes, parameters only have transition scope. That means that a parameter can be defined with the same name at different output and input transitions, but not twice at the same transition. Thus, the parameter valuation for each transition has to be considered in order to filter ambiguous parameter definitions. Equation 4.17 state the function **ambiguousParams()**. This function is applied to each parameter valuation of a corresponding input and output transitions as shown in 4.16.

Example 4.10 (Ambiguous Parameter Definition)
Consider an ESTS which consists of an output transition $t_1 \in T$ with label $l = send$ and an input transition $t_2 \in T$ with label $l = receive$. The output transition has two corresponding parameters such that $\varsigma^{t_1} = \mathfrak{U}^{\mathbf{par}(t_1)} = \{(p_1, 0.0), (p_1, true), (p_2, true)\}$. The input only has one defined parameter such that $\varsigma^{t_2} = \mathfrak{U}^{\mathbf{par}(t_2)} = \{(p_1, true)\}$. Applying Equation 4.17 to either parameter valuations ς^{t_1} and ς^{t_2}, the following result sets are retrieved: $\mathbf{ambiguousParams}(\varsigma^{t_1}) = \{(p_1, 0.0), (p_1, true)\}$ and $\mathbf{ambiguousParams}(\varsigma^{t_2}) = \emptyset$ □

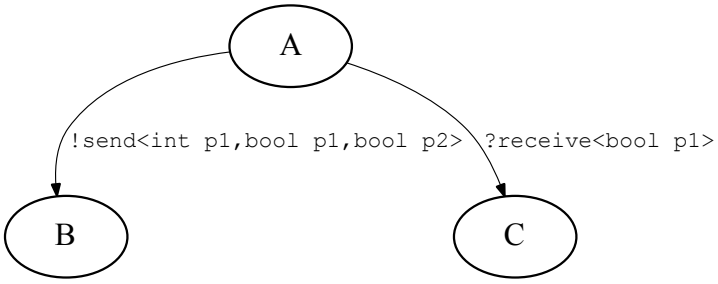


Figure 4.7.: Example of an ESTS consisting of an output and an input to depict the ambiguity of parameter definitions.

4.2.3. Validation of Guard and Attribute Update Functions

This section introduces a check which addresses the correctness of guard and attribute update function expressions. The check is responsible for proving these expressions to be syntactically correct with regard to the proper use of operators and operands in the expression. Therefore, the check validates an expression in terms of *(1) the use of invalid operands*, *(2) incompatibility of operands with respect to the operator*, *(3) existence of implicit type conversions* and *(4) whether an expression is allowed to be used in a root expression of guards and attribute update functions in general*. For the purpose of checking guards and actions the corresponding expressions are transformed to the BET structure to which the check is actually applied.

Operands / Operators	**Left**	**Right**	**Evaluated Value**
$ArithmOp \setminus \{++, --\}$	$exp \in ArithmExp \vee \mathfrak{c}, \iota, \varsigma \in \mathbb{Z} \cup \mathbb{R}$	$exp \in ArithmExp \vee \mathfrak{c}, \iota, \varsigma \in \mathbb{Z} \cup \mathbb{R}$	$expValue \in \mathbb{Z} \cup \mathbb{R}$
$ArithmOp \setminus \{+, -, *, /, \%\}$	$\iota, \varsigma \in \mathbb{Z}$		$expValue \in \mathbb{Z}$
$AssignOp$	$\iota \in \mathbb{Z} \cup \mathbb{R} \cup \mathbb{B}$	$exp \in ArithmExp \vee \mathfrak{c}, \iota, \varsigma \in \mathbb{Z} \cup \mathbb{R} \cup \mathbb{B}$	$expValue = \emptyset$
$ComparisionOp \setminus \{==, !=\}$	$exp \in ArithmExp \vee \mathfrak{c}, \iota, \varsigma \in \mathbb{Z} \cup \mathbb{R}$	$exp \in ArithmExp \vee \mathfrak{c}, \iota, \varsigma \in \mathbb{Z} \cup \mathbb{R}$	$expValue \in \mathbb{B}$
$ComparisionOp \setminus \{<, \leq, >, \geq\}$	$exp \in ArithmExp \vee \mathfrak{c}, \iota, \varsigma \in \mathbb{Z} \cup \mathbb{R} \cup \mathbb{B}$	exp $\in ArithmExp \vee \mathfrak{c}, \iota, \varsigma \in \mathbb{Z} \cup \mathbb{R} \cup \mathbb{B}$	$expValue \in \mathbb{B}$
$LogicalOp \setminus \{!\}$	$exp \in ComparisionExp \vee exp \in LogicalExp$	$exp \in ComparisionExp \vee exp \in LogicalExp$	$expValue \in \mathbb{B}$
$LogicalOp \setminus \{\&\&, \|\|\}$	$exp \in ComparisionExp$	-	$expValue \in \mathbb{B}$

Table 4.1.: Mapping of operators to valid operands.

In order to check the proper structure of an expression, syntactic rules must be in place, which indicate which types of operand is allowed for which type of operator. These constraints are depicted in Table 4.1, in which the possible left and right operands for each expression type are stated. The operands can either be another expression exp, a literal $\mathfrak{c}$, an attribute value ι or a parameter value ς. The column **Evaluated Value** states to which value domain, denoted as expected value $expValue$, an expression evaluates, once it has been executed. Considering the definition for $ArithmOp \setminus \{++, --\}$, where the left operand can be any arithmetic expression, due to definition $exp \in ArithmExp$, a literal, an attribute or a parameter value, due to definition $\mathfrak{c}, \iota, \varsigma \in \mathbb{Z} \cup \mathbb{R}$. The right operand is defined in the same way as the left operand. When executing the expression, it evaluates to a value which is either an integer or a real number depending on the type of the operands. In the case of unary operators, such as "++", "−−" and "!", only the left operand column is considered. An expression which does not conform to the definitions in Table 4.1 is considered syntactically incorrect.

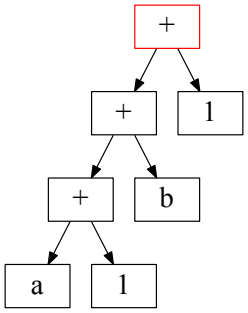


Figure 4.8.: Binary Expression Tree of expression $a + 1 + b + 1$.

Example 4.11 (Syntactically correct and incorrect expressions)
Considering the binary arithmetic expression $((a + 1) + b) + 1$, where a and b are arbitrarily chosen attributes of the integer value domain. The nested expression $a + 1$ conforms to the definition in Table 4.1, since the left operand is $\iota \in \mathbb{Z}$ and the right operand complies to $\mathfrak{c} \in \mathbb{Z}$. Afterwards the nested expression $((a + 1) + b)$ is considered, in which $a + 1$ functions as the left operand. This expression is syntactically correct as well, since the left operand is an arithmetic expression $exp \in ArithmExp$, whereas the right operand conforms to the definition $\iota \in \mathbb{Z}$. Finally, the overall expression $((a + 1) + b) + 1$, with the left operand $((a + 1) + b)$ and the right operand 1, is considered correct. This is the case as the left operand is an arithmetic expression and the right operand a literal. Thus, either operands comply to the syntactic definition of arithmetic operators.

By contrast, the expression $(a < 1) + b$ is syntactically not correct. The nested expression $a < 1$ itself is a correct comparison expression as it complies to the definition of $ComparisionOp \setminus \{++, --\}$. Nevertheless, using this expression in combination with an arithmetic operator as the left operand does not obey the syntactic rules for arithmetic expressions. □

Once the syntactic correctness of an expression has been ensured, the binary arithmetic operators, defined as the set $ArithmOp \setminus \{++, --\}$, and the comparison operators of set $ComparisionOp$ are checked with respect to their compatibility with the operands involved and their respective value domains. Thus, an expression can be considered erroneous although it conforms to the syntactic definitions in Table 4.1. Table 4.2 shows the compatibility constraints applied to binary arithmetic and comparison expressions. Basically the table states two cases where operands do not match and therefore lead to a check violation. The table shows for which operator and expression which violation is detected. The first case considers expressions with numeric operands, so that an operand value corresponds either to a literal, attribute valuation or parameter valuation. When the operands do not match an *implicit type conversion* has been detected. The second case addresses those expressions where Boolean operands are allowed as well. If at least one operand is of the Boolean type the other operand must be of the same type, otherwise an *incompatibility* of the operands is detected.

	Expression	Type of Violation
$op \in \{==, !=, <, \leq, >, \geq =, +, -, /, *, \%\}$	(a) $\mathbb{R}$ op $\mathbb{Z}$ (b) $\mathbb{Z}$ op $\mathbb{R}$	Implicit Type Conversion
$op \in \{==, !=, =\}$	(a) $\{\mathbb{R}, \mathbb{Z}\}$ op $\mathbb{B}$ (b) $\mathbb{B}$ op $\{\mathbb{R}, \mathbb{Z}\}$	Incompatible Operands

Table 4.2.: Compatibility of operands in terms of their value domains.

Finally, the guard and action expressions are checked for whether an arithmetic, assignment, comparison or logical expression can be used as root expression exp_{root}. The root expression exp_{root} of BET from Figure 4.8 is $exp_{root} = \langle +, exp, 1 \rangle$, the operator is "+", the left operand is an expression and the right operand a literal representing an integer value. The overall type of the expression $((a+1)+b)+1$ is determined by the operator type of the BET's root expression, which is highlighted in red in the figure. Since the operator of the root expression is "+" and thus the operator is an arithmetic operator, the overall expression belongs to an arithmetic expression, such that $exp_{root} \in ArithmExp$. Table 4.3 states the root expression constraints, where ✓ indicates the possibility to act as root expression. Expression types which are not present in the table at all are not allowed.

Expression Type	Guard	Action
$ArithmExp$		
$AssignExp$		✓
$ComparisionExp$	✓	
$LogicalExp$	✓	

Table 4.3.: Allowed root expressions for guards and actions.

Once all required prerequisites for determining a correct expression have been defined, checking of an expression is carried out by means of Algorithm 3. Therefore, the method *CHECK_EXPRESSION()* is called with the root expression exp_{root} of the BET. The root expression either corresponds to a guard or to an action. Thus, the transformation to the BET structure is carried out beforehand and therefore either by calling $\mathbf{bet}(\varphi)$ or $\mathbf{bet}(\rho)$.

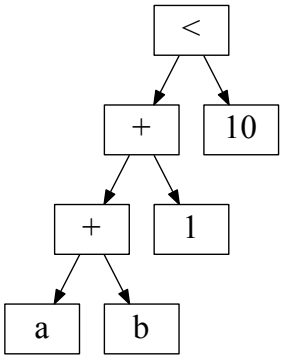
Basically the method consists the following main steps:

1. Check the left operand type opd_l of the given expression exp.
   a) If the left operand is either a variable valuation or a literal, the corresponding value domain is determined.
   b) If the left operand is an expression, the method *CHECK_EXPRESSION()* is called recursively with the left operand as argument. This step denotes the traversal of the left sub-tree in terms of the BET structure. The traversal results in the determination of the overall value domain of the left sub-tree.
2. Check the right operand opd_r type of the given expression exp
   a) If the right operand is either a variable valuation or a literal, the corresponding value domain will be determined.

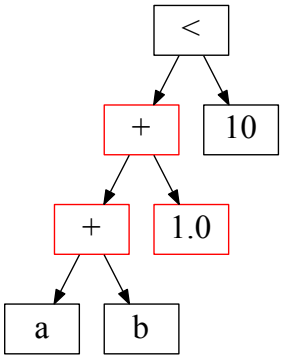
b) If the right operand is an expression, the method *CHECK_EXPRESSION()* is called recursively with the left operand as argument. This step denotes the traversal of the right sub-tree in terms of the BET structure. The traversal results in the determination of the overall value domain of the right sub-tree.

3. Call function *VALIDATE_EXPRESSION()* with the current expression exp and the determined value domains for the left and right sub-tree. The function validates the given expression with respect to the defined rules. The algorithm stops if the function returns result *INVALID*.
4. Return the overall expression value domain which depends on the left and right operand. If the expression operator belongs to a comparison or logical operator the value domain depends on the operator, since for these expression always the Boolean value domain is returned. On the other side, the determined value domain depending on the value domains of the left and right operands is returned.

The algorithm validates the BET bottom-up by calling the method *CHECK_EXPRESSION()* recursively as long as the left and/or right operands correspond to an expression. The reason is that only those operands, which are associated to an attribute, parameter or literal, correspond to leaf nodes in the context of an BET. The value domains of these operands define the overall value domain of the corresponding expression. The overall value domain of the expression defines the value domain of the parent expression if one exists. Thus, the tree and in further consequences each sub-tree is traversed until an expression is processed whose left or right operand corresponds to an attribute, parameter or literal. In that case the corresponding value domain is determined and passed to the previous recursive step. Once an operator of a certain recursive traversing step and the value domains of its operands are known, the expression is validated. In the event the validation is successful, the overall expression type is passed to the previous recursive step, which means to the parent expression for which this expression acts as operand. Otherwise a violation is reported and traversing of the BET is aborted.



(a)BET of expression $(((a + b) + 1) < 10$.



(b)BET of expression $(((a + b) + 1.0) < 10$.

Figure 4.9.: Example of guard expressions in the BET structure.

Algorithm 3 Validation of Guards and Action Expressions

```
1:  function CHECK_EXPRESSION(exp) → 𝔘
2:      if exp.opd_l ∈ 𝔘^V ∨ IS_LITERAL(exp.opd_l) then
3:          𝔘_l ← DETERMINE_DOMAIN(exp.opd_l);
4:      else
5:          𝔘_l ← CHECK_EXPRESSION(exp.opd_l);
6:      end if
7:
8:      if exp.opd_r ∈ 𝔘^V ∨ IS_LITERAL(exp.opd_r) then
9:          𝔘_r ← DETERMINE_DOMAIN(exp.opd_r);
10:     else
11:         𝔘_r ← CHECK_EXPRESSION(exp.opd_r);
12:     end if
13:
14:     result ← VALIDATE_EXPRESSION(exp, 𝔘_l, 𝔘_r);
15:     if result == INVALID then
16:         "Abort algorithm!";
17:     end if
18:
19:     if exp.op ∈ ComparisionOp ∪ LogicalOp then
20:         return 𝔹;
21:     end if
22:
23:     return 𝔘_l;
24: end function
```

Example 4.12 (Expression Validation)

Consider the expression $((a+b)+1) < 10$ whose BET representation is shown in Figure 4.9a, with a and b are assumed to be arbitrary chosen attributes of the integer domain. The root expression of the BET is $exp_{root} = \langle <, exp', 10\rangle$. According to the algorithm, in each recursive step the left sub tree is traversed first. Thus, *CHECK_EXPRESSION()* is called the first time with $exp_{root}.opd_l$ as argument, with the left operand corresponding to $exp' = \langle +, exp'', 1\rangle$. Since the left operand of exp' is an expression, *CHECK_EXPRESSION()* is called again recursively with exp'' as argument, with $exp'' = \langle +, a, b\rangle$. Due to the fact that the left operand of exp'' corresponds to an attribute no further recursive-step is required. Next, the right sub-tree of expression exp'' will be considered, but no recursive step is needed since the right operand corresponds to an attribute. Due to the fact that both operands of exp'' correspond to attributes the value domain of these operands are determined as integers. Thus, $\mathfrak{U}''_l = \mathfrak{U}''_r = \mathbb{Z}$ applies in this recursive step which validates the sub-expression $a+b$. Validating $a+b$ results in no violation and therefore this recursive step returns with the integer value domain as result which represents the overall value domain of $a+b$ and exp'', respectively. This overall domain value is passed to the previous, not yet finished, recursive step which processes the expression $(a+b)+1$. Thus, the domain value of $exp'.opd_l$ is now known, such that $\mathfrak{U}'_l = \mathbb{Z}$. Since the left operand is evaluated the value domain of the right operand has to be determined next. The right operand is an attribute of the integer value domain, such that $\mathfrak{U}'_r = \mathbb{Z}$. The validation of the exp' causes no violation and the overall domain value $\mathbb{Z}$ is returned. Finally, the entire expression exp_{root} is validated, due to the fact that the algorithm is now back at the first call of *CHECK_EXPRESSION()* . The left operand value domain has been evaluated as $\mathfrak{U}_l = \mathbb{Z}$. As the right operand corresponds to the literal 10 of the integer value domain, the value domain of the left operand is $\mathfrak{U}_r = \mathbb{Z}$. The call of *VALIDATE_EXPRESSION()* for expression exp_{root} leads to no violation. Therefore, the entire expression $((a+b)+1) < 10$ is valid.

Figure 4.9b depicts a BET of an invalid expression. Validation is carried out exactly in the same way as for the previous example. The validation differs only in the recursive step which validates expression $(a + b) + 1.0$. Since in this recursive step, with $exp' = \langle +, exp'', 1.0 \rangle$, the corresponding value domains are $\mathfrak{U}'_{left} = \mathbb{Z}$ and $\mathfrak{U}'_{right} = \mathbb{R}$. Due to the determined left and right operand value domains a *implicit type conversion* is detected according to the definition in Table 4.2. □

4.2.4. Detection of Hidden Transitions

The goal of this check is the detection of so called *hidden transitions*. This type of check during a static analysis has already been proposed in [16] by Schwarzl *et al.* in the context of UML state charts. Basically a hidden transition is a transition which cannot be executed since a guard of a higher prioritized transition, corresponding to the same label, exists that covers the same or a larger value range than its own guard. As a consequence the hidden transition will never be executed. A hidden transition violation can only occur within the outgoing transition set of a particular state. Thus, the check is applied to each outgoing transition set of each state of an ESTS.

Let $t^a_{l_1}, t^b_{l_2}$ be two transitions with priorities a and b, with $a > b$, and with labels l_1 and l_2, respectively. For the labels must hold that $l_1 = l_2$. Then both transitions correspond to the outgoing transition set of the same state, such that $t^a_{l_1}, t^b_{l_2} \in \textbf{stateOutTrans}(s)$, with s denoting the source state of both transitions. In the following, the declaration of the labels l_1 and l_2 are neglected and thus the transitions are denoted as t^a and t^b, and it is assumed that the labels are equal.

Let $\textbf{valueRange}(t^a.\varphi)^V$ be the value range over the variables of set V which is covered by guard φ of transition t^a. And let $\textbf{valueRange}(t^b.\varphi)^V$ be the value range over the variable set V which is covered by the guard of transition t^b. Since $a > b$ applies, t^a shows a higher priority than transition t^b. Moreover, it is assumed that both guards depend only on the variable x, such that $V' = \{x\}$ and $t^a.\varphi, t^b.\varphi \in \mathfrak{F}(V')$, with $\mathfrak{F}(V') \subseteq \mathfrak{F}(V)$. This assumption is being made only due to convenience purposes in order to simplify the following explanations.

In order to prove that transition t^a hides transition t^b, it has to be proven that the former transition guard entirely covers the value range of the latter transition guard. Thus, $\textbf{valueRange}(t^b.\varphi)^{V'} \setminus \textbf{valueRange}(t^a.\varphi)^{V'} = \emptyset$ must be proven. Due to the fact that determining the entire value range of guards are a tedious undertaking, an alternative approach is required. On that account, this check uses *constraint solving*.

The aim of *constraint solving* is to find a variable valuation for the involved variables, in this very case for x, which proves that the entire value range of $t^b.\varphi$ is covered by $t^a.\varphi$. Basically for variable x, a value y should be determined which complies with $y \notin \textbf{valueRange}(t^a.\varphi)^{V'} \wedge y \in \textbf{valueRange}(t^b.\varphi)^{V'}$. In the context of *constraint solving* a valuation for x has to be determined which solves the constraint $\mathfrak{c} = !(t^a.\varphi) \wedge t^b.\varphi$. While the first condition of the constraint denotes a value for x, which is not in the value range of the guard of transition t^a, the second condition depicts the case that the value of x must be in the value range of the guard of transition t^b. In terms of *constraint solving* a solution for the constraint, such that $\textbf{constraintSolving}(\mathfrak{c}) = \phi$, denotes that not the total

value range is being covered. By contrast no result, so that **constraintSolving(c)** = $\perp$, depicts a hidden transition violation.

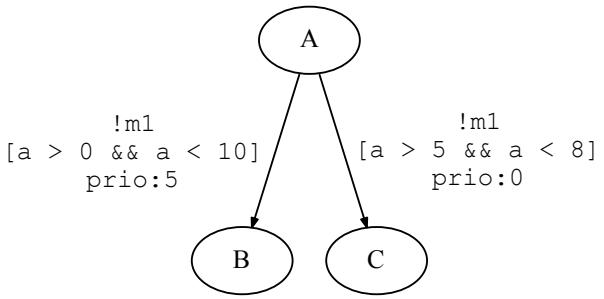


Figure 4.10.: Simple example depicting the transition hiding issue.

Example 4.13 (Hidden Transition)
Consider Figure 4.10 which depicts state A with two outgoing output transitions t^5, t^0, with t^5 corresponding to $A \xrightarrow{!m1} B$ which is an input transition with priority 5, and t^0 corresponding to $A \xrightarrow{!m1} C$ which is an input transition with priority 0. Both transitions are outgoing transitions of state A, such that $t^5, t^0 \in$ **stateOutTrans**(A). Moreover, transition t^5 shows the higher priority than transition t^0. The two guards only depend on a single attribute named a, such that the variable set $V' = \{a\}$. Thus, the corresponding value ranges are **valueRange**$(t^5.\varphi)^{V'} = \{(a, \{1,2,3,4,5,6,7,8,9\})\}$ and **valueRange**$(t^0.\varphi)^{V'} = \{(a, \{6,7\})\}$. As stated previously, a hidden transition exists if the higher prioritized transition covers the same or a larger value range of the lower prioritized transition. Applying *constraint solving* to the constraint $\mathfrak{c} = (a > 0 \;\&\&\; a < 10) \;\&\&\; (a > 5 \;\&\&\; a < 8)$ shows that the constraint is not solvable and thus the result is **constraintSolving**$(\mathfrak{c}) = \perp$. □

As stated in [27] the priority of a transition defines the execution order if multiple enabled transitions, corresponding to the same label, exist. Thus, only transitions with the same label can hide each other. For this reason the transitions must be grouped according to their label.

Let $PRIO = \{p_x, p_{x+1}, ..., p_{x+n}\}$, with $p_x > p_{x+1} > ... > p_{x+n}$, be a set of all priorities assigned to the transitions of the outgoing transition set. Then $T_y^* = \{T_y^{p_x}, T_y^{p_{x+1}}, ..., T_y^{p_{x+n}}\}$ is an ordered list of transition sets, each set corresponding to a certain priority, with $p_x, p_{x+1}, ..., p_{x+n} \in PRIO$ and therefore denoting a grouping by priorities. The sets comprise transitions of type y, with $y \in \{o_l,\ \gamma,\ \tau,\ d_l,\ i_l\}$, with o_l denoting transitions of type output corresponding to label l, γ of type completion, τ of type unobservable, d_l of type delay with the timeout term l and i_l transitions of type input corresponding to label l. This fact depicts a grouping according to the above defined transition type constraints. Thus, the sets in dependence of the transition types are defined as follows:

- The set $T_{d_l}^p = \{t \in$ **stateOutTrans**$(s) \mid t \in T_d \wedge l \in$ **timeoutTerm**$(s) \wedge t.p = p\}$ denotes all delay transitions of the outgoing transition set of state s which correspond to priority p and to

the label l which corresponds to the timeout term.

- The set $T_\gamma^p = \{t \in \textbf{stateOutTrans}(s) \mid t \in T_\gamma \wedge t.p = p\}$ denotes all completion transitions of the outgoing transition set of state s which correspond to priority p.
- The set $T_\tau^p = \{t \in \textbf{stateOutTrans}(s) \mid t \in T_\tau \wedge t.p = p\}$ denotes all unobservable transitions of the outgoing transition set of state s which correspond to priority p.
- The set $T_{o_l}^p = \{t \in \textbf{stateOutTrans}(s) \mid t \in T_o \wedge l \in \textbf{outputMessages}(s) \wedge t.p = p\}$ denotes all output transitions of the outgoing transition set of state s which correspond to priority p and to message l.
- The set $T_{i_l}^p = \{t \in \textbf{stateOutTrans}(s) \mid t \in T_i \wedge l \in \textbf{inputMessages}(s) \wedge t.p = p\}$ denotes all input transitions of the outgoing transition set of state s corresponding to priority p and to message l.

The list T_y^* shows a descending ordering based on the priority. Additionally the set $\overline{T}$ is defined containing all outgoing transitions grouped by type and priority, such that $\overline{T} = T_\gamma^* \cup T_\tau^* \bigcup_{l \in \textbf{inputMessages}(s)} T_{i_l}^* \bigcup_{l \in \textbf{outputMessages}(s)} T_{o_l}^* \bigcup_{l \in \textbf{timeoutTerm}(s)} T_{d_l}^*$.

Algorithm 4 Detection of Hidden Transitions

```
1:  procedure DETECT_HIDDEN_TRANSITIONS(T̄)
2:      for all transGroup ∈ T̄ do
3:          transGroupList ← TO_LIST(transGroup);
4:          for i ← 1, transGroupList.SIZE() do
5:              for j ← 2, transGroupList.SIZE() do
6:                  highPrioGroup ← transGroupList.GET(i);
7:                  lowPrioGroup ← transGroupList.GET(j);
8:                  CHECK_GROUPS(highPrioGroup, lowPrioGroup);
9:              end for
10:         end for
11:     end for
12: end procedure
13:
14: procedure CHECK_GROUPS(highPrioGroup, lowPrioGroup)
15:     for all t ∈ highPrioGroup do
16:         for all t′ ∈ lowPrioGroup do
17:             result ← ⊥;
18:             c ←!(t.φ) ∧ t′.φ;
19:             result ← solveConstraint(c);
20:             if result == ⊥ then
21:                 "Transition t hides transition t′";
22:             end if
23:         end for
24:     end for
25: end procedure
```

The *constraint solving* task is applied to each possible pair of high and low prioritized transitions of the list T_y^*. On this account all transitions of set $T_y^{p_x}$ are checked for whether they hide any transition of sets the $T_y^{p_{x+1}}, ..., T_y^{p_{x+n}}$. Thus, transitions of a set corresponding to a certain priority p_x are solely

checked for whether the can hide transitions of sets corresponding to a lower priority p_{x+1} ... p_{x+n}, with $p_x > p_{x+1} > \ ... \ > p_{x+n}$.

Algorithm 4 depicts the algorithm in order to detect potential hidden transitions at a single state. The defined procedure *DETECT_HIDDEN_TRANSITIONS* expects as input the set $\overline{T}$. Hence, the outgoing transition set of the state is already grouped. For each transition group $transGroup \in \overline{T}$, the comprised transitions sets are checked pairwise. This fact is depicted in the algorithm with the statements $highPrioGroup \leftarrow transGroupList.GET(i)$ and $lowPrioGroup \leftarrow transGroupList.GET(j)$, where *highPrioGroup* corresponds to a set of transitions of a higher priority than the transitions of set *lowPrioGroup*. The actual check is carried out in procedure *CHECK_GROUPS*, where for all possible pairs of high and low prioritized transitions *constraint solving* is applied.

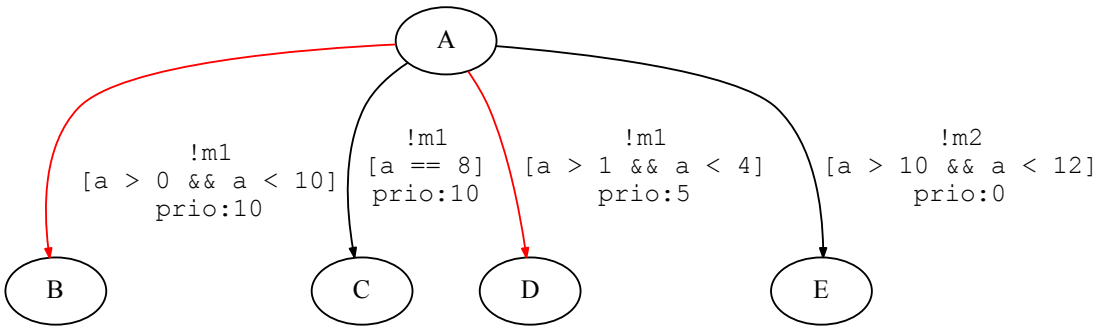


Figure 4.11.: Example for illustrating the application of Algorithm 4.

Example 4.14 (Detection of Hidden Transitions)
Consider Figure 4.11 illustrating state A which has four outgoing transitions, all of them of type output. Based on that, the set $\overline{T}$ contains two transition sets, in particular $T^*_{o_{m1}}$ and $T^*_{o_{m2}}$, with $T^*_{o_{m1}} = \{T^{10}_{o_{m1}}, T^5_{o_{m1}}\}$ and $T^*_{o_{m2}} = \{T^0_{o_{m2}}\}$. Furthermore, $T^{10}_{o_{m1}} = \{(A \xrightarrow{!m1} B), (A \xrightarrow{!m1} C)\}$, $T^5_{o_{m1}} = \{(A \xrightarrow{!m1} D)\}$ and $T^0_{o_{m2}} = \{(A \xrightarrow{!m2} E)\}$.

Applying Algorithm 4 the following pairwise *constraint solving* task is executed on the transition groups $T^{10}_{o_{m1}}$ and $T^5_{o_{m1}}$.

1. Solving the constraint "$!(a > 0 \ \&\& \ a < 10) \ \&\& \ (a > 1 \ \&\& \ a < 4)$" of transitions $A \xrightarrow{!m1} B$ and $A \xrightarrow{!m1} D$, respectively, results in no solution, which indicates that the low prioritized transition **is hidden**.
2. Solving the constraint "$!(A == 8) \ \&\& \ (a > 1 \ \&\& \ a < 4)$" of transitions $A \xrightarrow{!m1} C$ and $A \xrightarrow{!m1} D$, respectively, results in $\phi = \{(a, val)\}$ with $val \in \{2, 3\}$.

Thus, the algorithm detects a single hidden transition since transition $A \xrightarrow{!m1} B$ hides transition $A \xrightarrow{!m1} D$. No *constraint solving* is carried out for group $T^0_{o_{m2}}$ since no other group exists which corresponds to output transitions of higher priority and message $m2$. □

4.2.5. Non-determinism in Terms of Overlapping Guards

This check validates an ESTS in terms of non-deterministic behavior. The issue is addressed with respect to overlapping guards which can occur when a state has multiple outgoing transitions. Overlapping guards denote a certain variable valuation for the variables of the involved guards so that more than one guard evaluates to *true*. As a result the corresponding transitions are enabled and therefore executed at the same time. In Figure 4.13 seven non-determinism cases are depicted, which are detected by this check.

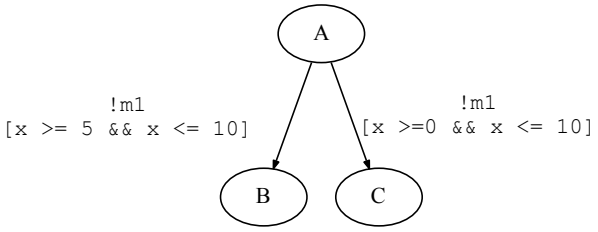


Figure 4.12.: Simple example of overlapping guards.

Example 4.15 (Simple example of overlapping guards)
Figure 4.12 depicts a simple ESTS consisting of the states A, B and C. State A has two outgoing transitions, both corresponding to an output transition in order to send message $m1$. Thus, the ESTS has the transition $(A \xrightarrow{!m1[x \geq 5 \,\&\&\, x \leq 10]} B)$ and $(A \xrightarrow{!m1[x \geq 0 \,\&\&\, x \leq 10]} C)$. In order to ensure that state A does not exhibit a non-deterministic behavior, the guards of both transitions have to be verified whether they have an overlapping value range or not. The validation of both guards "$x \geq 5$ && $x \leq 10$" and "$x \geq 0$ && $x \leq 10$" shows that both guards evaluate to *true* for $x \in [5;10]$. Thus, both transitions are enabled which leads to a non-determinism behavior. □

(1) The first case is depicted by Figure 4.13a which shows an unobservable transition. Actually this case represents an exception since its detection is not carried out by checking the guards. The reason is that the existence of an unobservable transition is already an indicator for non-determinism. Therefore, as soon as a state has an outgoing transition of type unobservable a potential non-determinism situation is detected. Thus, the detection of this non-determinism case is not covered by this check.

(2) The second case, which is illustrated in Figure 4.13b, is caused by two input transitions of same priority, which are enabled at the same time. This non-determinism issue would lead to two different states. Thus, the system cannot decide to which state it should proceed. Both input transitions must be equivalent in terms of their corresponding message. Meaning that for an arbitrary pair of input transitions $t, t' \in T_i$ applies that $t.l = t'.l$.

(3) The third case is caused by a pair of an input and output transition, as depicted in Figure 4.13c,

which are both enabled. Thus, the system cannot decide whether an output should be produced or it should be waited for the input.

(4) The non-determinism example shown in Figure 4.13d denotes the issue if two output transitions are being enabled at the same time. Hence, the system cannot decide which output should be produced. In the event that the corresponding messages of the output transitions are equal, such that $t.l = t'.l$ with $t, t' \in T_o$, the priority of the transitions must be equal as well. Thus, in this case non-determinism within a set of output transitions corresponding to the same message and priority can occur. In contrast to that also non-determinism within a set of output transitions with different priority can occur, but only if their corresponding messages are not equal.

(5) The fifth case, depicted in Figure 4.13e, denotes non-determinism caused by two completion transitions of same priority which are enabled at the same time. Once again the system cannot decide in such a situation to which state it should proceed.

(6) The next non-determinism scenario is related to a pair of enabled output and completion transition, which is illustrated in Figure 4.13f. In this case the system cannot decide whether the output should be produced or not.

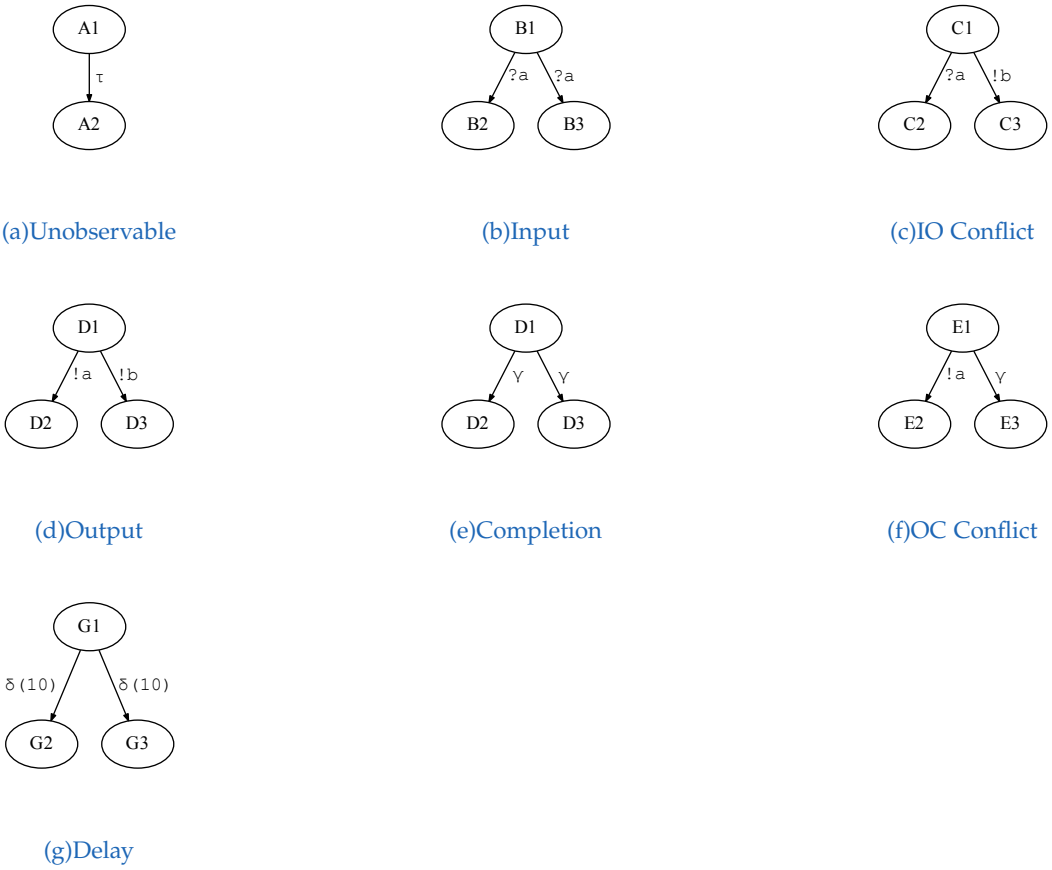
(7) The last non-determinism case concerns delay transitions as shown in Figure 4.13g, where each of the delay transition corresponds to another timing group. If the timeout terms are identical in terms of their values, both delay transitions are executed at the same time which leads to a non-determinism behavior. Moreover, the transitions must have the same priority.

Basically the algorithm to detect overlapping guards consists of two steps:

1. Group transitions according to the previously defined non-determinism cases.
2. Pairwise constraint solving of conjugated guards of each group by considering the comprised transition types.

The grouping step is applied to each outgoing transition set **stateOutTrans**(s), which is defined in Equation 3.1, of each state s of an ESTS , with $s \in E$. The goal is that the transitions are compared with respect to the prior described non-determinism cases. Thus, the following groups are introduced:

- $\forall_{m \in \textbf{inputMessages}(s)} \tilde{T}^{p}_{i_{(m)}} = \{t \in \textbf{stateOutTrans}(s) \mid t \in T_i \wedge t.l = m \wedge t.p = p\}$, with $\tilde{T}^{p}_{i_{(m)}}$ is a set containing all input transitions which correspond to priority p of the outgoing transition set of state s. All input transitions correspond to the same message m. An input transition set is created for each input message $m \in \textbf{inputMessages}(s)$ of state s. This groups are related to non-determinism case (2).
- The group $\tilde{T}^{*}_{io} = \{t \in \textbf{stateOutTrans}(s) \mid t \in T_i \cup T_o\}$ is related to case (3) and contains all input and output transitions of the outgoing transition set of state s.
- $\forall_{m \in \textbf{outputMessages}(s)} \tilde{T}^{p}_{o_{(m)}} = \{t \in \textbf{stateOutTrans}(s) \mid t \in T_o \wedge t.l = m \wedge t.p = p\}$, with $\tilde{T}^{p}_{o_{(m)}}$ is a set containing all output transitions which correspond to priority p of the outgoing transition set of state s. All output transitions correspond to the same message m. An output transition set is being created for each output message $m \in \textbf{outputMessages}(s)$ of state s. These groups are related to non-determinism case (4).



(a)Unobservable (b)Input (c)IO Conflict

(d)Output (e)Completion (f)OC Conflict

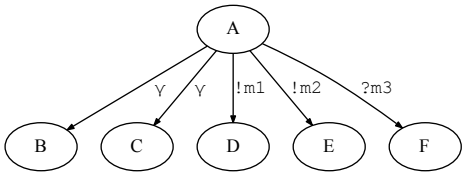
(g)Delay

Figure 4.13.: Non-Determinism cases in terms of overlapping guards. (Adapted from [27])

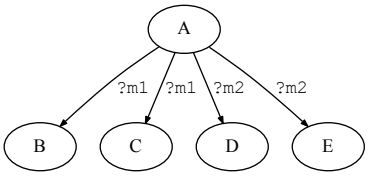
- The group $\tilde{T}_o^* = \{t \in \textbf{stateOutTrans}(s) \mid t \in T_o\}$ is related to case (4) as well, and contains all output transitions corresponding to the outgoing transition set of state s.
- The group $\tilde{T}_\gamma^p = \{t \in \textbf{stateOutTrans}(s) \mid t \in T_\gamma \wedge t.p = p\}$ is related to case (5) and contains all completion transitions of priority p which correspond to the outgoing transition set of state s.
- The group $\tilde{T}_{o\gamma}^* = \{t \in \textbf{stateOutTrans}(s) \mid t \in T_o \cup T_\gamma \wedge t.p = p\}$ is related to case (6) and contains all input and output transitions which correspond to priority p of the outgoing transition set of state s.
- $\forall_{l \in \textbf{timeoutTerm}(s)} \tilde{T}_{d_{(l)}}^p = \{t, t' \in \textbf{stateOutTrans}(s) \mid (t \in T_d^g) \wedge (t' \in T_d^{g'}) \wedge (t.p = p) \wedge (t.l = t'.l) \wedge (t.l = l) \wedge (t'.l = l) \wedge (t'.p = p) \wedge (g \neq g')\}$ is related to case (7) and contains only delay transitions, corresponding to the same timeout term $''l$, of different timing groups such that $g \neq g'$. A delay transition group is created for each timeout term $l \in \textbf{timeoutTerm}(s)$ of state s.

While the groups $\tilde{T}_{io}^*$ and $\tilde{T}_{o\gamma}^*$ exist once, the groups $\tilde{T}_\gamma^p$, $\tilde{T}_{o_{(m)}}^p$, $\tilde{T}_o^*$, $\tilde{T}_{d_{(l)}}^p$ and $\tilde{T}_{i_{(m)}}^p$ can exist several times. The number of these groups depend on the one hand from the distinct priorities and on the other hand from the distinct input and output messages, and timeout terms of the outgoing

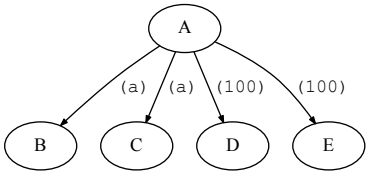
transition set of a state. No group are created for the first non-determinism case since unobservable transitions are not considered in terms of checking for overlapping guards.



(a)Example for demonstrating grouping in terms of cases (3), (4), (5) and (6).



(b)Example for demonstrating grouping for case (2).



(c)Example for demonstrating grouping for case (7).

Figure 4.14.: Examples of outgoing transition sets in order to demonstrate grouping.

Example 4.16 (Grouping)
Consider Figure 4.14 depicting three examples of outgoing transition sets for state A. Assuming that all of the transitions exhibit the same priority the following transition groups are created.

For the example depicted in Figure 4.14a the following groups are created:

- $\tilde{T}^p_{i_{(?m3)}} = \{(A \xrightarrow{?m3} F)\}$
- $\tilde{T}^*_{io} = \{(A \xrightarrow{!m1} D), (A \xrightarrow{!m2} E), (A \xrightarrow{?m3} F)\}$
- $\tilde{T}^p_{o_{(!m1)}} = \{(A \xrightarrow{!m1} D)\}$
- $\tilde{T}^p_{o_{(!m2)}} = \{(A \xrightarrow{!m2} E)\}$
- $\tilde{T}^*_{o} = \{(A \xrightarrow{!m1} D), (A \xrightarrow{!m2} E)\}$
- $\tilde{T}^p_{\gamma} = \{(A \xrightarrow{\gamma} B), (A \xrightarrow{\gamma} C)\}$
- $\tilde{T}^*_{o\gamma} = \{(A \xrightarrow{!m1} D), (A \xrightarrow{!m2} E), (A \xrightarrow{\gamma} B), (A \xrightarrow{\gamma} C)\}$
- $\tilde{T}^p_{d_{(l)}} = \emptyset$

For the example illustrated in Figure 4.14b two input transition groups are created, $\tilde{T}^p_{i_{(m1)}} = \{(A \xrightarrow{?m1} B), (A \xrightarrow{?m1} C)\}$ and $\tilde{T}^p_{i_{(m2)}} = \{(A \xrightarrow{?m2} B), (A \xrightarrow{?m2} C)\}$. The last example, depicted in Figure 4.14c, addresses the grouping of delay transitions. As no other transitions exist in the outgoing transition set, the remaining groups are empty. Thus, only the groups $\tilde{T}^p_{d_{(a)}} = \{(A \xrightarrow{a} B), (A \xrightarrow{a} C)\}$ and

$\tilde{T}^{p}_{d_{(100)}} = \{(A \xrightarrow{100} B), (A \xrightarrow{100} C)\}$ are created.

□

The detection whether a pair of transition guards overlap is achieved by applying *constraint solving* to a conjugated pair of transitions guards. The conjugated guard expressions are passed to the *constraint solver* which tries to find a solution for the expression so that the expression evaluates to *true*, which indicates that the expression is solvable. Thus, for an arbitrary pair of transitions t, t' of a transition group, the conjugated guard $\hat{\varphi} = t.\varphi \wedge t'.\varphi$ is solved. Thus, $\hat{\varphi}$ denotes the constraint $\mathfrak{c}$ to be solved, such that $\mathfrak{c} = \hat{\varphi}$.

The delay transition group $\tilde{T}^{p}_{d_{(l)}}$ constitutes an exception in respect of overlapping guards detection. For these transitions also the timeout term is considered due to the fact that delay transitions are executed after the delay has been elapsed. Thus, non-determinism caused by delay transitions can only occur if the timeout terms of more than one enabled delay transitions are equal. In other words multiple delay transitions are executed at the same time. In the example of Figure 4.14c this would lead to the situation that the system cannot decide whether it should proceed to state B or to state C. Therefore, in the case of delay transitions not only their guards are conjugated, but also an additional condition is added which compares the corresponding timeout terms. The reason lies in the fact that the related guards can restrict and therefore influence the value of the timeout term. Thus, the constraint to which *constraint solving* is applied is $\hat{\varphi} \wedge t.l = t'.l$, with $t, t' \in \tilde{T}^{p}_{d_{(l)}}$ are arbitrary delay transitions of the transition group $\tilde{T}^{p}_{d_{(l)}}$ which is related to priority p. In terms of the overlapping guards check, no solution, such that the solver result is equal to $\perp$, indicates that the two involved guards do not overlap.

The prior described *constraint solving* task is applied to each transition pair within a group. When performing pairwise *constraint solving* three cases are distinguished:

1. For each group which only consists of a single type of transition, like $\tilde{T}^{p}_{i_{(m)}}$, $\tilde{T}^{p}_{o_{(m)}}$, $\tilde{T}^{p}_{d_{(l)}}$ and $\tilde{T}^{p}_{\gamma}$, *constraint solving* is applied to each pair of transition. Thus, *constraint solving* is carried out $(n * (n-1)/2)$ times, where n denotes the number of transitions comprised by the group.
2. For each group which comprises two types of transitions, like $\tilde{T}^{*}_{io}$ and $\tilde{T}^{*}_{o\gamma}$, only transition pairs of different types are solved. So that for each transition pair $t, t' \in \tilde{T}^{*}_{io}$ must comply to $(t \in T_i \wedge t' \in T_o) \vee (t \in T_o \wedge t' \in T_i)$. This also applies to group $\tilde{T}^{*}_{o\gamma}$ with the difference that other transition types are involved, so that the transition pair $t, t' \in \tilde{T}^{*}_{o\gamma}$ complies to $(t \in T_o \wedge t' \in T_\gamma) \vee (t \in T_\gamma \wedge t' \in T_o)$ Thus, *constraint solving* is carried out $(n * m)$ times, where n denotes the number of transitions of the first transition type and m the number of transitions of the second transition type.
3. The group $\tilde{T}^{*}_{o}$ contains all output transitions of the outgoing transition set of a state. The *constraint solving* task is applied to each pair of transition with different output messages assigned. Thus, a *constraint solving* task is only executed for an arbitrary pair of output transitions $t, t' \in \tilde{T}^{*}_{o}$ where it holds that $t.l \neq t'.l$. The *constraint solving* task is executed for each transition pair of the group $\tilde{T}^{*}_{o}$ which complies to that condition.

Algorithm 5 Overlapping Guards Detection

```
1:  procedure CHECK_OVERLAPPING_GUARDS(group)
2:      for all transition pairs t, t' ∈ group do
3:          solverResult ← ⊥;
4:          c ← t.φ ∧ t'.φ';
5:          if group == T̃^p_{d_(l)} then
6:              solverResult ← solveConstraint(c ∧ t.l == t'.l);
7:          else
8:              solverResult ← solveConstraint(c);
9:          end if
10:         if solverResult ≠ ⊥ then
11:             "Guards t.φ and t'φ overlap!";
12:         end if
13:     end for
14: end procedure
```

Algorithm 5 shows the algorithm in order to detect overlapping guards within a group. The transitions of the group undergo pairwise *constraint solving* by considering the restriction in terms of groups containing two types of transitions. The *constraint solving* function **solveConstraint**(c) is called with the conjugated guard $\hat{\varphi}$ as constraint c, except when processing the delay transition group $\tilde{T}^{p}_{d_{(l)}}$. In this case the conjugated guard are extended by the condition which compares the timeout terms of the corresponding delay transitions, so that the given argument for the function is $\hat{\varphi} \wedge t.l = t'.l$.

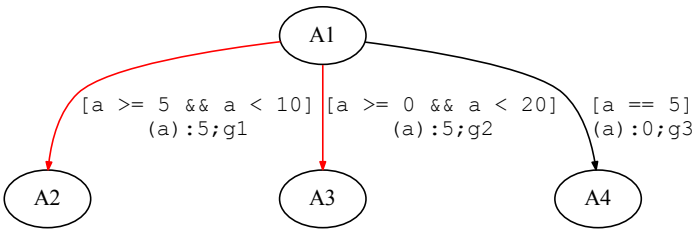


Figure 4.15.: Example of ESTS which exhibit a non-determinism in terms of delay transitions.

Example 4.17 (Overlapping Guard Detection)
Consider the ESTSs in Figures 4.15 and 4.16. Both ESTSs consist of only one state which has outgoing transitions, which are $A1$ and $B1$, respectively. The guards of the transitions are denoted in square brackets, followed by the label of the transition. Separated by a semicolon the priority is displayed which is followed - again separated by a semicolon - by the name of the timing group in case of a delay transition. The transition which are highlighted with blue and red, respectively, indicates a pair of transitions which exhibit non-determinism behavior.

Applying the Algorithm 5 to the ESTS in Figure 4.15 results in the following grouping sets for state $A1$. $A1$ is the only state which undergoes the overlapping guards check, since it is the only one

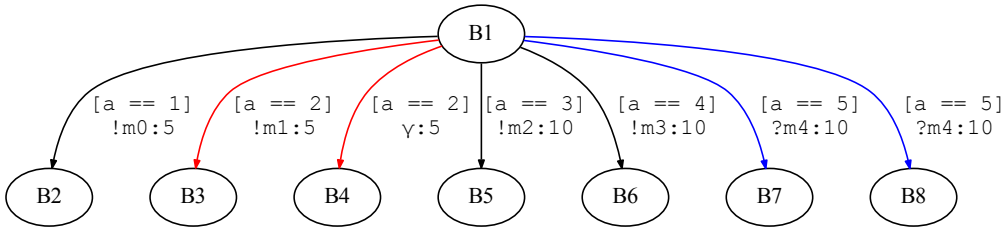


Figure 4.16.: Example ESTS which exhibit non-determinism due to an OC conflict since an output transition and a completion transition will enabled at the same time. Moreover the example comprises a non-determinism issue in terms of 2 enabled input transitions.

which has outgoing transitions. Since the outgoing transition set of state $A1$ in Figure 4.15 only consists of delay transitions the following transition groups are created:

- $\tilde{T}^{5}_{d_{(a)}} = \{(A1 \xrightarrow{(a)} A2), (A1 \xrightarrow{(a)} A3\}$
- $\tilde{T}^{0}_{d_{(a)}} = \{(A1 \xrightarrow{(a)} A4)\}$

Next, *constraint solving* is applied to the constraint $\mathfrak{c} = "\hat{\varphi}$ && $a == a"$, with the conjugated guard $\hat{\varphi} = "a \geq 5$ && $a < 10$ && $a \geq 0$ && $a < 20"$, for transition group $\tilde{T}^{5}_{d_{(a)}}$. Since the group $\tilde{T}^{5}_{d_{(a)}}$ contains two transitions a single *constraint solving* task is executed. By contrast, for the group $\tilde{T}^{0}_{d_{(a)}}$ no *constraint solving* task at all is executed as this group only contains a single transition. Due to the fact that the *constraint solver* can find at least one variable valuation to satisfy the given constraint, such that $\phi = \{(a, 5)\}$ for instance, the corresponding guards overlap in terms of their value range, and non-determinism issue is detected which is related to case (7).

Figure 4.16 shows an example which comprises two non-determinism issues. Based on the figure the following transition groups are created:

- $\tilde{T}^{5}_{o_{(m0)}} = \{(B1 \xrightarrow{!m0} B2)\}$
- $\tilde{T}^{5}_{o_{(m1)}} = \{(B1 \xrightarrow{!m1} B3)\}$
- $\tilde{T}^{10}_{o_{(m2)}} = \{(B1 \xrightarrow{!m2} B5)\}$
- $\tilde{T}^{10}_{o_{(m3)}} = \{(B1 \xrightarrow{!m3} B6)\}$
- $\tilde{T}^{*}_{o} = \{(B1 \xrightarrow{!m0} B2), (B1 \xrightarrow{!m1} B3), (B1 \xrightarrow{!m2} B5), (B1 \xrightarrow{!m3} B6)\}$
- $\tilde{T}^{10}_{i_{(m4)}} = \{(B1 \xrightarrow{?m4} B7),\ (B1 \xrightarrow{?m4} B8)\}$
- $T^{*}_{io} = \{(B1 \xrightarrow{!m0} B2), B1 \xrightarrow{!m1} B3), (B1 \xrightarrow{!m2} B5), (B1 \xrightarrow{!m3} B6),\ (B1 \xrightarrow{?m4} B7),\ (B1 \xrightarrow{?m4} B8)\}$
- $T^{5}_{\gamma} = \{(B1 \xrightarrow{\gamma} B4)\}$
- $T^{*}_{o\gamma} = \{(B1 \xrightarrow{!m0} B2), B1 \xrightarrow{!m1} B3), (B1 \xrightarrow{!m2} B5), (B1 \xrightarrow{!m3} B6),\ (B1 \xrightarrow{\gamma} B4)\}$

Next, the following *constraint solving* tasks are executed, whereas for groups $\tilde{T}^{5}_{o_{(m0)}}$, $\tilde{T}^{5}_{o_{(m1)}}$, $\tilde{T}^{10}_{o_{(m2)}}$, $\tilde{T}^{10}_{o_{(m3)}}$, and T^{5}_{γ} no *constraint solving* is applied, since those groups contain only a single transition.

- For group $\tilde{T}_o^*$:
  - Solve the constraint $\mathfrak{c} \leftarrow (a == 1 \;\&\&\; a == 2)$ for transitions $(B1 \xrightarrow{!m0} B2)$ and $(B1 \xrightarrow{!m1} B3)$, which is *not solvable*, such that solver result is $\perp$.
  - Solve the constraint $\mathfrak{c} \leftarrow (a == 1 \;\&\&\; a == 3)$ for transitions $(B1 \xrightarrow{!m0} B2)$ and $(B1 \xrightarrow{!m2} B5)$, which is *not solvable*, such that solver result is $\perp$.
  - Solve the constraint $\mathfrak{c} \leftarrow (a == 1 \;\&\&\; a == 4)$ for transitions $(B1 \xrightarrow{!m0} B2)$ and $(B1 \xrightarrow{!m3} B6)$, which is *not solvable*, such that solver result is $\perp$.
  - Solve the constraint $\mathfrak{c} \leftarrow (a == 2 \;\&\&\; a == 3)$ for transitions $(B1 \xrightarrow{!m1} B3)$ and $(B1 \xrightarrow{!m2} B5)$, which is *not solvable*, such that solver result is $\perp$.
  - Solve the constraint $\mathfrak{c} \leftarrow (a == 2 \;\&\&\; a == 4)$ for transitions $(B1 \xrightarrow{!m1} B3)$ and $(B1 \xrightarrow{!m3} B6)$, which is *not solvable*, such that solver result is $\perp$.
  - Solve the constraint $\mathfrak{c} \leftarrow (a == 3 \;\&\&\; a == 4)$ for transitions $(B1 \xrightarrow{!m2} B5)$ and $(B1 \xrightarrow{!m3} B6)$, which is *not solvable*, such that solver result is $\perp$.
- For each transition pair in $\tilde{T}^{10}_{i_{(m4)}}$
  - Solve constraint $\mathfrak{c} \leftarrow (a == 3 \;\&\&\; a == 3)$ for transitions $(B1 \xrightarrow{?m4} B7)$ and $(B1 \xrightarrow{?m4} B8)$, which results in $\phi = \{(a, 3)\}$ and therefore **depicts non-determinism behavior** which is related to case (2).
- Applying *constraint solving* to each distinct transition pair in T^*_{io} exhibits no non-determinism.
- Applying *constraint solving* to each distinct transition pair of group $T^*_{o\gamma}$ results into the detection of the following non-determinism case:
  - Solve the constraint $\mathfrak{c} \leftarrow (a == 2 \;\&\&\; a == 2)$ for transitions $(B1 \xrightarrow{!m1} B3)$ and $(B1 \xrightarrow{\gamma} B4)$ results in $\phi = \{(a, 2)\}$ and therefore **depicts non-determinism behavior** related to case (6).

□

4.3. Structure-based Checks

In the following, checks are presented which addresses the entire structure of an ESTS by extracting two path constructs. The first path construct represents a loop consisting only of unobservable, completion and output transitions. The second construct representing a specific sub-structure of an ESTS, namely a *Instantly Executable Transition Cascade*, which only consists of unobservable, completion and output transitions as well.

4.3.1. Instantly Executable Transition Loops

The goal of this check is to detect a loop path solely consisting of unobservable, completion and output transitions. Due to the static context of this check, no dynamic values are available and therefore all guards are considered *true*. Thus, each loop detected could be a potential endless loop

but must not be necessarily one. For that reason this check has only an informative purpose due to the lack of dynamic values. Therefore, in order to prove whether a loop is indeed endless, it has to be checked in a dynamic context.

A loop path shows the characteristics that the first state and the last state are equal and each transition of the path must be an unobservable, a completion or an output transition. The loops are calculated on the basis of the IPT structure. More precisely, the detection is carried out by validating each path of set $\mathfrak{P}^{\mathfrak{T}}$ in respect of the existence of loops.

A path contains a loop if the previously defined function **containsLoop(𝔭)** of Equation 4.6 returns *true*. By means of Equation 4.18, which depicts the function **startEdge**(𝔭), the start edge of the loop of a given path 𝔭 is determined. The equation denotes that the start edge $\mathfrak{e}_x$ of a path 𝔭 is the edge whose source node state $\mathfrak{n}_x.s$ is equal to the destination node state $n'_m.s$ of the last edge e_m of the path. The result is either a set containing a single edge, corresponding to the start edge of the loop, or the empty set, which states that the path does not contain a loop.

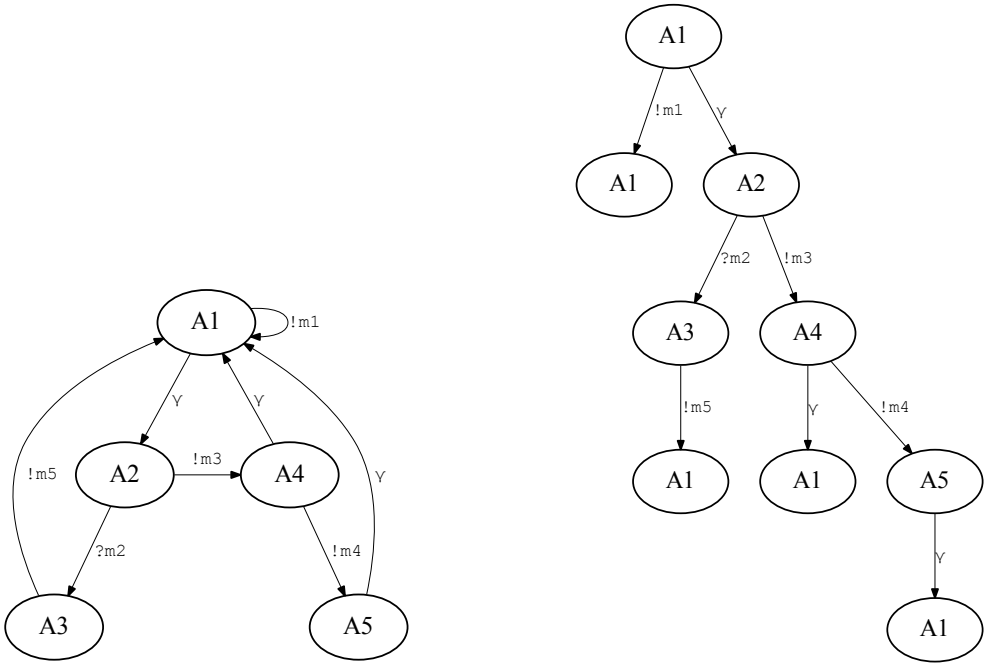
$$\textbf{startEdge}(\mathfrak{p}) = \{e_x \in \mathfrak{p} \mid \mathfrak{n}_x.s = \mathfrak{n}'_m.s \wedge x \leq m\} \tag{4.18}$$

Equation 4.19 states the formula to retrieve the path that depicts the loop from a given independent path 𝔭 with respect to the determined start edge $\mathfrak{e}_x$ of Equation 4.18. The stated function **retrieveLoopPath(𝔭, $\mathfrak{e}_x$)** returns a set $\mathfrak{l}$ containing all edges of path 𝔭 from index x, which is the index of the start edge, to the index m of the last node in 𝔭. In other words all edges with an index greater than or equal to x and less than or equal to m are considered and all other edges are neglected. Thus, the function results in set $\mathfrak{l} = \{\mathfrak{e}_x, \mathfrak{e}_{x+1}, ..., \mathfrak{e}_m\}$ with $\mathfrak{l} \subseteq \mathfrak{p}$. The set $\mathfrak{L} = \{\mathfrak{l}_1, ..., \mathfrak{l}_n\}$ denotes the set of all loops detected in an ESTS , with $\mathfrak{L} \subseteq \mathfrak{P}^{\mathfrak{T}}$.

$$\textbf{retrieveLoopPath}(\mathfrak{p}, \mathfrak{e}_x) = \{\mathfrak{e}_i \in \mathfrak{p} \mid x \leq i \leq m)\} \tag{4.19}$$

$$\textbf{isProperLoop}(\mathfrak{l}) = \begin{cases} \mathit{false} & \exists\ \mathfrak{e} \in \mathfrak{l} : \mathfrak{e}.t \notin T_{o\gamma\tau} \\ \mathit{true} & \text{otherwise} \end{cases} \tag{4.20}$$

Based on Equations 4.18 and 4.19 the existence of loops have been determined without considering the transition type of the edges. Which means that the loop determined might not be a loop solely consisting of *instantly executable* transitions. Loops consisting at least of one edge which does not correspond to an unobservable-, a completion or an output transition are not of relevance. Thus, each loop path $\mathfrak{l}$ must be verified accordingly by means of function **isProperLoop**(), which is stated in Equation 4.20. The function expects as input a loop path $\mathfrak{l}$ and returns *false* if at least one edge of the given loop path does not correspond to a transition of the set $T_{o\gamma\tau}$, otherwise the function returns *true*.



(a)ESTS with loops.

(b)IPT representation of the ESTS.

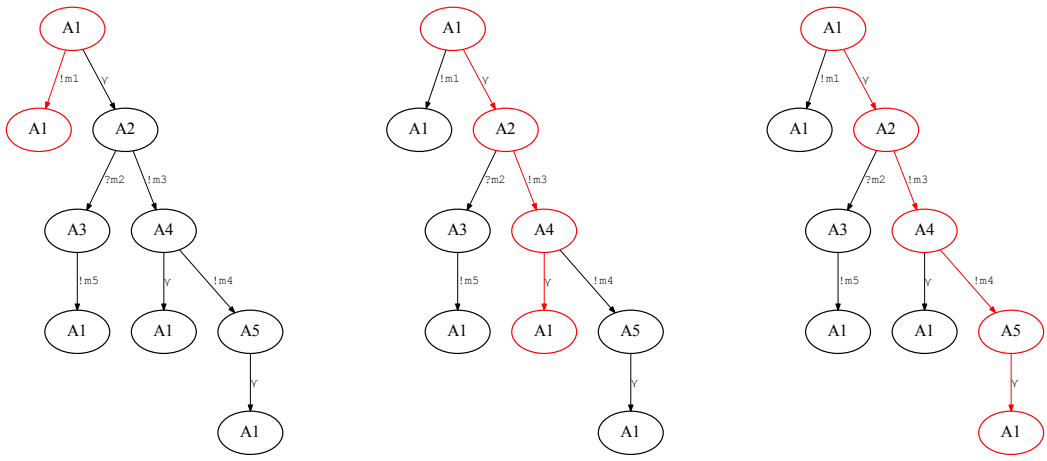
Figure 4.17.: An ESTS consisting of loops and its corresponding IPT.

Algorithm 6 shows the application of the previously defined formulas by means of the function **DETECT_LOOP**(). The function expects as input argument an independent path set $\mathfrak{P}^{\mathfrak{T}}$ and returns a set of loop paths $\mathfrak{L}$. For each path $\mathfrak{p} \in \mathfrak{P}^{\mathfrak{T}}$ the following steps are applied:

1. Determine the start edge of a loop by applying Equation 4.18.
2. Retrieve from the independent path $\mathfrak{p}$ the loop path $\mathfrak{l}$, which is a sub path of $\mathfrak{p}$, such that $\mathfrak{l} \subseteq \mathfrak{p}$.
3. Verify the loop path $\mathfrak{l}$ if the path corresponds to an instantly executable loop path. If so, add the loop path to the result set $\mathfrak{L}$, such that $\mathfrak{l} \in \mathfrak{L}$.
4. Check the number of detected loops against a maximum number of detectable loops *MaxDetectedLoops*. If the maximum is reached the algorithm stops. Since in a directed graph the detection algorithm runtime can be exponentially in the worst case, the limit *MaxDetectedLoops* is introduced. The limit denotes a configurable upper barrier for the detection of loops and thus ensures that the algorithm is executed in a feasible amount of time.

If the result set $\mathfrak{L}$ is empty, no *instantly executable* loops are detected.

Example 4.18 (Loop Detection)
Consider the ESTS depicted in Figure 4.17a and its IPT structure shown in Figure 4.17b. The set of independent paths $\mathfrak{P}^{\mathfrak{T}}$ of the path tree in Figure 4.17b is $\mathfrak{P}^{\mathfrak{T}} = \{\mathfrak{p}_1, \mathfrak{p}_2, \mathfrak{p}_3, \mathfrak{p}_4\}$ with:



(a)First loop (b)Second loop (c)Third loop

Figure 4.18.: Depiction of three loops as IPT.

Algorithm 6 Loop Detection

```
1: function DETECT_LOOP(𝔓^𝔗) → 𝔏
2:     𝔏 = ∅;
3:     for 𝔭 ∈ 𝔓^𝔗 do
4:         𝔢_x = startEdge(𝔭);                          ▷ applying Equation 4.18
5:         𝔩 = retrieveLoopPath(𝔭, 𝔢_x);                ▷ applying Equation 4.19
6:         if isProperLoop(𝔩) == true then              ▷ applying Equation 4.20
7:             𝔏 = 𝔏 ∪ 𝔩;
8:         end if
9:         if |𝔏| == MaxDetectedLoops then
10:            return 𝔏;
11:        end if
12:    end for
13:    return 𝔏;
14: end function
```

- $\mathfrak{p}_1 = \{(A1, !m1, A1)\}$
- $\mathfrak{p}_2 = \{(A1, \gamma, A2), (A2, ?m2, A3), (A3, !m5, A1)\}$
- $\mathfrak{p}_3 = \{(A1, \gamma, A2), (A2, !m3, A4), (A4, \gamma, A1)\}$
- $\mathfrak{p}_4 = \{(A1, \gamma, A2), (A2, !m3, A4), (A4, !m4, A5), (A5, \gamma, A1)\}$

By means of Equation 4.18 for each $\mathfrak{p} \in \mathfrak{P}^{\mathfrak{T}}$ the following start edges are determined:

- $\mathfrak{e}_x^{\mathfrak{p}_1} = (A1, !m1, A1)$ is determined, since $\mathfrak{n}_x.s = A1$, $\mathfrak{n}'_m.s = A1$, $x = 0$, $m = 0$ and therefore complies to $A1 == A1 \wedge 0 \leq 0$, depicting a self loop.
- $\mathfrak{e}_x^{\mathfrak{p}_2} = (A1, \gamma, A2)$ is determined, since $\mathfrak{n}_x.s = A1$, $\mathfrak{n}'_m.s = A1$, $x = 0$, $m = 2$ and therefore complies to $A1 == A1 \wedge 0 \leq 2$.
- $\mathfrak{e}_x^{\mathfrak{p}_3} = (A1, \gamma, A2)$ is determined, since $\mathfrak{n}_x.s = A1$, $\mathfrak{n}'_m.s = A1$, $x = 0$, $m = 2$ and therefore complies to $A1 == A1 \wedge 0 \leq 2$.
- $\mathfrak{e}_x^{\mathfrak{p}_4} = (A1, \gamma, A2)$ is determined, since $\mathfrak{n}_x.s = A1$, $\mathfrak{n}'_m.s = A1$, $x = 0$, $m = 3$ and therefore

complies to $A1 == A1 \wedge 0 \leq 3$.

The following loop paths are retrieved by applying Equation 4.19 to each path of $\mathfrak{P}^{\mathfrak{T}}$ by considering the determined start edges.

- $\mathfrak{l}_1 = \{(A1, !m1, A1)\}$
- $\mathfrak{l}_2 = \{(A1, \gamma, A2), (A2, ?m2, A3), (A3, !m5, A1)\}$
- $\mathfrak{l}_3 = \{(A1, \gamma, A2), (A2, !m3, A4), (A4, \gamma, A1)\}$
- $\mathfrak{l}_4 = \{(A1, \gamma, A2), (A2, !m3, A4), (A4, !m4, A5), (A5, \gamma, A1)\}$

Finally, each loop path is verified according to Equation 4.20. The loop paths $\mathfrak{l}_1$, $\mathfrak{l}_3$ and $\mathfrak{l}_4$ comply to Equation 4.20 and $\mathfrak{l}_2$ does not due to the edge $(A2, ?m3, A3)$, which is related to an input transition.

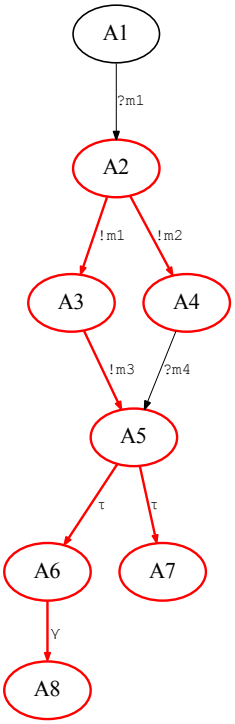
Thus, the algorithm detects three loops, which are illustrated in Figure 4.18, and therefore the result set is $\mathfrak{L} = \{\mathfrak{l}_1, \mathfrak{l}_3, \mathfrak{l}_4\}$ □

4.3.2. Instantly Executable Transition Cascades

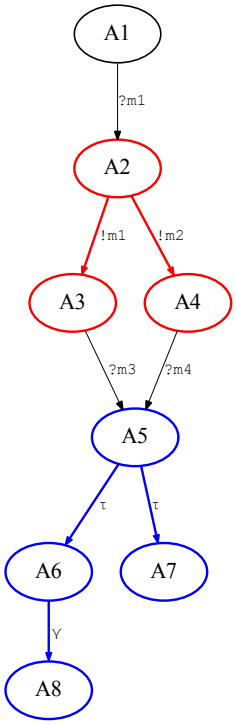
The goal of this check is to detect a specific sub-structure in an ESTS which solely consists of the *instantly executable* transition like the unobservable, completion and output namely an *Instantly Executable Transition Cascade*. Basically a *cascade* is a set of transitions such that $cascade \subset T_{o\gamma\tau}$, with$T_{o\gamma\tau}$ denoting the set of instantly executable transitions. Besides the detection of such cascades, for each cascade the possible paths are determined in order to compare the number of paths against the limit *MaxPathCount*. As a further consequence each cascade which exceeds the *MaxPathCount* limit is reported by this check.

Example 4.19 (Cascade)
Examples of such cascade are shown in Figure 4.19. Figure 4.19a depicts only one cascade, $cascade_1$, whose corresponding states and transitions are highlighted in red. The cascade consists of transitions $\{(A2 \xrightarrow{!m1} A3), (A2 \xrightarrow{!m2} A4), (A3 \xrightarrow{!m3} A5), (A5 \xrightarrow{\tau} A6), (A5 \xrightarrow{\tau} A7), (A6 \xrightarrow{\gamma} A8)\}$. Figure 4.19b depicts two cascades due to the fact that the transitions from state $A3$ to $A5$ and $A4$ to $A5$ belong to an input transition. The corresponding cascades are $cascade_2 = \{(A2 \xrightarrow{!m1} A3), (A2 \xrightarrow{!m2} A4)\}$, which is highlighted in red, and $cascade_3 = \{(A5 \xrightarrow{\tau} A6), (A5 \xrightarrow{\tau} A7), (A6 \xrightarrow{\gamma} A8)\}$, which is highlighted in blue. □

A *cascade* shows the characteristic that it depicts a connected fragment $e_{cascade}$ of the original ESTS e, with $e_{cascade} \subseteq e$. Connected means that from a *start state* to each *end state* of the cascade a path can be constructed. In order to define the terms *start state* and *end state* some fundamental definitions have to be stated beforehand. The detection of a cascade is based on the IPT representation of an ESTS, and thus the definitions of a *start state* and *end state* are stated in terms of an IPT, such that $\mathfrak{T} = \textbf{estsToPathTree}(e)$. In the context of an IPT connected means that from a *start node* to each *end node* a path can be constructed. The IPT structure is used in order to benefit from its structure as in contrast to an ESTS no loops exists and therefore such cases must not be considered.



(a)ESTS containing a single instantly executable transition cascade.



(b)ESTS containing two instantly executable transition cascades.

Figure 4.19.: Examples of *Instantly Executable Transition Cascades*.

The first step consists of the determination of potential *start nodes* of a cascade in the IPT, which is carried out by applying Equation 4.21. Basically, a *start node* exhibits the property that the incoming transition is not an instantly executable transition, such that $\mathbf{nodeInTrans}(\hat{n}) \notin T_{o\gamma\tau}$, and the outgoing transition set contains at least one instantly executable transition, such that $\exists t \in T_{o\gamma\tau} \wedge t \in \mathbf{nodeOutTrans}(\hat{n})$.

$$\mathbf{startNodes}(\mathfrak{T}) = \{n \in \mathfrak{N} \mid \mathbf{nodeInTrans}(n) \notin T_{o\gamma\tau} \wedge (\exists t \in \mathbf{nodeOutTrans}(n) : t \in T_{o\gamma\tau})\} \tag{4.21}$$

Example 4.20 (Extracting Start Nodes)
Applying Equation 4.21 to the IPT which is depicted in Figure 4.20 results in the detection of two *start nodes* corresponding to the states $A2$ and $A5$. This example shows the reason that these nodes are only considered as potential *start nodes*. The cascade, starting at node $A5$ is already part of the cascade starting at node $A2$. Thus, in Figure 4.20 actually only a single cascade exists which starts at $A2$. □

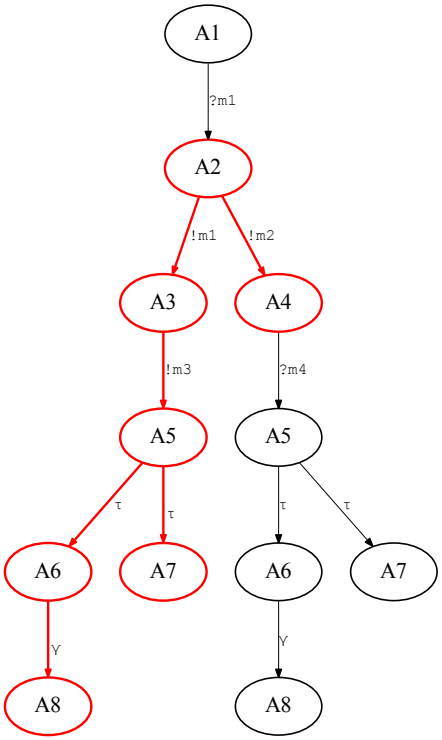


Figure 4.20.: IPT which is based on the ESTS of Figure 4.19a and containing a single *Instantly Executable Transitions Cascade*, which is highlighted in red.

Each cascade has to be verified whether it is indeed a cascade or part of another cascade. Let $cascade_1$ and $cascade_2$, with $cascade_1, cascade_2 \in T_{o\gamma\tau}$, be two arbitrary cascades with different *start nodes*, then $cascade_1$ is part of $cascade_2$ if $cascade_1 \subseteq cascade_2$. This means that each transition which is related to $cascade_1$ is also related to $cascade_2$.

The verification and extraction of the cascades are carried out by means of Algorithm 7. The algorithm depicts the function *EXTRACT_CASCADES*, which extracts all cascades from the given IPT, which represents an entire ESTS, based on the determined *start nodes* SN. The function returns a set of IPTs $\widehat{\mathfrak{T}}^*$, where each IPT of the set corresponds to a cascade. The returned IPTs exhibit the characteristics that all edges correspond to transitions of the instantly executable transition type. Furthermore, the IPTs and as a further consequence the cascades are mutually distinct, which means that the IPTs differ in at least one edge.

Algorithm 7 Extracting an IPT for each cascade

```
1: procedure EXTRACT_CASCADES(𝔗, SN) → 𝔗̂*
2:     𝔗̃* ← ⋃_{sn∈SN} subTree(sn, 𝔗);
3:     𝔓* ← ⋃_{𝔗∈𝔗̃*} treeToPaths(𝔗);
4:     𝔓̂* ← ⋃_{𝔓∈𝔓*} reducePaths(𝔓);
5:
6:     𝔓̃* ← 𝔓̂*;
7:     for all reduced path set pairs 𝔓̂_x, 𝔓̂_y ∈ 𝔓̂*, with x ≠ y do
8:         𝔓̃* ← 𝔓̃* \ getIncorrectCascade(𝔓̂_x, 𝔓̂_y);
9:     end for
10:
11:    𝔗̂* ← ⋃_{𝔓∈𝔓̃*} pathsToTree(𝔓);
12:    return 𝔗̂*;
13: end procedure
```

The first part of the algorithm deals with the extraction of *potential* cascades with respect to the determined *start nodes*. The extraction of the cascades consists of three steps:

1. Create based on each *start node*, with $sn \in SN$ denoting the *start node*, a sub-tree, whose root node corresponds to sn.
2. Create for each created sub-tree a corresponding path set by means of $\mathfrak{P}^* = \bigcup_{\mathfrak{T} \in \tilde{\mathfrak{T}}^*} \textbf{treeToPaths}(\mathfrak{T})$, with $\mathfrak{P}^*$ denoting the set of all path sets of all sub-trees.
3. The last step reduces each path set by removing all edges which are not related the a cascade. This is established by means of Equation 4.22 and function **reducePaths**($\mathfrak{P}$). The function is called for each path set of the set $\mathfrak{P}^*$. Thus, this step results in the reduced set of path sets $\hat{\mathfrak{P}}^*$.

$$\textbf{reducePathSet}(\mathfrak{P}) = \bigcup_{\mathfrak{p} \in \mathfrak{P}} \textbf{reducePath}(\mathfrak{p}, \mathfrak{e}_x) \tag{4.22}$$

$$\textbf{reducePaths}(\mathfrak{p}, \mathfrak{e}_x) = \{\mathfrak{e}_i \in \mathfrak{p} \mid 1 \leq i < x\} \tag{4.23}$$

Equation 4.22 reduces all paths of a given path set $\mathfrak{P}$ and returns the reduced set of paths $\hat{\mathfrak{P}}$. For each path $\mathfrak{p}$, with $\mathfrak{p} \in \mathfrak{P}$, the function **reducePath**($\mathfrak{p}, \mathfrak{e}_x$), which is stated in Equation 4.23, is applied. The edge $\mathfrak{e}_x$, with $\mathfrak{e}_x \in \mathfrak{p}$ and $1 \leq x < m$, corresponds to the first edge in the path which does not belong to an instantly executable transition. Thus $\mathfrak{e}_x.t \notin T_{o\gamma\tau}$. The paths are reduced by removing all edges from a path starting at edge $\mathfrak{e}_x$.

In the next step Algorithm 7 deals with the verification of cascades which are already part of other cascades. This step results in the set $\tilde{\mathfrak{P}}^*$, which denotes the set of all path sets representing those cascades which are mutually distinct. First of all, the set $\tilde{\mathfrak{P}}^*$ comprises all cascades of all detected *potential* cascades, such that $\tilde{\mathfrak{P}}^* = \hat{\mathfrak{P}}^*$. Next, all cascades are removed which are already part of another cascade. This is achieved by applying function **getIncorrectCascade**($\hat{\mathfrak{P}}_x, \hat{\mathfrak{P}}_y$) to all possible pair of reduced path sets $\hat{\mathfrak{P}}_x, \hat{\mathfrak{P}}_y \in \hat{\mathfrak{P}}^*$, with $x \neq y$. As depicted in Equation 4.25 the function returns

either the path set which is part of the other path set or the empty set. The empty set denotes that neither the first path set is part of the second path set nor is this the case the other way round. In other words the function returns exactly the reduced path set which should be removed from the set of actual cascade path sets $\widetilde{\mathfrak{P}}^*$. Thus, after all pairs of reduced path sets are checked only mutually distinct path sets are present in the set $\widetilde{\mathfrak{P}}^*$.

$$\textbf{transPathSet}(\mathfrak{P}) = \bigcup_{\mathfrak{p} \in \mathfrak{P}} \{\mathfrak{e}.t \mid \mathfrak{e} \in \mathfrak{p}\} \tag{4.24}$$

Equation 4.25 utilizes Equation 4.24, which transforms the path sets into a set of transitions. Based on the transition set, the check whether a cascade is part of another cascade is carried out. The reason is that in a path set a particular transition can be assigned to multiple edges. In contrast the transition set contains a transition at most once.

$$\textbf{getIncorrectCascade}(\hat{\mathfrak{P}}_x, \hat{\mathfrak{P}}_y) = \begin{cases} \{\hat{\mathfrak{P}}_x\} & \textit{iff } \textbf{transPathSet}(\hat{\mathfrak{P}}_x) \subseteq \textbf{transPathSet}(\hat{\mathfrak{P}}_y) \\ & \wedge\ \textbf{transPathSet}(\hat{\mathfrak{P}}_y) \not\subseteq \textbf{transPathSet}(\hat{\mathfrak{P}}_x) \\ \{\hat{\mathfrak{P}}_y\} & \textit{iff } \textbf{transPathSet}(\hat{\mathfrak{P}}_y) \subseteq \textbf{transPathSet}(\hat{\mathfrak{P}}_x) \\ & \wedge\ \textbf{transPathSet}(\hat{\mathfrak{P}}_x) \not\subseteq \textbf{transPathSet}(\hat{\mathfrak{P}}_y) \\ \emptyset & \textit{otherwise} \end{cases} \tag{4.25}$$

Finally, Algorithm 7 creates the set $\widehat{\mathfrak{T}}^*$, with $\widehat{\mathfrak{T}}^* = \{\widehat{\mathfrak{T}}_1, \widehat{\mathfrak{T}}_2, ..., \widehat{\mathfrak{T}}_n\}$, depicting the set of IPTs, where each IPT corresponds to a proper cascade.

Example 4.21 (Extraction of Cascades)
Consider the ESTS depicted in Figure 4.19a and its corresponding representation as IPT in Figure 4.20. The ESTS and its corresponding IPT are defined as follows:

- The ESTS e is defined with the set of states $S = \{A1, A2, A3, A5, A6, A7, A8\}$ and the set of transitions $T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8\}$ with:
  - $t_1 = (A1 \xrightarrow{?m1} A2)$
  - $t_2 = (A2 \xrightarrow{!m1} A3)$
  - $t_3 = (A2 \xrightarrow{!m2} A4)$
  - $t_4 = (A3 \xrightarrow{!m3} A5)$
  - $t_5 = (A4 \xrightarrow{?m4} A5)$
  - $t_6 = (A5 \xrightarrow{\tau} A6)$
  - $t_7 = (A5 \xrightarrow{\tau} A7)$
  - $t_8 = (A6 \xrightarrow{\gamma} A8)$
- The IPT $\mathfrak{T}$ created for ESTS e, with $\mathfrak{T} = \textbf{estsToPathTree}(e)$ is defined by the node set $\mathfrak{N} = \{\mathfrak{n}_1, \mathfrak{n}_2, \mathfrak{n}_3, \mathfrak{n}_4, \mathfrak{n}_5, \mathfrak{n}_6, \mathfrak{n}_7, \mathfrak{n}_8, \mathfrak{n}_9, \mathfrak{n}_{10}, \mathfrak{n}_{11}, \mathfrak{n}_{12}\}$ with $\mathfrak{n}_1 = A1$, $\mathfrak{n}_2 = A2$, $\mathfrak{n}_3 = A3$, $\mathfrak{n}_4 = A4$, $\mathfrak{n}_5 = A5$, $\mathfrak{n}_6 = A6$, $\mathfrak{n}_7 = A7$, $\mathfrak{n}_8 = A8$, $\mathfrak{n}_9 = A5$, $\mathfrak{n}_{10} = A6$, $\mathfrak{n}_{11} = A7$, $\mathfrak{n}_{12} = A8$. And the edge set $\mathfrak{E} = \{\mathfrak{e}_1, \mathfrak{e}_2, \mathfrak{e}_3, \mathfrak{e}_4, \mathfrak{e}_5, \mathfrak{e}_6, \mathfrak{e}_7, \mathfrak{e}_8, \mathfrak{e}_9, \mathfrak{e}_{10}, \mathfrak{e}_{11}\}$ with:

- $e_1 = \{n_1, t_1, n_2\}$
- $e_2 = \{n_2, t_2, n_3\}$
- $e_3 = \{n_3, t_4, n_5\}$
- $e_4 = \{n_5, t_6, n_6\}$
- $e_5 = \{n_6, t_8, n_8\}$
- $e_6 = \{n_5, t_7, n_7\}$
- $e_7 = \{n_2, t_3, n_4\}$
- $e_8 = \{n_4, t_5, n_9\}$
- $e_9 = \{n_9, t_6, n_{10}\}$
- $e_{10} = \{n_{10}, t_8, n_{12}\}$
- $e_{11} = \{n_9, t_7, n_{11}\}$

Applying Equation 4.21 to the IPT $\mathfrak{T}$ results in the set of *start nodes* $SN = \{n_2, n_9\}$. By considering Algorithm 7, for each startnode $sn \in SN$ a sub-tree is created which results in the set $\widetilde{\mathfrak{T}}^* = \{\widetilde{\mathfrak{T}}_{(n_2)}, \widetilde{\mathfrak{T}}_{(n_9)}\}$, and as a further consequence into the set $\mathfrak{P}^* = \{\mathfrak{P}^{\mathfrak{T}_{(n_2)}}, \mathfrak{P}^{\mathfrak{T}_{(n_9)}}\}$, with:

- $\mathfrak{P}^{\mathfrak{T}_{(n_2)}} = \{p_1, p_2, p_3, p_4\}$ with $p_1 = \{e_2, e_3, e_4, e_5\}$, $p_2 = \{e_2, e_3, e_6\}$, $p_3 = \{e_7, e_8, e_9, e_{10}\}$ and $p_4 = \{e_7, e_8, e_{11}\}$.
- $\mathfrak{P}^{\mathfrak{T}_{(n_9)}} = \{p_1, p_2\}$ with $p_1 = \{e_9, e_{10}\}$ and $p_2 = \{e_{11}\}$.

Next, each path set of the set $\mathfrak{P}^*$ is reduced as shown subsequently:

1. Reduce path set $\mathfrak{P}^{\mathfrak{T}_{(n_2)}} \in \mathfrak{P}^*$, which results in the set $\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_2)}} = \{\widehat{p}_1, \widehat{p}_2, \widehat{p}_3\}$, which is depicted as IPT in Figure 4.21b.
   a) Reduce path $p_1 \in \mathfrak{P}^{\mathfrak{T}_{(n_2)}} \rightarrow$ no edges are removed from path. Thus $\widehat{p}_1 = p_1$.
   b) Reduce path $p_2 \in \mathfrak{P}^{\mathfrak{T}_{(n_2)}} \rightarrow$ no edges are removed from path. Thus $\widehat{p}_2 = p_2$.
   c) Reduce path $p_3 \in \mathfrak{P}^{\mathfrak{T}_{(n_2)}} \rightarrow$ edges e_8, e_9 and e_{10} are removed from path, since e_8 corresponds to an input transition. Thus $\widehat{p}_3 = \{e_7\}$.
   d) Reduce path $p_4 \in \mathfrak{P}^{\mathfrak{T}_{(n_2)}} \rightarrow$ edges e_8 and e_{11} are removed from path, since e_8 corresponds to an input transition. Thus, $\widehat{p}_4 = \{e_7\}$. As $\widehat{p}_4 = \widehat{p}_3$, $\widehat{p}_4$ is neglected.
2. Reduce path set $\mathfrak{P}^{\mathfrak{T}_{(n_9)}} \in \mathfrak{P}^*$, which results in the set $\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_9)}} = \mathfrak{P}^{\mathfrak{T}_{(n_9)}}$. The corresponding IPT is depicted in Figure 4.21c.

The set $\widehat{\mathfrak{P}}^* = \{\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_2)}}, \widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_9)}}\}$ denotes the result of the reduction step.

Finally, the cascades corresponding to the reduced path sets of set $\widehat{\mathfrak{P}}^*$ are checked whether a cascade is part of another cascade. The following steps are applied according to Algorithm 7:

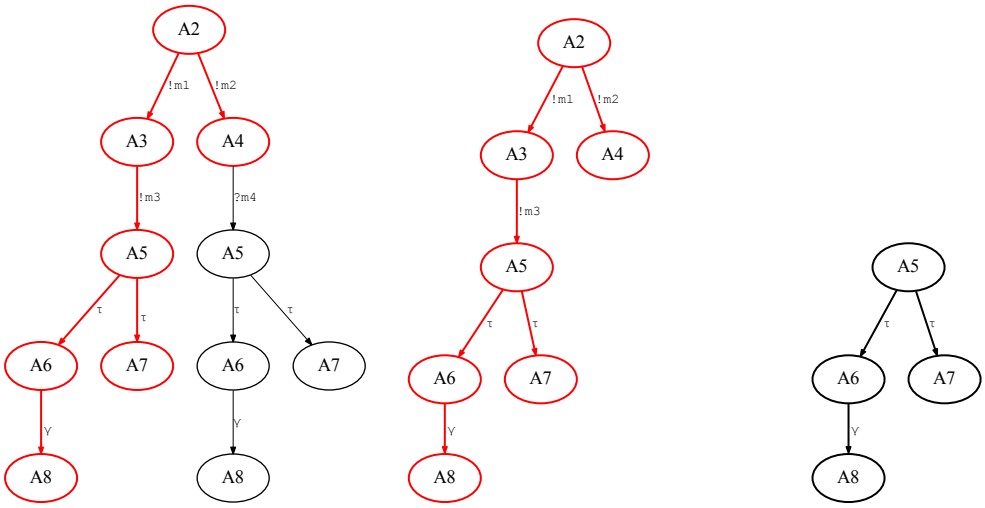
1. $\widetilde{\mathfrak{P}}^* = \widehat{\mathfrak{P}}^*$
2. Call **getIncorrectCascade**$(\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_2)}}, \widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_9)}})$
   a) Create set $T^{\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_2)}}}$ based on the path set $\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_2)}}$ by applying $T^{\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_2)}}} =$ **transPathSet**$(\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_2)}})$, such that $T^{\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_2)}}} = \{t_2, t_3, t_4, t_5, t_6, t_7, t_8\}$.
   b) Create set $T^{\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_9)}}}$ based on the path set $\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_9)}}$ by applying $T^{\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_9)}}} =$ **transPathSet**$(\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_9)}})$, such that $T^{\widehat{\mathfrak{P}}^{\mathfrak{T}_{(n_9)}}} = \{t_6, t_7, t_8\}$.

c) Since $T^{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_2)} \subseteq T^{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_9)} \wedge T^{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_9)} \not\subseteq T^{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_2)}$ complies to $false \wedge true$, case one of Equation 4.25 does not apply.

d) Since $T^{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_9)} \subseteq T^{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_2)} \wedge T^{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_2)} \not\subseteq T^{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_9)}$ complies to $true \wedge true$, case one of Equation 4.25 does apply and thus function **getIncorrectCascade** returns the set $\{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_9)\}$.

3. Next, $\widetilde{\mathfrak{P}}^* = \widetilde{\mathfrak{P}}^* \setminus \{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_9)\}$ with $\widetilde{\mathfrak{P}}^* = \{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_2), \widehat{\mathfrak{P}}^{\mathfrak{T}}(n_9)\} \setminus \{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_9)\}$, resulting in $\widetilde{\mathfrak{P}}^* = \{\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_2)\}$.
4. Finally, $\widehat{\mathfrak{T}}^* = \{\widehat{\mathfrak{T}}\widehat{\mathfrak{P}}^{\mathfrak{T}}(n_2)\}$. Thus, the algorithm extracts one cascade, which is highlighted in red in Figure 4.21a.

□



(a)IPT corresponding to the cascade with *start node A2*.

(b)Reduced IPT corresponding to the cascade with *start node A2*.

(c)IPT corresponding to the cascade with *start node A5*, the reduced IPT has the same appearance.

Figure 4.21.: IPTs of cascades and their reduced representations which are based on the IPT in Figure 4.20 with *start nodes A2* and *A5*.

Once the cascades are determined the computation of possible paths through a cascade is performed. During path computation the following basic rules are applied:

1. If a node has an outgoing transition which is either of type unobservable or completion, the path counter is incremented by 2, which is depicted in Figure 4.22a.
2. If a node has an outgoing transition which is of type output the path counter is incremented by *1*, which is depicted in Figure 4.22b and 4.22c.

The rules are only of basic nature and defines how path computation is carried out considering a single tree node and by ignoring special structures within a cascade, such as branches. Due to the static context of this check no concrete values for the guards of the involved transitions are present.

For this reason, both possible cases of guards are considered during path computation, which means that a guard can either be *true* or *false*.

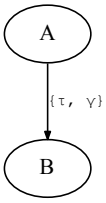
On that account the first rule, depicted in Figure 4.22a, counts two paths for a transition which has an outgoing transition of type unobservable or completion. The first path is counted for the case that the guard remains *false* and therefore the transition is not enabled, so that the system remains in state A. The second path is counted when the guard is *true*, so that the transition will be traversed and therefore the system changes from state A to state B. Thus, the first rule counts the paths (A) and $(A \xrightarrow{\{\tau,\gamma\}} B)$.

The second rule is applied in case of an output transition due to a special treatment of output guards. Those guards are used in order to set the values for the message parameters. Therefore, in context of path computation, such guards are considered always *true* and therefore the transition is considered always be enabled. Thus, for Figure 4.22b only one path is computed which is $(A \xrightarrow{!} B)$. Furthermore, if two output transitions are present in a row, as shown in Figure 4.22c, both transitions are considered part of the same path. This means that if a path is reaching a destination state with an output transition, the output transition is part of the same path. Thus, with respect to Figure 4.22c a single path, $(A \xrightarrow{!} B \xrightarrow{!} C)$, is computed.

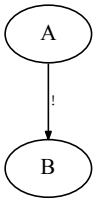
The algorithm to compute the possible paths of a cascade is applied to an IPT by iterating through the nodes of the tree in a *Depth First Search* style, starting at the *root node*. The path computation is finished if all branches of the IPT has been traversed. During iterating through the tree, each node is checked whether one of the four cases illustrated in Figure 4.23 applies. These cases are based on the previously discussed basic rules. Based on the cases the path counter is incremented accordingly as shown in Equation 4.26.

$$\textbf{computePathCount}(n) = \begin{cases} |\{t \in \textbf{nodeOutTrans}(n)) \mid t \in T_\tau \cup T_\gamma\}| + 1 & \textbf{isPathCountCase1}(n) = \mathit{true} \\ |\{t \in \textbf{nodeOutTrans}(n) \mid t \in T_o\}| & \textbf{isPathCountCase2}(n) = \mathit{true} \\ |\{t \in \textbf{nodeOutTrans}(n) \mid t \in T_{o\gamma\tau}\}| & \textbf{isPathCountCase3}(n) = \mathit{true} \\ |\{t \in \textbf{nodeOutTrans}(n) \mid t \in T_o\}| - 1 & \textbf{isPathCountCase4}(n) = \mathit{true} \end{cases} \tag{4.26}$$

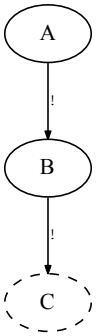
The first case handles the IPT *root node* which has outgoing transitions of type unobservable and completion as shown in Figure 4.23a. As stated in Equation 4.27 a node which is the *root node* and which has at least one outgoing transition of type unobservable or completion is covered by this case. In this case the path counter is increased by the number of outgoing transitions of type unobservable and completion. Furthermore, an additional path is counted which is related to the path that no transition is enabled and the system remains at the source state. With respect to Figure 4.23a the paths (A) and $(A \xrightarrow{\tau,\gamma} B)$ are counted.



(a)Two paths are counted, since the outgoing transition is either of type unobservable or completion and the guard is considered *true* and *false*.



(b)One path is counted since the guard of the output transition is always considered *true*.



(c)One path is counted since both output transition guards are considered *true*.

Figure 4.22.: Basic rules for path computation considering only the outgoing transition type.

$$\textbf{isPathCountCase1}(n) = \begin{cases} \textit{true} & \exists\, t \in \textbf{nodeOutTrans}(n) : t \in T_\tau \cup T_\gamma \wedge \textbf{parentNode}(n) = \textit{NIL} \\ \textit{false} & \text{otherwise} \end{cases} \tag{4.27}$$

The second case handles the IPT *root node* which has at least one outgoing transition of type output, as shown in Figure 4.23b and stated in Equation 4.28. In this particular case the path counter is incremented by the number of output transitions. Compared to the previous case, no additional path is counted since output transitions are considered always be enabled. Therefore, only the path $(B \xrightarrow{t} C)$ will be counted.

$$\textbf{isPathCountCase2}(n) = \begin{cases} \textit{true} & \exists\, t \in \textbf{nodeOutTrans}(n) : t \in T_o \wedge \textbf{parentNode}(n) = \textit{NIL} \\ \textit{false} & \text{otherwise} \end{cases} \tag{4.28}$$

The third case addresses nodes which are not the *root node* of the IPT and whose outgoing transition set comprises at least one transition which is related to an unobservable or completion transition. The remaining outgoing transitions can correspond to any other instantly executable transition type. This case is depicted in Figure 4.23c and the corresponding Equation 4.29 denotes the determination whether a node applies to this case or not. This is the case when a node, which is not the *root node*, that has at least one outgoing transition corresponding to type unobservable or completion. In this case the path counter is incremented by the number of outgoing transitions of type output, unobservable and completion. In contrast to the first case, no additional path is counted although an outgoing transition of either type unobservable or completion is present. The reason is that the path, which indicates that the system remains at the source state, has already been counted when processing the parent node. Thus, with respect to Figure 4.23c the paths $(C \xrightarrow{\tau,\gamma} D)$ and $(C \xrightarrow{!,\tau,\gamma} E)$ are counted. The path which implies that the system remains at node C - due to not satisfied guards - is covered by path $(B \rightarrow C)$ which is counted when processing node B.

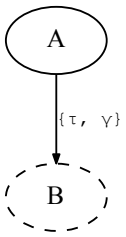
$$\textbf{isPathCountCase3}(\mathfrak{n}) = \begin{cases} \mathit{true} & \exists\, t \in \textbf{nodeOutTrans}(\mathfrak{n}) : (t \in T_\tau \cup T_\gamma) \wedge (\textbf{parentNode}(\mathfrak{n}) \neq \mathit{NIL}) \\ \mathit{false} & \text{otherwise} \end{cases} \tag{4.29}$$

The last case deals with the fact that the outgoing transition set of a non-*root node* only comprises output transitions, as stated in Equation 4.30. In this case the path count is increased by the number of output transitions - 1. The -1 stems from the fact that one of the output transition is considered as part of a path, which has been calculated when processing the parent node. Furthermore, this handling deals with the issue which has been pointed out by Figure 4.22c.

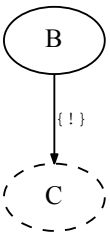
$$\textbf{isPathCountCase4}(\mathfrak{n}) = \begin{cases} \mathit{true} & \nexists\, t \in \textbf{nodeOutTrans}(\mathfrak{n}) : t \in T_\tau \cup T_\gamma \wedge \textbf{parentNode}(\mathfrak{n}) \neq \mathit{NIL} \\ \mathit{false} & \text{otherwise} \end{cases} \tag{4.30}$$

Algorithm 8 Instantly Executable Cascades Path Calculation

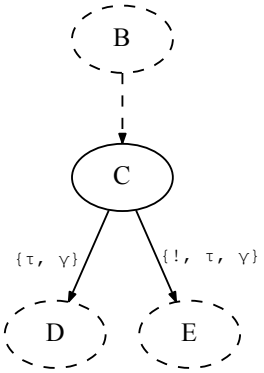
```
1:  procedure CHECK_CASCADES(𝔗̂*, MaxPathCount)
2:      CascadeSet ← ∅;
3:      for all 𝔗 ∈ 𝔗̂* do
4:          pathCount ← 0;
5:          cascade ← ∅;
6:
7:          𝔑^DFS ← PRE_ORDER_DFS_SORTING(𝔗);
8:          for n ∈ 𝔑^DFS do
9:              cascade ← cascade ∪ {t ∈ nodeOutTrans(n) | t ∈ T_oγτ};
10:             pathCount ← pathCount + computePathCount(n);        ▷ Application of Equation 4.26
11:         end for
12:
13:         if pathCount > MaxPathCount then
14:             CascadeSet ← CascadeSet ∪ cascade;
15:         end if
16:     end for
17:     REPORT(CascadeSet);
18: end procedure
```



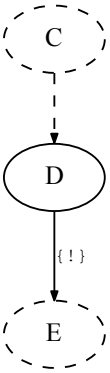
(a)Case 1: *start node A*, whose outgoing transition set comprises transitions of type unobservable and completion.



(b)Case 2: *start node B*, whose outgoing transition set comprises transitions of type output.



(c)Case 3: Node *C* which is no *start node* and the outgoing transition set contains at least one transition which is of type unobservable or completion.



(d)Case 4: Node *D* which is no *start node* and the outgoing transition set contains only output transitions.

Figure 4.23.: Cases to be considered for the current processed node during path counting.

Algorithm 8 shows how the cascades exceeding the *MaxPathCount* are computed. The algorithm defines the procedure *CHECK_CASCADES* which expects as input the set $\widehat{\mathfrak{T}}^*$, which is computed by Algorithm 7, and the *MaxPathCount* limit. The nodes of each IPT of the set $\widehat{\mathfrak{T}}^*$ are iterated in a *pre-order Depth First Search* style. Thus, on Line 7 an *pre-order*-based node set is created. The next step of the algorithm implies the computation of the paths for each IPT. In order to achieve that, for each node of the set $\mathfrak{N}^{DFS}$ the function **computePathCount(n)** is called. The function computes the count by which the path count *pathCount* will be incremented. Furthermore, all outgoing transitions of type unobservable, completion and output of the node are added to the set *cascade*. The set denotes the transitions of which the cascade consists of. After all nodes are checked the computed *pathCount* is checked against the limit *MaxPathCount*. In the event that the limit is exceeded the current cascade is added to the set *CascadeSet*, which denotes the set of cascades which have exceeded the

limit. Finally, all cascades which exceeds the limit are reported by calling procedure *REPORT()*.

Example 4.22 (Instantly Executable Cascades Path Calculation)
In this example the application of Algorithm 8 is shown by applying them to the IPT of Figure 4.21b. First, the nodes of the path tree are sorted by function *PRE_ORDER_DFS_SORTING*, such that $\mathfrak{N}^{DFS} = \{A2, A3, A5, A6, A8, A7, A4\}$. Next, for each node of set $\mathfrak{N}^{DFS}$ **computePathCount**(n) is called which results in which results in *5 paths*.

- **computePathCount**$(A2) = 2$ due to **isPathCountCase2**$(A2) = true$.
- **computePathCount**$(A3) = 0$ due to **isPathCountCase4**$(A3) = true$.
- **computePathCount**$(A5) = 2$ due to **isPathCountCase3**$(A5) = true$.
- **computePathCount**$(A6) = 1$ due to **isPathCountCase3**$(A6) = true$.
- **computePathCount**$(A8) = 0$ due to **nodeOutTrans**$(A8) = \emptyset$.
- **computePathCount**$(A7) = 0$ due to **nodeOutTrans**$(A7) = \emptyset$.
- **computePathCount**$(A4) = 0$ due to **nodeOutTrans**$(A4) = \emptyset$.

The computed paths are:

1. $A2 \rightarrow A3 \rightarrow A5$
2. $A2 \rightarrow A3 \rightarrow A5 \rightarrow A6$
3. $A2 \rightarrow A3 \rightarrow A5 \rightarrow A6 \rightarrow A8$
4. $A2 \rightarrow A3 \rightarrow A5 \rightarrow A7$
5. $A2 \rightarrow A4$

□

4.4. Metrics

This section introduces metrics addressing the complexity of an ESTS model in terms of data flow, control flow, structure and size. Based on this metrics the overall complexity of an ESTS model can be inferred.

In Section 4.4.1 some metrics with respect to the size of a model are introduced. A control flow complexity metric is discussed in Section 4.4.2. In Sections 4.4.3, 4.4.4, 4.4.5, 4.4.6 and 4.4.7 metrics in terms of data flow complexity are discussed. Finally, Section 4.4.8 discusses the complexity of guard expressions.

4.4.1. Size Metrics

Genero *et al.* [59] stated that the size of behavioral diagrams, in their concrete case UML State Machines, has influence on its understandability. On this account in the following section several metrics related to the size of an ESTS are defined.

Number of Transitions by Type (NTT) The NTT metric provides information about the total number of transitions with respect to their type. Table 4.4 states the formulas in order to calculate the metric.

Single ESTS	Set E of ESTSs	Description
$\mathrm{NTT}_{T_i} = \lvert T_i \rvert$	$\mathrm{NTT}^E_{T_i} = \sum_{e \in E} \mathrm{NTT}^e_{T_i}$	Total number of input transitions.
$\mathrm{NTT}_{T_o} = \lvert T_o \rvert$	$\mathrm{NTT}^E_{T_o} = \sum_{e \in E} \mathrm{NTT}^e_{T_o}$	Total number of output transitions.
$\mathrm{NTT}_{T_\gamma} = \lvert T_\gamma \rvert$	$\mathrm{NTT}^E_{T_\gamma} = \sum_{e \in E} \mathrm{NTT}^e_{T_\gamma}$	Total number of completion transitions.
$\mathrm{NTT}_{T_d} = \lvert T_d \rvert$	$\mathrm{NTT}{T_d}^E = \sum_{e \in E} \mathrm{NTT}^e_{T_d}$	Total number of delay transitions.

Table 4.4.: NTT Metrics Formulas

Number of Transitions (NT) The NT is defined as the total number of transitions of an ESTS given by $\mathrm{NT} = \lvert T \rvert$ or by $\mathrm{NT} = \mathrm{NTT}_{T_i} + \mathrm{NTT}_{T_o} + \mathrm{NTT}_{T_\gamma} + \mathrm{NTT}_{T_d}$. The total number of transitions over a set E of ESTSs is given by $\mathrm{NT}^E = \sum_{e \in E} \mathrm{NT}^e$.

Number of States (NS) The NS is defined as the total number of states of an ESTS given by $\mathrm{NS} = \lvert S \rvert$. The total number of transitions of a set E is given by $\mathrm{NS}^E = \sum_{e \in E} \mathrm{NS}^e$.

Number of Messages (NM) The NM metric represents the total number of incoming and outgoing messages and therefore is related to the number of elements in the input and output label sets. Thus, $\mathrm{NM} = \lvert L_i \rvert + \lvert L_o \rvert$.

Number of Transitions with a Guard (NTG) The NTG metric represents the number of transitions with a guard not always evaluating to *true*.

Number of Transitions with an Effect (NTE) The NTE metric indicates the number of transitions with an effect. A transition has an effect when its corresponding attribute update function ρ calculates a new attribute valuation ι' which is not equal to the previous valuation ι. Therefore, NTE is given by $\mathrm{NTE} = \lvert \{ t \in T \mid \iota \neq \iota' \} \rvert$. NTE over a set E of ESTSsis given by $\mathrm{NTE}^E = \sum_{e \in E} \mathrm{NTE}^e$.

Number of Attributes (NA) The NA metric depicts the number of attributes. Moreover, the metrics $\mathrm{NA}_{\mathbb{Z}}$, $\mathrm{NA}_{\mathbb{B}}$ and $\mathrm{NA}_{\mathbb{R}}$ are introduced depicting the number of attributes corresponding to the integer, Boolean and real number domain. The formulas in order to compute the metrics are stated in Table 4.5.

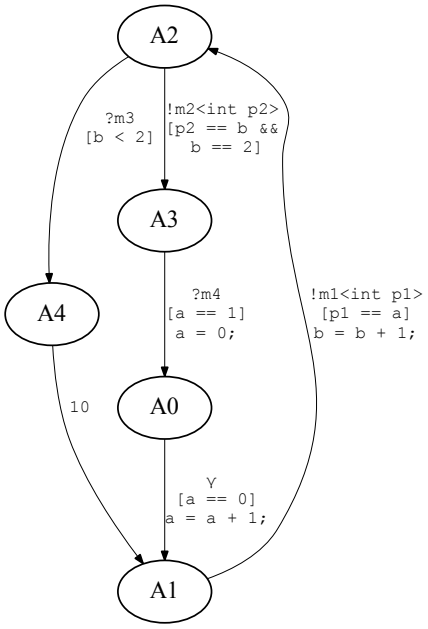


Figure 4.24.: Example of ESTS for which the size metrics in Table 4.6 are calculated.

Formula	Description
$\mathrm{NA} = \lvert A \rvert$	Total number of attributes.
$\mathrm{NA}_{\mathbb{Z}} = \lvert \{a \in A \mid \iota \in \mathfrak{U}^A \wedge \iota.v = a \wedge \iota.u = \mathbb{Z}\} \rvert$	Number of integer attributes.
$\mathrm{NA}_{\mathbb{B}} = \lvert \{a \in A \mid \iota \in \mathfrak{U}^A \wedge \iota.v = a \wedge \iota.u = \mathbb{B}\} \rvert$	Number of Boolean attributes.
$\mathrm{NA}_{\mathbb{R}} = \lvert \{a \in A \mid \iota \in \mathfrak{U}^A \wedge \iota.v = a \wedge \iota.u = \mathbb{R}\} \rvert$	Number of real number attributes.

Table 4.5.: NA Metrics Formulas

Example 4.23 (Size Metrics)

Based on Figure 4.24 the previous metrics defined are calculated whose results are given in Table 4.6. $\mathrm{NTT}_{T_i} = 2$ due to the input transitions $(A2 \xrightarrow{?m3} A4)$ and $(A3 \xrightarrow{?m4} A0)$, $\mathrm{NTT}_{T_o} = 2$ because of the output transitions $(A2 \xrightarrow{!m2} A3)$ and $(A1 \xrightarrow{!m1} A2)$. The ESTS only consists of a single completion transition $(A0 \xrightarrow{\gamma} A1)$ so that $\mathrm{NTT}_{T_\gamma} = 1$. $\mathrm{NTT}_{T_d} = 1$ due to the delay transition $(A4 \xrightarrow{10} A1)$. The NT metric can be calculated based on the previous NTT values and is therefore 6. Since the set of states S is $\{A0, A1, A2, A3, A4\}$, $\mathrm{NS} = 5$. As the delay transition $(A4 \xrightarrow{10} A1)$ is the only transition without a defined guard, which means the guard always evaluates to true, thus the NTG metric is 5. The ESTS in Figure 4.24 consists of three transitions which have an effect, since they have defined actions. Those transition are namely $(A3 \xrightarrow{?m4} A0)$ with action $a = 0$, $(A0 \xrightarrow{\gamma} A1)$ with action $a = a + 1$ and $(A1 \xrightarrow{?m4} A2)$ with action $b = b + 1$ Thus, $\mathrm{NTE} = 3$. In the model two attributes, namely a and b are used, both corresponding to the integer domain value. Therefore, $\mathrm{NA} = 2$,

$NA_{\mathbb{Z}} = 2$, $NA_{\mathbb{B}} = 0$ and $NA_{\mathbb{R}} = 0$. □

Metric	Value
NTT_{T_i}	2
NTT_{T_o}	2
NTT_{T_γ}	1
NTT_{T_d}	1
NT	6
NS	5
NM	4
NTG	5
NTE	3
NA	2
$NA_{\mathbb{Z}}$	2
$NA_{\mathbb{B}}$	0
$NA_{\mathbb{R}}$	0

Table 4.6.: Size metrics calculated for the ESTS which is depicted in Figure 4.24.

4.4.2. McCabes's Cyclomatic Complexity

Hereinafter, *McCabe's Cyclomatic Complexity*[61] is discussed which addresses the aspect of complexity of an ESTS in terms of execution paths. The metric originally measures the complexity of programs with respect to the *maximum number of linearly independent paths*. Therefore, McCabe associated a program to a directed graph, namely a Program Control Graph (PCG)[84]. The graph shows the characteristic that each node of the graph can be reached from the entry node and each node can reach the exit node.

$$V(G) = E - N + p + 1 \tag{4.31}$$

Equation 4.31 depicts the cyclomatic complexity formula according to McCabe. $V(G)$ denotes the cyclomatic complexity number of graph G, E the number of edges of graph G, N the number of nodes and p the number of components. McCabe stated that the cyclomatic complexity is equal to the maximum number of linearly independent circuits in a strongly connected graph. In case of a not strongly connected graph a virtual edge from the exit node to the entry node is added which is depicted by the additional $+1$ in the formula. Thus, in an already strongly connected graph the $+1$ can be neglected. In other words, cyclomatic complexity $V(G)$ calculates the minimum number of paths through a program to achive path coverage.

As an ESTS is a directed graph, Equation 4.31 can be simply applied to it as stated in Equation 4.32, whereas it is assumed that an ESTS is always a strongly connected graph.

$$V(e) = |T| - |S| + 1 \tag{4.32}$$

Considering Equation 4.31, E corresponds to the number of transitions T and N to the number of states. The parameter p is considered to be one since the metric is applied on a single ESTS.

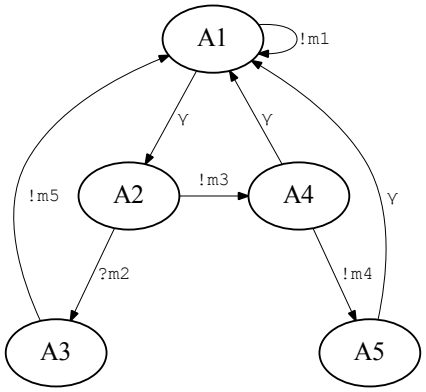


Figure 4.25.: Example of an ESTS in order to calculate cyclomatic complexity metric.

Example 4.24 (Cyclomatic Complexity)
Given the ESTS e, depicted in Figure 4.25, the following cyclomatic complexity is calculated by formula $V(e) = 8 - 5 + 1$, which results in $V(e) = 4$. One can choose the following linearly independent paths:

- $A1 \rightarrow A1$
- $A1 \rightarrow A2 \rightarrow A3 \rightarrow A1$
- $A1 \rightarrow A2 \rightarrow A4 \rightarrow A1$
- $A1 \rightarrow A2 \rightarrow A4 \rightarrow A5 \rightarrow A5$

□

In the context of an ESTS the McCabe's Cyclomatic Complexity (MCC) metric can provide information about the minimum number of test cases needed to accomplish full state and transition coverage.

4.4.3. Mean Attribute On Attribute Dependency Metric

This section introduces a metric which measures the degree an attribute depends on other attributes, called Mean Attribute On Attribute Dependency (MAAD) metric. Dependency means that the

value of a particular attribute is affected by the values of other attributes. The metric depicts the *mean value* over all dependency degrees of all attributes of an ESTS. The goal of this metric is to measure complexity in terms of data flow. Based on this metric it can be inferred that the higher the dependency degree of an attribute the complexer is the data flow with respect to this attribute.

Before the metric can be computed, the attribute dependencies of the attributes have to be determined. Thus, subsequently the dependency computation in terms of a single attribute is discussed. The process in order to determine all dependencies for a certain attribute is named hereinafter *dependency resolution*. Consider the expression $a = b + 1$, where a and b are attributes. Since for attribute a a new value is assigned, the sub-expression $b + 1$ is of interest as it denotes the actual computation of the new value for a. As the expression $b + 1$ contains a single attribute, namely b, a is considered to depend on attribute b. The previous example was rather simple. Therefore, consider a more complex example which is declared by the following expressions:

- $x = a + b$
- $a = c + d + 1$
- $b = 10$
- $c = d + 10$
- $d = 0$

In this example the dependency degree for attribute x is wanted. By looking at expression $x = a + b$, it can be seen that x depends on two other attributes, a and b. When examining expression $a = c + d + 1$ it can be determined that attribute a itself depends on two other attributes, namely c and d. These two attributes has an indirect affect on attribute x, as they affect attribute a directly, which in turn affects x directly. Thus, the entire attribute dependency hierarchy has to be resolved over all involved expressions. The dependence hierarchy for a certain variable can be depicted as a *Dependence Graph* as shown in Figure 4.26. The dependency computation for attribute x results in the set $\{a, b, c, d\}$ and thus x shows a dependency degree of 4.

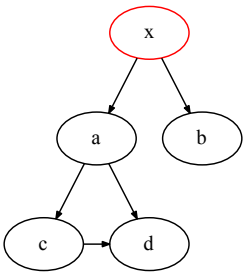


Figure 4.26.: Dependency graph for variable x based on expressions $x = a + b$, $x = c + d + 1$, $b = 10$, $c = d + 10$, $d = 0$

After the basic concept of attribute dependency resolution has been stated, the attribute dependency in terms of an ESTS is discussed next. Subsequently, it will be shown by means of a simple example

how the dependency resolution procedure is applied to an ESTS for a single attribute and which other concepts are involved.

Consider the simple ESTS in Figure 4.27. The labeling of the transitions in terms of transition types have been neglected. Only three transitions show a labeling, where each of the labels denote an action. Basically, attribute dependency resolution correlates with the analysis of data flow. Thus, one of the tools in the field of data flow analysis is used, namely the *Def-Use Data Structure*. On that account the first step involves the creation of the *Def-Use Data Structure*. This is carried out by iterating over the set of transitions T and extracting all assigned as well as referenced attributes. With reference to the example ESTS in Figure 4.27 the following sets *Defs* and *Uses* are created:

- As the ESTS contains three actions, three *def* entries are created, such that $Defs = \{def_0,\ def_1,\ def_2\}$ with:
  - $def_0 = (a,\ (A \rightarrow B),\ a = 10,\ \{use_0,\ use_1\})$
  - $def_1 = (b,\ (B \rightarrow C),\ b = a,\ \{use_2\})$
  - $def_2 = (x,\ (C \rightarrow D),\ x = a + b,\ \emptyset)$
- The *use* entries use_0, use_1 and use_2 are created, since attribute a is referenced two times and attribute b one time, such that $Uses = \{use_0,\ use_1,\ use_2\}$, with:
  - $use_0 = (a,\ (B \rightarrow C),\ b = a,\ \{def_0\})$
  - $use_1 = (a,\ (C \rightarrow D),\ x = a + b,\ \{def_0\})$
  - $use_2 = (b,\ (C \rightarrow D),\ x = a + b,\ \{def_1\})$

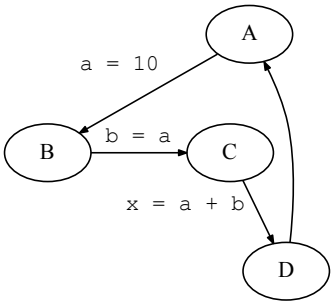


Figure 4.27.: Simple example depicting attribute dependencies in terms of an ESTS.

Based on the calculated *Def-Use* data the dependencies for attribute x of expression $x = a + b$ can be illustrated as a dependency graph as well, as shown in Figure 4.28. The dependency graph starts at def_2 entry which corresponds to expression $x = a + b$ and thus represents the definition of attribute x. The graph nodes consist of *def* and *use* entry nodes. A *def* node can only have an outgoing arrow to a *use* node. In contrast a *use* node can only have an outgoing arrow to a *def* node. The $def \rightarrow use$ relation, which is named *referencing*-relation, models the relation between the defined attribute and its referenced attributes and therefore as a further consequence to all attributes it directly depends on. Due to the fact that a referenced attribute can depend on other attributes as

well, the corresponding definitions of the attribute must be considered as well. This is depicted by the *use* $\rightarrow$ *def* relation, namely a *is defined by*-relation. All attribute dependencies can be extracted from the dependency graph by considering all *def* nodes which represent the definition of each directly or indirectly referenced attributes. For the concrete example this leads to the result that attribute x depends on a and b.

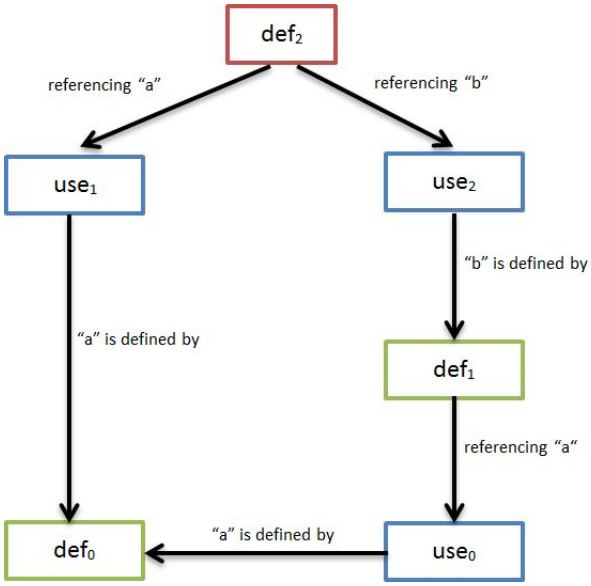


Figure 4.28.: *Def-Use* dependency graph for attribute x of expression $x = a + b$ of the ESTS depicted in Figure 4.27.

Algorithm 9 shows the function *COMPUTE_DEPENDENCIES* in order to calculate the dependencies for a single attribute. The algorithm follows the basic concept of a dependency graph, but without actually building a graph, as the algorithm traverses a virtual dependency graph by means of the *Def-Use* data in a *depth-first search*[85] style. The function expects as input the attribute a for which the dependencies should be computed as well as the sets *Defs* and *Uses*, and returns the set $A_{dependency}$, which comprises all attributes on which the given attribute depends.

The algorithm starts with the initialization of the data structures *stack* and $Defs_{visited}$. The former is a stack which contains all elements, either a *def* or a *use*, which are processed next. The latter denotes a set of definition entries which has already been processed and thus are considered being *visited*, in the context of a dependency graph. The actual dependency computation starts now with retrieving all definition entries from the set *Defs* which belong to the given attribute as shown in the for-loop condition. These definitions are considered as start definitions def_{start} and denote those definitions where the dependency resolution commences. In the context of a dependency graph a def_{start} corresponds to the start node. Thus, the start definition def_{start} is pushed onto the stack *stack*.

The while-loop depicts the actual traversing of the virtual dependency graph. The while-loop is executed as long as the stack is not empty. During each while-loop iteration the top element of the stack is retrieved which is assigned to the variable *element*. If the element is a *use* entry, such that *element* $\in$ *Uses*, all referenced definition entries are pushed onto the stack, which is carried

Algorithm 9 Dependency Calculation for a Single Attribute

```
1:  function COMPUTE_DEPENDENCIES(a ∈ A, Defs, Uses) → A^a_dependency
2:      stack ← ∅;
3:      Defs_visited ← ∅;
4:      for all def_start ∈ Defs with def_start.a == a do
5:          stack.PUSH(def_start);
6:          while stack ≠ ∅ do
7:              element ← stack.POP();
8:              if element ∈ Uses then
9:                  stack.PUSHALL(element.RefDefs);
10:             else
11:                 if element ≠ def_start ∧ element.a ∉ A_dependency then
12:                     A^a_dependency ← A^a_dependency ∪ element.a;
13:                 end if
14:                 if element ∉ Defs_visited then
15:                     A_refs ← extractAttributeReferences(element.exp);
16:                     for all a_ref ∈ A_refs do
17:                         use ← getUseEntry(a_ref, element.t, element.exp);      ▷ Applying Equation ??
18:                         if use ≠ NIL then
19:                             stack.PUSH(use);
20:                         end if
21:                     end for
22:                 end if
23:                 Defs_visited ← Defs_visited ∪ element;
24:             end if
25:         end while
26:     end for
27:     return A^a_dependency;
28: end function
```

out by statement $stack.pushAll(element.RefDefs)$. This case depicts in the context of a dependency graph the creation of a *referencing*-relation. If the *element* is a *def* entry, such that $element \in Defs$, the following steps are applied:

1. Add the attribute which is defined by the *element*, with $element \in Defs$, to the $A_{dependency}$ set, but only if the element is not equal to the start definition def_{start}, such that $element \neq def_{start}$, and the corresponding attribute of the element is not yet in the set, so that $element.a \notin A_{dependency}$.
2. If the definition, which is represented by *element*, has not yet been visited, such that $element \notin Defs_{visited}$, the following steps are applied:
   a) Extract all attributes of the corresponding expression $element.exp$ by means of function **extractAttibuteReferences**$(element.exp)$ which results in set A_{refs}.
   b) For each attribute $a_{ref} \in A_{refs}$ the corresponding *use* entries are looked up by means of function **filterUses**$(a_{ref}, element.t, element.exp)$, which should either return a *use* entry or *NIL* if no such entry could be found in the set *Uses*. If a *use* entry is present, such that $use \neq NIL$, this entry is pushed onto the stack, which corresponds to the *is defined by* relation in context of a dependency graph.
   c) In the last step of handling a *def* entry, the entry is added to the $Defs_{visited}$ set which prevents the algorithm to handle definitions multiple times.

Once the dependencies are calculated, the MAAD metric and the corresponding standard deviation value are computed. This is accomplished by computing for all attributes of an ESTS the

attribute dependencies. Thus, let e be an ESTS with attributes $A = \{a_0, a_1, ..., a_n\}$, then $A^*_{dependency} = \bigcup_{a \in A} COMPUTE\_DEPENDENCIES\ (a, Defs, Uses)$ denotes the set of all calculated dependencies of all attributes of ESTS e, such that $A^*_{dependency} = \{A^{a_0}_{dependency}, A^{a_1}_{dependency}, ..., A^{a_n}_{dependency}\}$. In order to calculate the mean and standard deviation for each subset of the set $A^*_{dependency}$, their cardinality have to be calculated by means of Equation 4.12, such that $X = \textbf{cardinality}(A^*_{dependency})$. Based on set X the mean is calculated as $\mu = \textbf{mean}(X)$ and the standard deviation as $\sigma = \textbf{standardDeviation}(X, \mu)$.

Example 4.25 (Dependency, Mean and Standard Deviation Computation)
By means of the example ESTS of Figure 4.27 the application of Algorithm 9 is shown. Moreover, based on the computed dependencies the metrics are calculated.

The given ESTS consist of three attributes, such that $A = \{a, b, x\}$. Thus, the algorithm computed the dependency sets $A^*_{dependency} = \{A^a_{dependency}, A^b_{dependency}, A^x_{dependency}\}$ with $A^a_{dependency} = \emptyset$, $A^b_{dependency} = \{a\}$ and $A^x_{dependency} = \{a, b\}$. Hence, the algorithm is demonstrated for attribute x.

1. Execute algorithm with $COMPUTE\_DEPENDENCIES(x, Defs, Uses)$.
2. Determine start definition $def_{start} = def_2$ which is pushed onto $stack$, such that $stack = \{def_2\}$.
3. Retrieve top element of $stack$, such that $element = def_2$, $stack = \emptyset$.
4. Current element $element$ is a definition entry which belongs to the start definition def_{start}. Thus, set $A_{dependency}$ remains unchanged.
5. The corresponding expression $x = a + b$ shows 2 attribute dependencies, such that **extractAttributeReferences**$(x = a + b)$ returns the set $A_{ref} = \{a, b\}$ which belong to use_1 and use_2, respectively. The entries use_1 and use_2 are pushed onto $stack$, such that $stack = \{use_1, use_2\}$.
6. Current element $element$ is considered as being visited and thus added to set $Defs_{visited}$, so that $Defs_{visited} = \{def_2\}$.
7. Retrieve top element from stack such that $element = use_1$ and $stack = \{use_2\}$.
8. Push all definitions on which use_1 depends onto $stack$, such that $stack = \{def_0, use_2\}$.
9. Retrieve top element from stack, such that $element = def_0$ and $stack = \{use_2\}$.
10. Current element does not belong to start definition, wherefore a dependency is reported such that $A_{dependency} = \{a\}$.
11. The expression $a = 10$, corresponding to def_0, shows no attribute dependencies and therefore **extractAttributeReferences**$(a = 10)$ returns NIL, so that no element is pushed onto stack.
12. Current element $element$ is considered as being visited and thus added to set $Defs_{visited}$, so that $Defs_{visited} = \{def_2, def_0\}$.
13. Retrieve top element from stack such that $element = use_2$ and $stack = \emptyset$.
14. Push all definitions on which use_2 depends onto $stack$, such that $stack = \{def_1\}$.
15. Retrieve top element from stack, such that $element = def_1$ and $stack = \emptyset$.
16. Current element does not belong to start definition, wherefore a dependency is reported such that $A_{dependency} = \{a, b\}$.
17. The expression $b = a$, corresponding to def_1, shows a single attribute dependency and therefore **extractAttributeReferences**$(b = a)$ returns the set $A_{ref} = \{a\}$ which belongs to use_0. The entry use_0 is pushed onto stack, such that $stack = \{use_0\}$
18. Retrieve top element from stack such that $element = use_0$ and $stack = \emptyset$.

19. Push all definitions on which use_0 depends onto *stack*, such that $stack = \{def_0\}$.
20. Retrieve top element from stack, such that $element = def_0$ and $stack = \emptyset$.
21. Current element does not belong to start definition, wherefore a dependency is reported which actually is already comprised in the set $A_{dependency} = \{a, b\}$.
22. As $def_0 \in Defs_{visited}$ no further actions are carried out.
23. Since the *stack* remains empty dependency computation finished with result $A_{dependency} = \{a, b\}$.

Based on set $A^*_{dependency}$ the cardinality set $X = \{0, 1, 2\}$ is calculated. Thus the mean is computed as $\mu = \textbf{mean}(X)$ which results in $\mu = 1$ and the standard deviation as $\sigma = \textbf{standardDeviation}(X, \mu)$ which results in $\sigma = 1,291$. Thus, the metric is denoted as MAAD $= 1 \pm 1$. □

4.4.4. Mean Attribute On Transition Read Dependency and Mean Attribute on Transition Write Dependency Metrics

This section introduces the metrics Mean Attribute On Transition Read Dependency (MATRD) and Mean Attribute On Transition Write Dependency (MATWD). The aim of these two metrics is to measure the degree attributes depend on transitions in terms of *read* and *write* operations which are called from here on *read dependencies* and *write dependencies*. While the MATRD metric depicts the mean value over all attribute dependencies to transitions which are referencing and reading the attribute, respectively. The MATWD metric depicts the mean value over all attributes dependencies to transitions which are modifying the attribute's value.

Algorithm 10 shows the calculation of the *read* and *write* dependencies. The algorithm expects as input the set of attributes A and the *Def-Use* data in terms of the definition set *Defs* and the usage set *Uses*. The result of the algorithm are the sets T^*_{read} and T^*_{write}. These sets consist of subsets containing the transitions which show either a read, corresponding to set T^*_{read}, or write dependency corresponding to set T^*_{write}. A single subset corresponds to a certain attribute, so that the sets are defined as $T^*_{read} = \{T^{a_0}_{read}, T^{a_1}_{read}, ..., T^{a_n}_{read}\}$ and $T^*_{write} = \{T^{a_0}_{write}, T^{a_1}_{write}, ..., T^{a_n}_{write}\}$ with $a_0, a_1, ..., a_n \in A$ and $n = |A|$.

Algorithm 10 Transition Read And Write Dependency Computation

```
1:  function COMPUTE_DEPENDENCIES(A, Defs, Uses) → T*_read, T*_write
2:      for all a ∈ A do
3:          T^a_read ← {use.t | use ∈ Uses ∧ use.a = a};
4:          T*_read ← T*_read ∪ {T^a_read};
5:
6:          T^a_write ← {def.t | def ∈ Defs ∧ def.a = a};
7:          T*_write ← T*_write ∪ {T^a_write};
8:      end for
9:      return T*_read, T*_write;
10: end function
```

The algorithm computes in a for-loop for each attribute of the attribute set the read and write dependencies by means of the *Uses* and *Defs*, respectively. This is carried out in terms of read dependencies by retrieving all *use* entries of the set *Uses* which belongs to a given attribute a, such

that $use.a == a$. The result is the a set of transitions, T^a_{read}, corresponding to those *use* entries. Write dependencies for a certain attribute a are determined similar but by considering the *Defs* set and by retrieving those *def* entries which belongs to a, such that $def.a = a$. The result is a set of transitions, T^a_{write}, corresponding to those *def* entries.

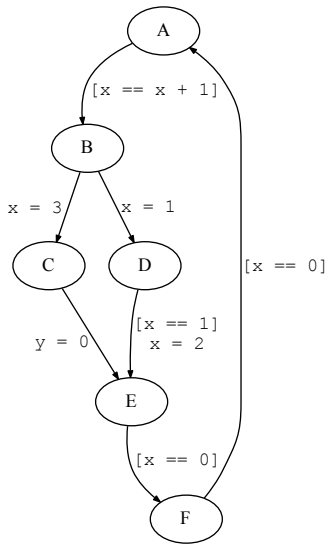


Figure 4.29.: Example of an ESTS to demonstrate computation of *read* and *write* dependencies in terms of transitions.

Based on the computed sets T^*_{read} and T^*_{write} the metrics are calculated. Beforehand, the sets X_{read} and X_{write} must be computed which are defined as $X_{read} = \textbf{cardinality}(T^*_{read})$ and $X_{write} = \textbf{cardinality}(T^*_{write})$. As a further consequence the mean values are calculated by $\mu_{read} = \textbf{mean}(X_{read})$ and $\mu_{write} = \textbf{mean}(X_{write})$. The corresponding standard deviations are defined as $\sigma_{read} = \textbf{standardDeviation}(X_{read}, \mu_{read})$ and $\sigma_{write} = \textbf{standardDeviation}(X_{write}, \mu_{write})$. Thus, MATRD $= \mu_{read} \pm \sigma_{read}$ and MATWD $= \mu_{write} \pm \sigma_{write}$.

Example 4.26 (Dependency, Mean and Standard Deviation Computation)
Consider Figure 4.29 which is used in this example to demonstrate the computation of the MATRD and MATWD metrics. The attribute set is defined as $A = \{x, y\}$, whereas the sets *Defs* and *Uses* are taken over from the Example 4.4.

Applying Algorithm 10 results in the following transition sets for each attribute of set A.

- $T^x_{read} = \{(A \to B), (D \to E), (E \to F), (F \to A)\}$
- $T^x_{write} = \{(A \to B), (B \to C), (B \to D), (D \to E)\}$
- $T^y_{read} = \emptyset$
- $T^y_{write} = \{(C \to E)\}$

The algorithm returns the result sets $T^*_{read} = \{T^x_{read}, T^y_{read}\}$ and $T^*_{write} = \{T^x_{write}, T^y_{write}\}$. In order to calculate the metrics first the sets X_{read} and X_{write} are created. The calculation of the metrics

are based on the sets $X_{read} = \{4, 0\}$ and $X_{write} = \{4, 1\}$ and thus result in MATRD $= 2 \pm 2$ and MATWD $= 2,5 \pm 1,5$. □

4.4.5. Mean Attribute Def-Use Distance Metric

The Mean Attribute Def-Use Distance (MADUD) metric measures the mean distance over all *def-use* pairs. The computation of the *def-use* pair distances are carried out for each attribute. In other words, the distances are calculated between a *def-use* entry pair where both entries correspond to the same attribute. This fact is shown in Algorithm 11 which determines all required distances and computes the mean value over the collected distances.

Algorithm 11 Mean Distance of Attribute *Def-Use* Pairs

```
1:  function MEAN_DISTANCE_DEFINITION_USE(A, Defs, Uses) → μ, σ
2:      Dist ← ∅;
3:      for all a ∈ A do
4:          for all def ∈ {def ∈ Defs | def.a == a} do
5:              for all use ∈ {use ∈ Uses | use.a == a} do
6:                  dist ← shortestDistance(def.t, use.t);          ▷ Application of Equation 4.7
7:                  if dist ≠ NIL then
8:                      Dist ← Dist ∪ dist;
9:                  end if
10:             end for
11:         end for
12:     end for
13:     μ ← mean(Dist);                                             ▷ Application of Equation 4.10
14:     σ ← standardDeviation(Dist, μ);                             ▷ Application of Equation 4.11
15:     return μ, σ;
16: end function
```

The algorithm expects as input the set of attributes A as well as the sets *Defs* and *Uses*, which are representing the sets of all definitions (defs) and usages (uses), respectively. The output of the algorithm is the mean value μ and the standard deviation σ. The algorithm iterates over the attribute set and retrieves first of all the *def* entries from set *Defs*, which correspond to the current attribute a. For each of these definitions the distance to all *use* entries, which correspond to the current attribute a, are determined by means of function **shortestDistance**(t, t'), as stated in Equation 4.7. The collected distances are represented by the set *Dist* whose elements are integer values. Finally, based on this set the mean value and the standard deviation are calculated.

Example 4.27 (Mean *Def-Use* Distance)
Consider Figure 4.29 which is used in this example to demonstrate the computation of the MADUD metric. The attribute set is defined as $A = \{x, y\}$, whereas sets *Defs* and *Uses* are take over from Example 4.4. It is assumed that the shortest distances have been determined beforehand for this example.

Applying Algorithm 11 results in the following *def-use* pair distances.

- Determine distances for *def-use* pairs of attribute x.
  - Distance for def_0-use_0 is 4 $\rightarrow$ $Dist = \{4\}$.
  - Distance for def_0-use_1 is 6 $\rightarrow$ $Dist = \{4, 6\}$.

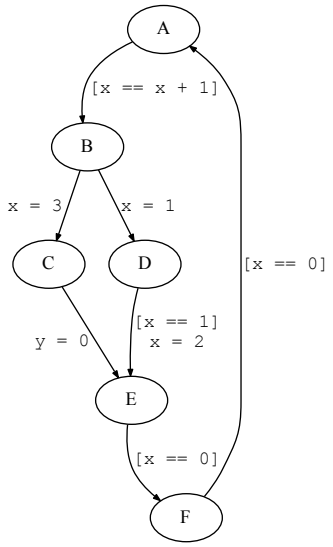


Figure 4.30.: Example of an ESTS to demonstrate computation of MADUD metric.

  - Distance for def_0-use_2 is 2 $\rightarrow$ $Dist = \{4, 6, 2\}$.
  - Distance for def_0-use_3 is 3 $\rightarrow$ $Dist = \{4, 6, 2, 3\}$.
  - Distance for def_1-use_0 is 4 $\rightarrow$ $Dist = \{4, 6, 2, 3, 4\}$.
  - Distance for def_1-use_1 is 1 $\rightarrow$ $Dist = \{4, 6, 2, 3, 4, 1\}$.
  - Distance for def_1-use_2 is 2 $\rightarrow$ $Dist = \{4, 6, 2, 3, 4, 1, 2\}$.
  - Distance for def_1-use_3 is 3 $\rightarrow$ $Dist = \{4, 6, 2, 3, 4, 1, 2, 3\}$.
  - Distance for def_3-use_0 is 3 $\rightarrow$ $Dist = \{4, 6, 2, 3, 4, 1, 2, 3, 3\}$.
  - Distance for def_3-use_1 is 5 $\rightarrow$ $Dist = \{4, 6, 2, 3, 4, 1, 2, 3, 3, 5\}$.
  - Distance for def_3-use_2 is 1 $\rightarrow$ $Dist = \{4, 6, 2, 3, 4, 1, 2, 3, 3, 5, 1\}$.
  - Distance for def_3-use_3 is 2 $\rightarrow$ $Dist = \{4, 6, 2, 3, 4, 1, 2, 3, 3, 5, 1, 2\}$.
- No distances are calculated for attribute y, since no *use* entries are present.

The computed mean value $\mu = 3$ and standard deviation $\sigma = 1,472$ and thus the metric MADUD $= 3 \pm 1,472$. □

4.4.6. Mean Output To Input Transition Dependency Metric

The following metric addresses the measurement of the degree that an output transition depends on input transitions. The Mean Output To Input Transition Dependency (MOITD) denotes the mean dependency value over all transitions of an ESTS.

An output to input dependency is determined by considering the output and input parameters, respectively, of the transitions as well as the assigned and references attributes. In terms of the

output parameter all attributes affecting the parameter are determined. As by definition the output parameter can be set and thus affected by means of an equality guard expression, the guards of an output transition are considered. By means of these attributes the connection to the input transition is established by considering all input transitions where these attributes are re-assigned in an action, and beyond that an input parameter has an affect of the re-assigned value of the attribute. Thus, to establish the output to input dependency, the data flow of the parameters and attributes have to be taken into account.

The data flow of the attributes as well as the parameters can be depicted with the aid of the *Def-Use* data structure. Hereinafter, the dependency computation approach for a single output transition t_o is discussed. For this purpose the guard φ, with $\varphi \in \mathfrak{F}(V)$, of an output transition and the action ρ, with $\rho \in \mathfrak{A}(V)^A$, are depicted in the BET structure, with $exp_\varphi = \textbf{bet}(\varphi)$ and the $exp_\rho = \textbf{bet}(\rho)$, respectively.

The computation consists of the following steps and are depicted in Algorithm 12:

1. Retrieve all sub-expressions $exp'_\varphi \in EXP'_\varphi$, with EXP'_φ denoting the set of all sub-expressions, of the guard $t_o.\varphi$ where the following constraints apply:
   - The operator of the sub-expression must correspond to the equality operator, such that $\text{exp}'_\varphi.op \in \{==\}$.
   - Either the left or the right operand must correspond to a parameter valuation of the output transition message. Thus, $exp'.opd_l \in \mathfrak{U}^{\textbf{par}(l)} \vee exp'.opd_r \in \mathfrak{U}^{\textbf{par}(l)}$ with $t_o.l = l$.

   The extraction is carried out by the use of function **extractSubExpression**(exp).
2. For each sub-expression $exp'_\varphi \in EXP'_\varphi$ extract the referenced attributes. Thus, $A_{refs} = \bigcup_{exp'_\varphi \in EXP'_\varphi}$ **extractAttributeReferences**(exp'_φ).
3. For each referenced attribute, with $a_{ref} \in A_{refs}$, determine the *use* entry which corresponds to the referenced attribute a_{ref}, output transition t_o and to the guard exp_φ. This step results in the set $Uses'$ which holds all *use*entries of the referenced attributes. Thus, $Uses' = \bigcup_{a_{ref} \in A_{refs}} \{use \in Uses \mid use.a = a_{ref} \wedge use.t = t_o \wedge use.exp = exp_\varphi\}$.
4. For each $use \in Uses'$ the corresponding definition reference set *RefDefs* is considered, but only those definition references which correspond to an input transition. Thus, $Defs' = \bigcup_{use \in Uses'} \{def \in use.RefDefs \mid def.t \in T_i\}$, with set $Defs'$ denoting all definitions which define referenced attributes and which belong to an input transition.
5. For each definition $def \in Defs'$ determine the parameters which are referenced by the corresponding action expression $def.exp$ by means of function $P_{refs} =$ **extractParameterReferences**($def.exp$). The result of the function is the set P_{refs}, which contains all referenced parameters belonging to the input transition. Thus, $P_{refs} \subseteq \textbf{par}(l)$ with $def.t.l = l$. If the set P_{refs} is not empty, such that $P_{refs} \neq \emptyset$, the new value of the attribute is affected by an input parameter and as a further consequence affects the output parameter. For this reason the corresponding transition is added to the set $T^{t_o}_{dependency}$ which denotes the set of input transitions on which the output transition t_o depends.

Since the MOITD denotes the mean dependency of all output transitions of an ESTS, Algorithm 12 is applied to each output transition. Thus, let E be an ESTS with output transitions $T_o =$

Algorithm 12 Output to Input Dependency Computation

```
1:  function OUTPUT_INPUT_DEPENDENCY(t_o ∈ T_o, Defs, Uses) → T^{t_o}_dependency
2:      T^{t_o}_dependency ← ∅;
3:      exp_φ ← bet(t_o.φ);
4:      EXP'_φ ← extractSubExpressions(exp);
5:
6:      A_refs ← ∅;
7:      for all exp'_φ ∈ EXP'_φ do
8:          A_refs ← A_refs ∪ extractAttributeReferences(exp');
9:      end for
10:
11:     Uses' ← ∅;
12:     for all a_ref ∈ A_refs do
13:         Uses' ← Uses' ∪ {use ∈ Uses | use.a = a_ref ∧ use.t = t_o ∧ use.exp = exp_φ};
14:     end for
15:
16:     Defs' ← ∅;
17:     for all use ∈ Uses' do
18:         Defs' ← Defs' ∪ {def ∈ use.RefDefs | def.t ∈ T_i};
19:     end for
20:
21:
22:     for all def ∈ Defs' do
23:         P_refs ← extractParameterReferences(def.exp);
24:         if P_refs ≠ ∅ then
25:             T^{t_o}_dependency ← T^{t_o}_dependency ∪ def.t;
26:         end if
27:     end for
28: end function
```

$\{t_{o_0}, t_{o_1}, ..., t_{o_n}\}$, then the set $T^*_{dependency}$ denotes the set of all computed output to input dependencies, with $T^*_{dependency} = \bigcup_{t \in T_o} OUTPUT\_INPUT\_DEPENDENCY(t, Defs, Uses)$, such that $T^*_{dependency} = \{T^{t_{o_0}}_{dependency}, T^{t_{o_1}}_{dependency}, ..., T^{t_{o_n}}_{dependency}\}$.

In order to calculate the mean value as well as the standard deviation the cardinality of each subset of set $T^*_{dependency}$ is computed by applying Equation 4.12. Therefore, $X = \textbf{cardinality}(T^*_{dependency})$. Finally, the functions defined in Equations 4.10 and 4.11 are applied.

Example 4.28 (Mean Output To Input Dependency)

Consider Figure 4.31 which is used in this example to demonstrate the computation of the MOITD metric. The labels of the transitions denote either an input or output transition. While an input transition is prefixed by "?", an output transition is prefixed by "!". The variable definitions in angle brackets depict the input and output parameters, respectively. With respect to the example ESTS, the set of output transitions is $T_o = \{t_{o_{(m3)}}, t_{o_{(m5)}}\}$.

The *Def-Use* data structure looks as follows:

- $Defs = \{def_0, def_1, def_2\}$ with:
  - $def_0 = \{a,\ (A \xrightarrow{?m1} B),\ a = p0,\ \{use_0\}\}$
  - $def_1 = \{a,\ (B \xrightarrow{?m4} C),\ a = p0,\ \{use_0\}\}$
  - $def_2 = \{b,\ (C \xrightarrow{?m2} D),\ b = 20 + p0,\ \{use_1\}\}$
- $Uses = \{use_0, use_1\}$ with:

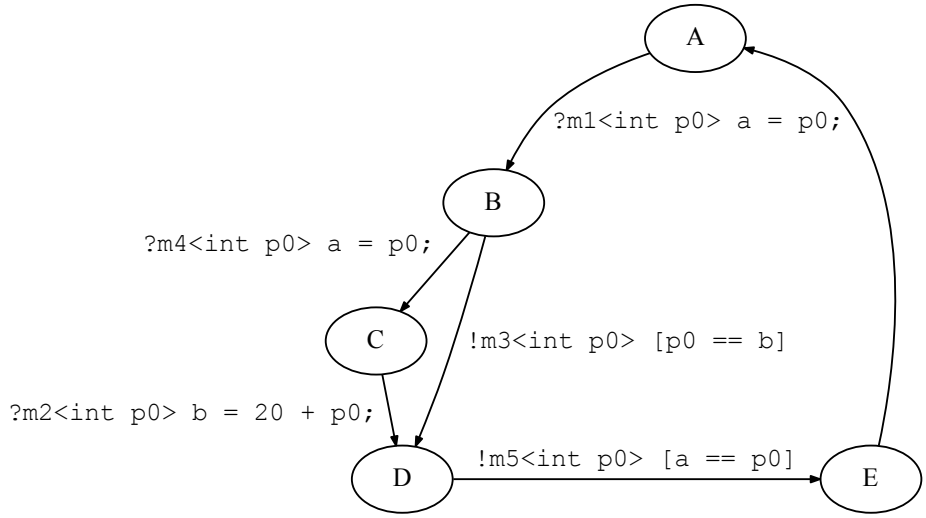


Figure 4.31.: Example of an ESTS to demonstrate the computation of the MOITD metric.

- $use_0 = \{a,\ (D \xrightarrow{!m5} E),\ a == p0,\ \{def_0, def_1\}\}$
- $use_1 = \{b,\ (B \xrightarrow{!m3} D),\ p0 == b,\ \{def_2\}\}$

Applying Algorithm 12 to each transition of the set T_o leads to the following results which are denoted step by step hereinafter:

1. Applying the algorithm to output transition $t_{o_{(m3)}} = (B \xrightarrow{!m3} D)$.
   a) $EXP'_{\varphi} = \{(p0 == b)\}$
   b) $A_{refs} = \{b\}$
   c) $Uses' = \{use_1\}$
   d) $Defs' = \{def_2\}$
   e) $P_{refs} = \{p0\}$. Since $P_{refs} \neq \emptyset$, the corresponding input transition is added to set $T^{t_{o_{(m3)}}}_{dependency}$, such that $T^{t_{o_{(m3)}}}_{dependency} = \{t_{i_{(m2)}}\}$
2. Applying the algorithm to output transition $t_{o_{(m5)}} = (D \xrightarrow{!m5} E)$.
   a) $EXP'_{\varphi} = \{(a == p0)\}$
   b) $A_{refs} = \{a\}$
   c) $Uses' = \{use_0\}$
   d) $Defs' = \{def_0, def_1\}$
   e) Processing def_0 results in: $P_{refs} = \{p0\}$, and since $P_{refs} \neq \emptyset$, the corresponding input transition is added to set $T^{t_{o_{(m5)}}}_{dependency}$, such that $T^{t_{o_{(m5)}}}_{dependency} = \{t_{i_{(m1)}}\}$.
   f) Processing def_1 results in: $P_{refs} = \{p0\}$, and since $P_{refs} \neq \emptyset$, the corresponding input transition is added to set $T^{t_{o_{(m5)}}}_{dependency}$, such that $T^{t_{o_{(m5)}}}_{dependency} = \{t_{i_{(m1)}}, t_{i_{(m4)}}\}$.

Thus, the set $T^{*}_{dependency} = \{T^{t_{o(m3)}}_{dependency}, T^{t_{o(m5)}}_{dependency}\}$ and applying **cardinality**$(T^{*}_{dependency})$ results in $X = \{1, 2\}$. Based on set X the computed mean value $\mu = 1,5$ and standard deviation $\sigma = 0,5$ and thus the metric MOITD $= 1,5 \pm 0,5$. □

4.4.7. Mean Output To Input Transition Dependency Distance Metric

This metric measure the mean distance between the output transitions and their dependent input transitions. The computation is based on the MOITD metric since the Algorithm 12 is used to resolve the output to input transition dependencies as shown in Algorithm 13. The algorithm computes the mean distance for all output transitions. Thus, the following steps are applied:

1. For each output transition $t_o \in T_o$ apply Algorithm 12 to calculate the input dependencies.
2. Compute the distances for each output transition to the dependent input transitions by means of **shortestDistance**(t_i, t_o) as defined in Equation 4.7, with t_i denoting the input transition on which t_o depends.
3. Calculate the mean value μ and standard deviation σ, which depict the output of Algorithm 13.

Algorithm 13 Mean Output to Input Transition Dependency Distance

```
1:  function MEAN_DISTANCE(T_o, Defs, Uses) → dist
2:      Dist ← ∅;
3:      for all t_o ∈ T_o do
4:          T^{t_o}_dependency ← OUTPUT_INPUT_DEPENDENCY(t_o, Defs, Uses);   ▷ Application of Algorithm 12
5:          for all t_i ∈ T^{t_o}_dependency do
6:              dist ← shortestDistance(t_i, t_o);                           ▷ Application of Equation 4.7
7:              if dist ≠ NIL then
8:                  Dist ← Dist ∪ dist;
9:              end if
10:         end for
11:     end for
12:     μ ← mean(Dist);                                                      ▷ Application of Equation 4.10
13:     σ ← standardDeviation(Dist, μ);                                      ▷ Application of Equation 4.11
14:     return μ, σ;
15: end function
```

Example 4.29 (Mean Distance of Output To Input Dependencies)
Applying Algorithm 13 to the ESTS of Figure 4.31 leads to the distance set $Dist = \{2, 2, 4\}$. The distances of the set $Dist$ are related to the distances of the following transitions:

- Distance from $(A \xrightarrow{?m1} B)$ to $(D \xrightarrow{!m5} E) \rightarrow 2$.
- Distance from $(B \xrightarrow{?m4} C)$ to $(D \xrightarrow{!m5} E) \rightarrow 2$.
- Distance from $(C \xrightarrow{?m1} D)$ to $(B \xrightarrow{!m3} D) \rightarrow 4$.

Based on set $Dist$ the computed mean value $\mu = 2,6667$ and standard deviation $\sigma = 0,953$ and thus the metric MOITD $= 2,6667 \pm 0,953$. □

4.4.8. Mean Guard Complexity Metric

The Mean Guard Complexity (MGC) metric depicts the mean complexity degree of all guard expressions of an ESTS. The complexity degree of a single guard is related to the count of the used logical operators "&&" and "||". The approach to measure complexity in terms of logical operator count has been taken over from the static analysis tool *Checkstyle*[12], which determines complex Boolean expressions in *Java* source code by this means.

First, in order to calculate the metric for each guard of an ESTS the complexity degree has to be determined. This is accomplished by means of Algorithm 14, which expects as input a single guard φ and returns the complexity degree $complexityDegree$, with $complexityDegree \in \mathbb{N}_0$, for the given guard. The algorithm handles the given guard as BET exp as defined in Definition 4.1. Initially, the guard expression is pushed onto the stack $stack$, which holds all expressions which should be processed, with the next expression to be processed corresponding to the top element of the stack. The while-loop is executed as long as the $stack$ is not emptyl. In each while-loop iteration the following steps are carried out:

- Retrieve the top element from $stack$, which is the current expression exp to be processed.
- If the operator of the current expression corresponds to a logical expression, the complexity counter will be incremented.
- Push the left operand opd_l of the current expression onto stack, but only if the left operand is neither a variable valuation nor a literal.
- Push the right operand opd_r of the current expression onto stack, but only if the left operand is neither a variable valuation nor a literal.

Algorithm 14 Determination of Guard Complexity

```
1:  function GUARD_COMPLEXITY(φ) → complexityDegree
2:      complexityDegree ← 0;
3:      stack.PUSH(bet(φ));
4:      while stack ≠ ∅ do
5:          exp ← stack.POP();
6:          if exp.op ∈ LogicalOp then
7:              complexityDegree ← complexityDegree + 1;
8:          end if
9:          if exp.opd_l ≠ NIL ∧ exp.opd_l ∉ {c, 𝔘^V} then
10:             stack.PUSH(exp.opd_l);
11:         end if
12:         if exp.opd_r ≠ NIL ∧ exp.opd_r ∉ {c, 𝔘^V} then
13:             stack.PUSH(exp.opd_r);
14:         end if
15:     end while
16:     return complexityDegree;
17: end function
```

In order to calculate the MGC metric the mean over the complexity of all guards has to be formed. On that account the set $X_{complexity}$ must be defined which contains the complexity degrees of all guards. The set is defined as $X_{complexity} = \bigcup_{\forall \varphi \in \mathfrak{F}(V)} GUARD\_DEPENDENCY(\varphi)$. As a further consequence $MGC = \mu \pm \sigma$ with $\mu = \textbf{mean}(X_{dependency})$ and $\sigma = \textbf{standardDeviation}(X_{dependency}, \mu)$.

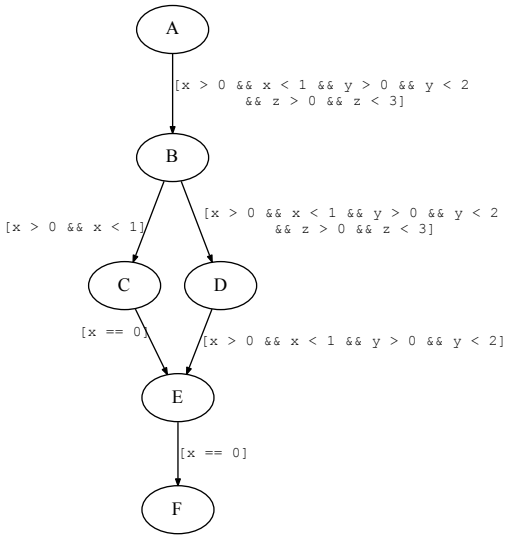


Figure 4.32.: Example of an ESTS to demonstrate MGC metric computation.

Example 4.30 (Mean Guard Complexity Computation)
Based on the ESTS of Figure 4.32 the guard complexity computation is demonstrated. The labels of the transitions only show the guards. With respect to Algorithm 14 the following guard complexity degrees are computed:

- Guard at transition $A \rightarrow B$ has complexity degree 5.
- Guard at transition $B \rightarrow C$ has complexity degree 1.
- Guard at transition $B \rightarrow D$ has complexity degree 5.
- Guard at transition $C \rightarrow E$ has complexity degree 0.
- Guard at transition $D \rightarrow E$ has complexity degree 3.
- Guard at transition $E \rightarrow F$ has complexity degree 0.

On the account of the calculated complexity degrees the set of all values is $X_{complexity} = \{5, 1, 5, 0, 3, 0\}$. Finally, MGC $= 2,33 \pm 2,134$. □

5. Experimental Results

This chapter presents the results which have been obtained with the tool *STSStaticAnalyzer*. The *STSStaticAnalyzer* tool has been developed during this thesis and is realized in *Java* and contains the implementations of the checks which have been discussed in Chapter 4. The results in this chapter are retrieved from a system consisting of three UML models as depicted and described in Section 5.2 and from an illustrative example consisting of two ESTS models described in Section 5.3. In order to apply static analysis to ESTS models, which are based on the UML models, they are transformed into ESTS domain beforehand. The transformation approach is discussed in [27] in detail. The *STSStaticAnalyzer* tool is executed on a set containing all three ESTS model instances or in case of the illustrative example on a set of two ESTS models. For the application of message consistency checks, all ESTS models belonging to the same system has to be passed to the *STSStaticAnalyzer*.

This chapter is organized as follows: First, limitations concerning the tool implementation are discussed. Second, a short description of the example system and the illustrative example is provided. Afterwards, the results obtained by executing the checks on the models are presented and discussed.

5.1. Limitation

The *STSStaticAnalyzer* tool shows an implementation limitation in terms of that the concrete realization of the attribute *Def-Use* data structures deviates from the definition in this thesis. The concrete *Def-Use* data structure implementation is limited to the extent that for neither *def* nor *use* entries the corresponding dependent sets, *RefUses* and *RefDefs*, respectively, are realized. Thus, each *def* entry of an attribute has an impact on every *use* entry corresponding to the attribute. On the other hand a *use* entry of an attribute depends on all *def* entries corresponding to the attribute. Therefore, the result of each check, which utilizes the *Def-Use* data structure, has to be considered with regard to this limitation.

5.2. Illustrative Example Keyless Access Controller

The illustrative example is taken from [27] and depicts a system consisting of three parallel running components, namely the Keyless Access Controller (KAC), the Key Location Detector (KLD) and Power Controller (PC) component. The architecture of the example system is shown in Figure 5.1.

The behavior of these three components are modeled as UML State Charts as shown in Figures 5.2, 5.3 and 5.4. The system illustrated provides the functionality to unlock and lock a car as well as turning on and off the engine and thus the power supply. All of these functions depend on the location of the key.

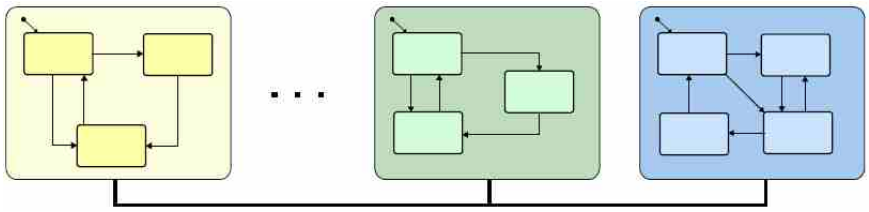


Figure 5.1.: Keyless Access System Architecture. (Image adapted from [27])

The three components communicate with each other by means of messages like *evKeyInRange* in Figure 5.2. Internal messages, which are messages that are sent between the components, are prefixed with *ev*. By contrast, external messages are prefixed with *ex* and depicts messages sent from external sources.

The KAC depicted in Figure 5.2, is the main component of the system defining the behavior in terms when the car is locked or unlocked. Basically, the KAC component consists of two main states, namely *CarStopped* and *CarMoving*. The state *CarStopped* illustrates under which condition the car is locked or unlocked if it is not moving. This fact is depicted by the two sub-states *CarLocked* and *CarUnlocked*. The component's state changes depends on the current state and on the messages received. The external message *extSpeed* notifies the component that the car is in motion. Thus, the car changes into the state *AutoUnlocked* and as a further consequence to *AutoLocked* if the speed exceeds the defined limit of 20. The state *Warning* and its sub-states depict an exception. The KAC changes to the exception mode warning if a message is received notifying that the key is either out of range or in the range again. A situation which should not be possible while the car is in motion. The state *WarnLightOn* indicates that a warning light is turned on. The warning can be acknowledged by the driver, which is indicated by the external message *extAckWarning*, automatically after 1000 time units or if the a message is received that the key is inside the car. All of these facts result in the state change to *AutoUnlocked*.

The purpose of the KLD component, depicted in Figure 5.3, is to determine where the key is located. The component distinguishes two main states, namely *KeyOutsideCar* and *KeyInsideCar*. The former consists of two sub-states stating whether the key is in or out of range of the car. Depending on the state changes the KAC and PC component are notified by means of messages.

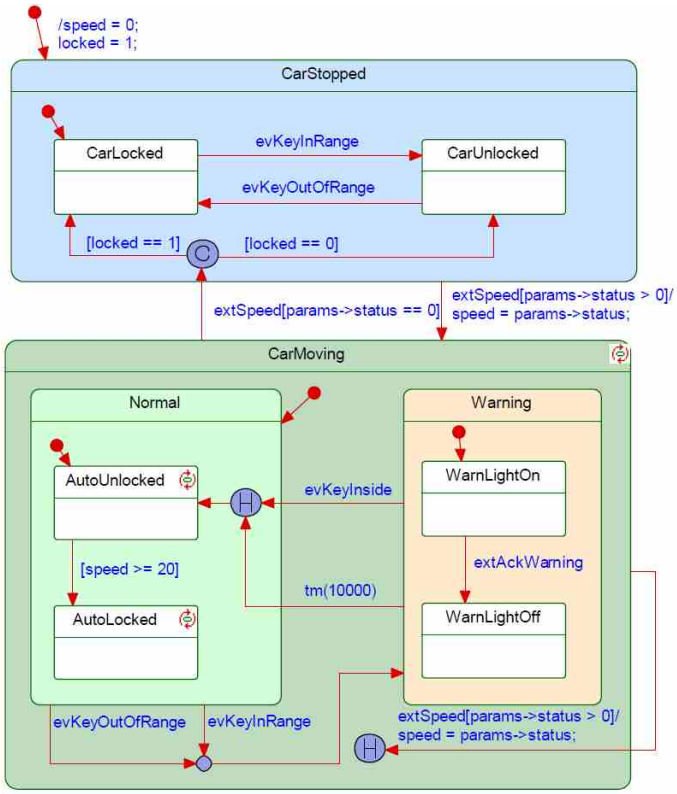


Figure 5.2.: UML State Chart of the Keyless Access Controller. (Image adapted from [27])

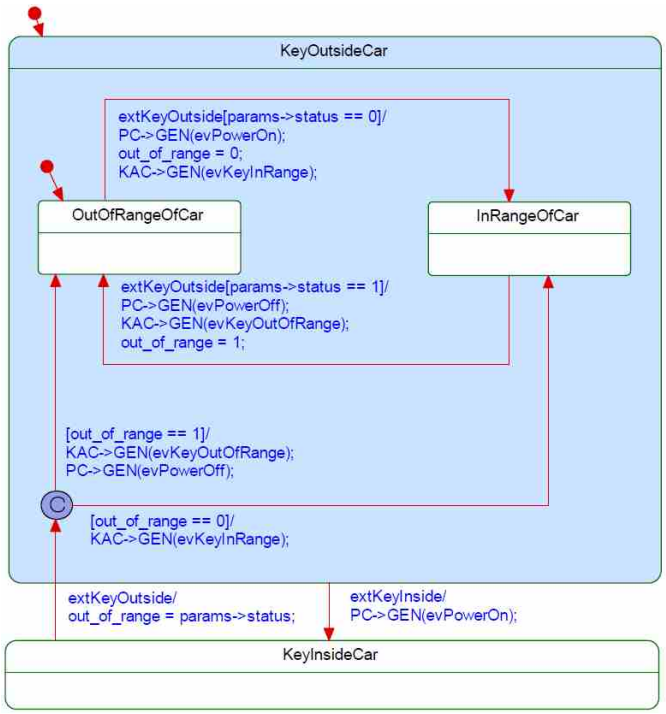


Figure 5.3.: UML State Chart of the Key Location Detector. (Image adapted from [27])

The PC component is responsible for switching between a high and low power mode. In which mode the PC component is located depends on the KLD component status. The PC is in high power mode only if the key is inside range or inside the car, otherwise in low power mode. The low power mode, which is denoted by state *LowPower*, consists of two sub-states *Standby* and *Off*. After the PC component is situated in the *Standby* state for a configured time, it changes to the *Off* state, which indicates that the power is turned off.

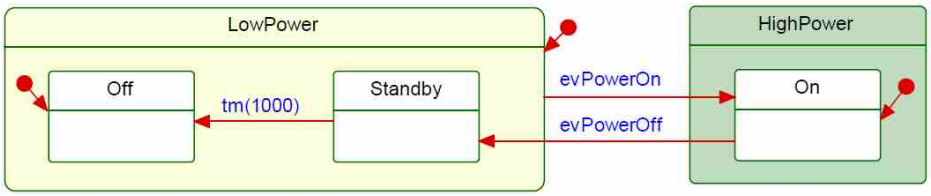


Figure 5.4.: UML State Chart of the Power Controller. (Image adapted from [27])

5.3. Illustrative Example

The two ESTS models depicted in Figure 5.5 have no functional and behavioral purpose, respectively, since they do not model a specific system behavior. On the contrary, these models are utilized to discuss the results for those metrics where no results are obtained from the *Keyless Access Controller* example.

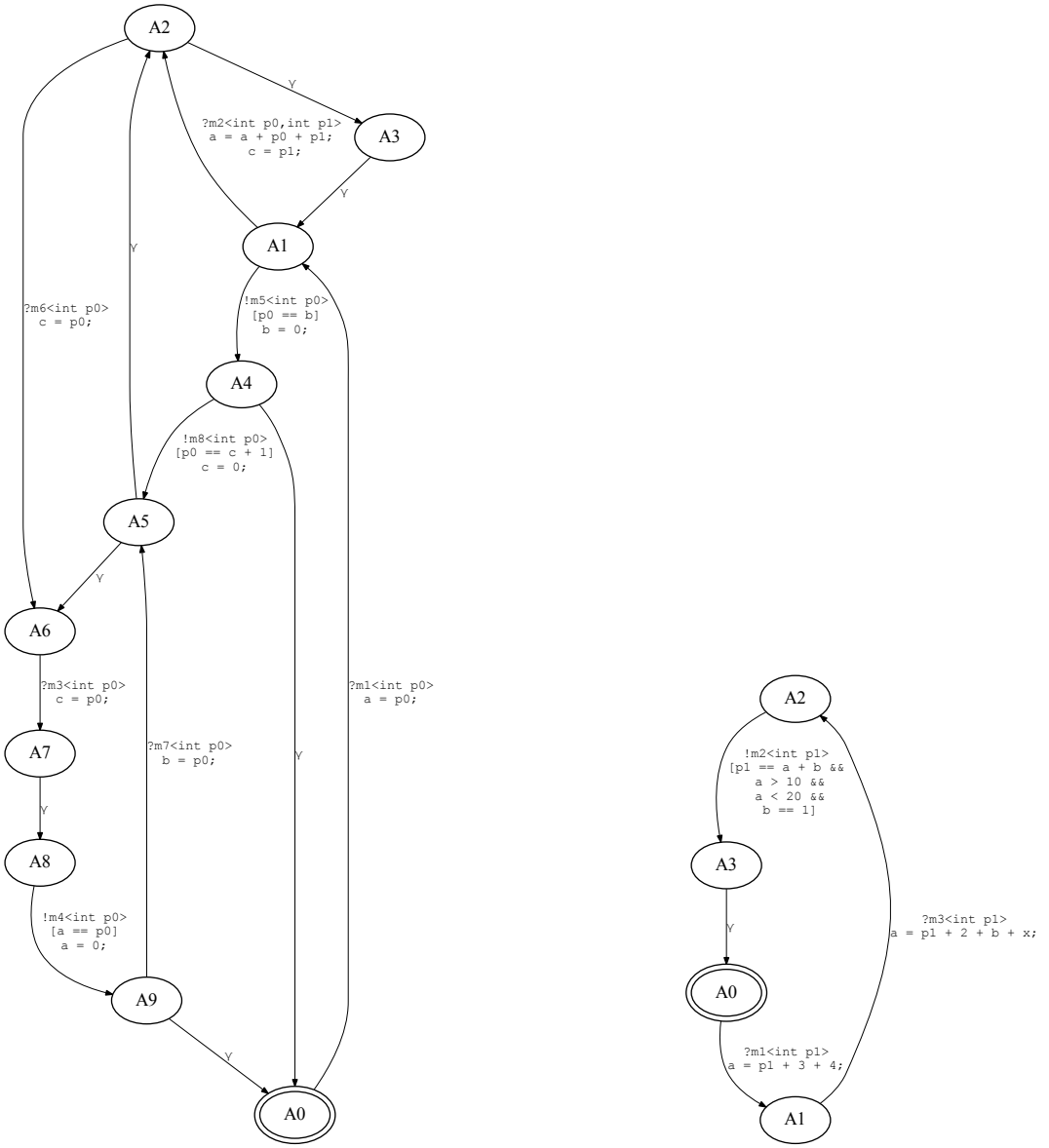
5.4. Results

The example UML State Charts are transformed to ESTS models by means of the transformation approach stated in [27]. The ESTS models created are depicted in the Appendix of this thesis in the Figures A.1, A.2 and A.3.

Table 5.2 shows the results obtained by applying syntactic, semantic and structure-based checks to the KAC ESTS model. The column *Type* corresponds to the check message type and is used to depict which check of Sections 4.2 and 4.3 produced the message. The mapping of check message type to its corresponding check section is shown in Table 5.1. The table only contains those checks which produced a check message.

Check Message Type	Section Title	Section Reference
Message Consistency	Input and Output Message Consistency	4.2.1
Non-determinism	Non-determinism in terms of Overlapping Guards	4.2.5
Loop	Instantly Executable Transition Loops	4.3.1

Table 5.1.: Mapping of check message type to check section.



(a)Model A

(b)Model B

Figure 5.5.: ESTSs of Model A and Model B.

The table contains three check messages with respect to *Message Consistency* and thus denoting a violation where all of them are related to input messages. Two violations are related to external input messages, namely *extSpeed* and *extAckWarning*. Since no model exists which sends these messages they are reported as violations. The only relevant violation in terms of *Message Consistency* is related to the input message *evKeyInside*. This message is neither sent from the KLD nor the PC component, but is expected by the KAC component. As shown in the UML State Chart in Figure 5.2, the *evKeyInside* message belongs to a transition connecting the *Warning* state to the *Normal* state. The next violation corresponds to a non-determinism issue occurring at state *AutoUnlocked*. At this state, two completion transitions, *AutoUnlocked* $\xrightarrow{historyconnector\_42 == 3}$ *AutoUnlocked* and *AutoUnlocked* $\xrightarrow{speed \geq 20}$ *AutoLocked* are enabled at the same time. This is the case if both guards are solvable for at least one attribute valuation, like $speed = 20$, $historyconnector\_42 = 3$. Four checks messages are related to the type *Loop*. Those messages only have informative purpose denoting four self-loops related to completion transitions.

Type	Message
Message Consistency	Input message *extSpeed* is not sent by any other ESTS.
Message Consistency	Input message *extAckWarning* is not sent by any other ESTS.
Message Consistency	Input message *evKeyInside* is not sent by any other ESTS.
Non-determinism	*AutoUnlocked* $\xrightarrow{historyconnector\_42 == 3}$ *AutoUnlocked* and *AutoUnlocked* $\xrightarrow{speed \geq 20}$ *AutoLocked* overlaps for values $speed = 20$ and $historyconnector\_42 = 3$
Instantly Loop	Path: *WarnLightOff* $\xrightarrow{historyconnector\_42 == 6}$ *WarnLightOff*
Instantly Loop	Path: *WarnLightOn* $\xrightarrow{historyconnector\_42 == 6}$ *WarnLightOn*
Instantly Loop	Path: *AutoLocked* $\xrightarrow{historyconnector\_42 == 3}$ *AutoLocked*
Instantly Loop	Path: *AutoUnlocked* $\xrightarrow{historyconnector\_42 == 3}$ *AutoUnlocked*

Table 5.2.: Result of applying structural and consistency checks to KAC ESTS model.

Table 5.3 depicts the check messages with respect to the KLD ESTS model. Only two message consistency issues are found, but either issues are related to external message and thus just have informative purpose. Applying the checks to the PC ESTS model results in no consistency and structural issues.

Type	Message
Message Consistency	Input message *extKeyInside* is not sent by any other ESTS.
Message Consistency	Input message *extKeyOutside* is not sent by any other ESTS.

Table 5.3.: Result of applying structural and consistency checks on KLD ESTS model.

In the following, the metric results obtained are presented. Table 5.4 shows a comparison of the size metrics, which have been stated in Section 4.4.1, and the MCC metric from Section 4.4.2, for all three

system components. Based on the NS and NT metric it can be inferred that the KAC is the most and the PC the least complex model.

The KAC model consists of 7 states and 59 transitions. The ratio of states to transitions, such that the number of transitions is significantly higher as the number of states, leads to the assumption of the existence of states with many incoming and outgoing transitions, respectively. A fact which increases the degree of model complexity. In contrast the KLD, 13 states and 16 transitions, and the PC, 4 states and 5 transitions, models have an almost even states to transitions ratio. The inference that the KAC model is the most complex is supported by comparing the MCC metric, which is for the KAC model considerably higher. According to the MCC metric the KAC model has 53 distinct paths, whereas the KLD and PC models has only 4 and 2 distinct paths, respectively, Thus, in context of testing the effort to accomplish full transition coverage is notably higher for the KAC model.

Metric	KAC	KLD	PC	Section
NS	7	13	4	4.4.1
NT	59	16	5	4.4.1
NTT_{T_i}	47 *79,66%*	6 *37,5%*	3 *60%*	4.4.1
NTT_{T_o}	0	9 *56,25%*	0	4.4.1
NTT_{T_d}	6 *10,17%*	0	1 *20%*	4.4.1
NTT_{T_γ}	6 *10,17%*	1 *6,25%*	1 *20%*	4.4.1
NTT_{T_τ}	0	0	0	4.4.1
NTE	57 *96,61%*	4 *25%*	0	4.4.1
NTG	51 *86,44%*	4 *25%*	0	4.4.1
NM	5	6	2	4.4.1
MCC	53	4	2	4.4.2
NA	4	1	0	4.4.1
$NA_{\mathbb{Z}}$	4 *100%*	1 *100%*	0	4.4.1
$NA_{\mathbb{B}}$	0	0	0	4.4.1
$NA_{\mathbb{R}}$	0	0	0	4.4.1

Table 5.4.: Comparison of Size Metrics and MCC Metric of KAC, KLD and PC ESTS models.

The NTE metric shows that 96,61% of the transitions in the KAC model have an effect, which means an action associated that changes the value of an attribute. The MGC metric shows for the KAC model a high percentage of 86,44%, which means that 86,91% of the transitions have guards assigned. A fact which is supported by the value 51. Due to the high number it can be inferred that the KAC component is more complex than the two others, since both features a very low value with respect to these metrics.

Considering the NTT_{T_i} metric of the KAC model shows that the majority of the transitions in the model correspond to input transitions. Thus, the KAC model is primarily triggered by other models and external sources, without triggering any other model itself as depicted by metric $NTT_{T_o} = 0$. Referring to the NTT_{T_i} and NTT_{T_o} metrics of the KLD it is shown that almost every transition, despite the single completion transition, are either input or output transitions.

The *Keyless Access* system consists of 5 attributes with all of them associated to the integer value domain. The majority of the attributes, precisely 4 attributes, correspond to the KAC model.

Furthermore, one attribute belongs to the KLD model and none to the PC model.

In Table 5.5 the results obtained for the MATRD and MATWD metrics are presented. Since the PC model comprises no attributes the value of either metrics is 0, and thus no *read* and *write dependencies* to transitions exist. The values concerning the KLD model illustrates the fact that only a single attribute is present in the model. The MATRD metric for the KAC model shows a mean value of $\mu = 12,25$ and a standard deviation value of $\sigma = 11,44$. This values denotes that attributes are referenced by 12,25 transitions on average, while the standard deviation indicates a large scatter range around the mean value. This fact depicts the existence of attributes which are only referenced by a few transitions, whereas other attributes exhibit a high count of the transition *read dependency*. A circumstance which is reflected by the corresponding *min* and *max* values. These values state that at least one attribute is read from only one transition and at least one attribute is read by 28 transition. Concerning the MATWD metric the KAC model again shows significant higher values in contrast to the other models. The mean value $\mu = 35,5$ denotes that attributes are written by 35,5 transitions on average. The standard deviation $\sigma = 8,76$ basically leads to the same inference as for the MATRD metric, which is reflected by the *min* and *max* values of 29 and 48, respectively.

MATRD Metric (Section 4.4.4)	**KAC**	**KLD**	**PC**
μ	12,25	2	0
σ	11,44	0	0
min	1	2	0
max	28	2	0
MATWD Metric (Section 4.4.4)			
μ	35,5	4	0
σ	8,74	0	0
min	29	4	0
max	48	4	0

Table 5.5.: Comparison of MATRD and MATWD Metric of KAC, KLD and PC ESTS models.

Table 5.6 shows the results calculated with respect to the MADUD metric. Due to the fact that the PC model contains no attributes no results are obtained with respect to this metric. Considering the KAC model the mean value $\mu = 1,8$ is computed with a standard deviation of $\sigma = 0,41$. Meaning that 1,8 states on average are located between the definition and the use of an attribute. The standard deviation denotes that the overall *def-use* distances are located close to the the mean value. Basically, it can be inferred that an attribute is read shortly after it has been written, and that the structural as well as the behavioral complexity between writing and reading an attribute is low. The KLD model is slightly more complex in terms of the MADUD metric, as the mean value is $\mu = 4$ which means that 4 states on overage are located between a definition of an attribute and its use. The standard deviation $\sigma = 0,76$ depicts that the deviation from the mean is low.

Finally, the results obtained for the MGC metric are discussed which are depicted in Table 5.7. Since NTG $= 0$ for the PC component, no value for the MGC is computed. In contrast the MGC metric for the KLD is 0 as well, but as the model shows a value of 4 for the NTG, it means that none of the four guards utilize a logical operator and thus solely consists of a single condition. The KAC

MADUD Metric (Section 4.4.5)	**KAC**	**KLD**	**PC**
μ	1,8	4	0
σ	0,41	0.76	0
min	1	3	0
max	3	5	0

Table 5.6.: Comparison of MADUD Metric of KAC, KLD and PC ESTS models.

model consists of 51 guards, as stated in in Table 5.4, with a mean complexity of $\mu = 0,63$ and a standard deviation of $\sigma = 0,49$. Thus, all guards utilize either no or only 1 logic operator, which is also reflected by the corresponding *min* and *max* values. Based on the mean value it can be stated that the majority of the guards consist of a single logical operator and thus of two conditions.

MGC Metric (Section 4.4.8)	**KAC**	**KLD**	**PC**
μ	0.63	0	0
σ	0.49	0	0
min	0	0	0
max	1	0	0

Table 5.7.: Comparison of MGC Metric of KAC, KLD and PC ESTS models.

No results have been obtained for the MAAD, MOITD and Mean Output To Input Transition Dependency Distance (MOITDD) metrics when applying the *STSStaticAnalyzer* tool to all three ESTS models. Thus, the illustrative example, shown in Figure 5.5, is used in order to discuss the results concerning these metrics. First, in Table 5.8 the size metrics with respect to *Model A* and *Model B* are shown. Based on this data *Model A* is more complex which is reflected by the number of states and transitions.

Metric	Model A	Model B	Section
NS	10	3	4.4.1
NT	15	3	4.4.1
NTT_{T_i}	5 *33,33%*	2 *50%*	4.4.1
NTT_{T_o}	3 *20%*	1 *25%*	4.4.1
NTT_{T_d}	0	0	4.4.1
NTT_{T_γ}	7 *46,67%*	1 *25%*	4.4.1
NTT_{T_τ}	0	0	4.4.1
NTE	8 *53,33%*	2 *20%*	4.4.1
NTG	3 *20%*	1 *80%*	4.4.1
NM	8	3	4.4.1
MCC	6	1	4.4.2
NA	3	3	4.4.1
$NA_{\mathbb{Z}}$	3 *100%*	3 *100%*	4.4.1
$NA_{\mathbb{B}}$	0	0	4.4.1
$NA_{\mathbb{R}}$	0	0	4.4.1

Table 5.8.: Comparison of Size Metrics and MCC Metric of ESTS *Model A* and *Model B*.

In Table 5.9 the results obtained for the MOITD metrics are depicted. The table shows by comparing the means values of *Model A* and *Model B* that the former model shows a considerably higher dependency of its output transitions to its input transitions. A fact that is reflected by the corresponding *min* and *max* values. Thus, it can be inferred that *Model A* shows a notably higher dependency degree of output to input transitions in contrast to *Model B*, and therefore can be considered more complex.

MOITD Metric (Section 4.4.6)	**Model A**	**Model B**
μ	5	1,5
σ	1,26	0,71
min	4	1
max	7	2

Table 5.9.: Comparison of MOITD Metric of ESTS *Model A* and *Model B*.

Table 5.10 presents the result obtained when calculating the MOITDD metric on *Model A* and *Model B*. The results denote that both models show the same mean distance between inputs and the dependent outputs. Indicating that in both models two states are located between an input and a dependent output on overage.

MOITDD Metric (Section 4.4.7)	**Model A**	**Model B**
μ	2	2
σ	1	0
min	1	2
max	3	2

Table 5.10.: Comparison of MOITDD Metric of ESTS *Model A* and *Model B*.

In conclusion it can be stated that the KAC model is the most complex model of the *Keyless Access System*. This argumentation can be drawn by considering the results depicted in Table 5.4. There, the KAC model shows considerable higher values in terms of the metrics NS, NT, NTE, NTG and MCC. Thus, this model constitutes a significant higher structural and control flow complexity in contrast to the KLD and PC models. This fact is as well subsidized by the results of the MATRD and MATWD metrics for the KAC model, which are presented in Table 5.5. The models only vary slightly in terms of the MADUD and MGC metrics.

Considering the fictional example, *Model A* can be rated as more complex than *Model B*. This conclusion is based on the consistently higher values of the metrics in Table 5.8. Finally, referring to the metrics MOITD and MOITDD, no major differences are in place.

6. Conclusion

Cyber-physical and embedded systems are complex and often used in safety critical domains. To reduce the complexity of such systems models are utilized which provide an abstract view of the systems and thus make them easier to comprehend. Moreover, by means of models a common understanding of the system can be achieved among various stakeholders. Another aspect of models is their application in the field of MBT, where models like an ESTS are used to describe the intended behavior of an SUT. In order to retrieve test cases test generation strategies are applied to these models. These test cases are executed on the SUT to prove its correctness and thus improve the quality of the SUT. As the models are utilized to increase the system quality, it is crucial to ensure that these models are of high quality themselves.

In this thesis a static analyzer suite for ESTS models has been proposed. The suite consists of two components, namely checks and metrics. The quality of the model in terms of syntax, consistency and correctness is ensured by the checks that are proposed. Moreover, the checks are able to extract specific constructs from an ESTS, namely loops and cascades. The metrics provide a quantitative measure of the quality of an ESTS in terms of complexity. The complexity is an indicator of the degree of comprehensibility and the maintainability of a model. The checks and metrics proposed have been implemented in a prototype tool called *STSStaticAnalyzer*, which has been applied to two illustrative examples.

6.1. Future Work

The approach discussed in this thesis can be considered to be one of the first attempts of applying static analysis to ESTS models. The static analyzer suite presented is a first suggestion for the application of checks and metrics to an ESTS model.

Potential future work might encompass the introduction of other metrics known from the software engineering field like fan-in/out, cohesion and coupling. Moreover, the scope of checks could be extended by ensuring syntactic correctness in terms of timing groups and output transition guards. Furthermore, the introduction of a check could be considered in order to statically detect deadlock situation by solving path constraints.

Static analysis cannot detect all potential issues that limit the quality of a model. It may be worth considering the combination of static analysis techniques with dynamic ones. Dynamic analysis

techniques simulate an ESTS model in order to detect deadlocks or race conditions as described in [86].

Appendix

Appendix A.

Experimental Results ESTS Models

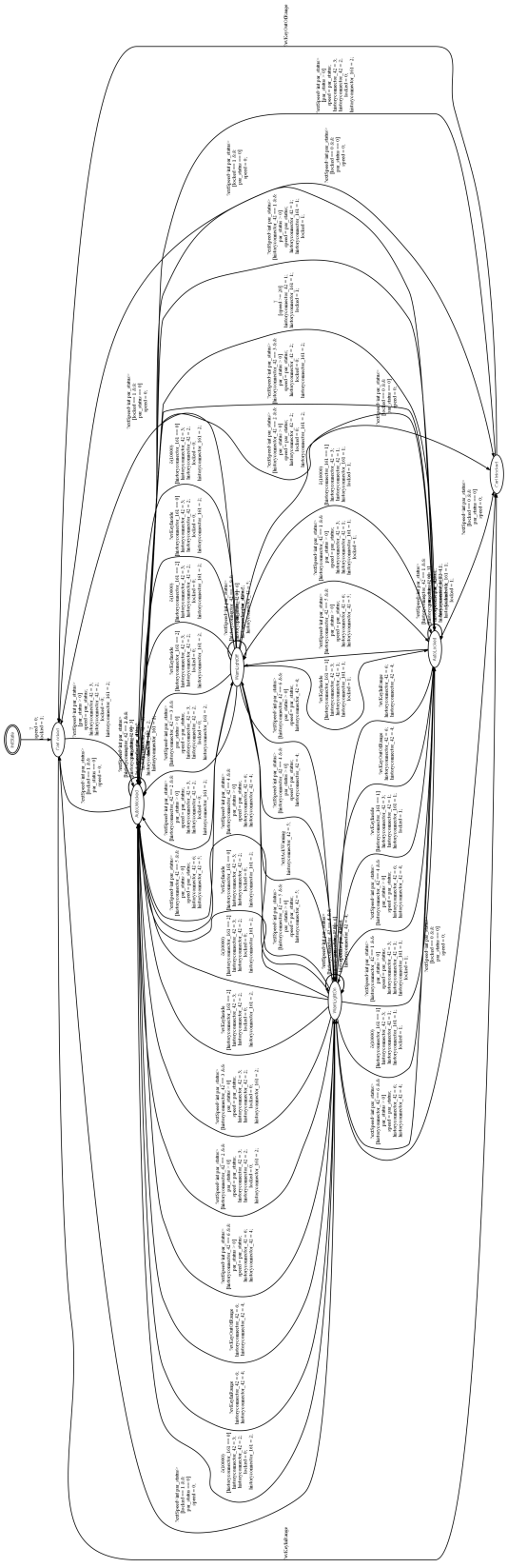


Figure A.1.: The transformed KAC ESTS model based on the UML State Chart depicted in Figure 5.2.

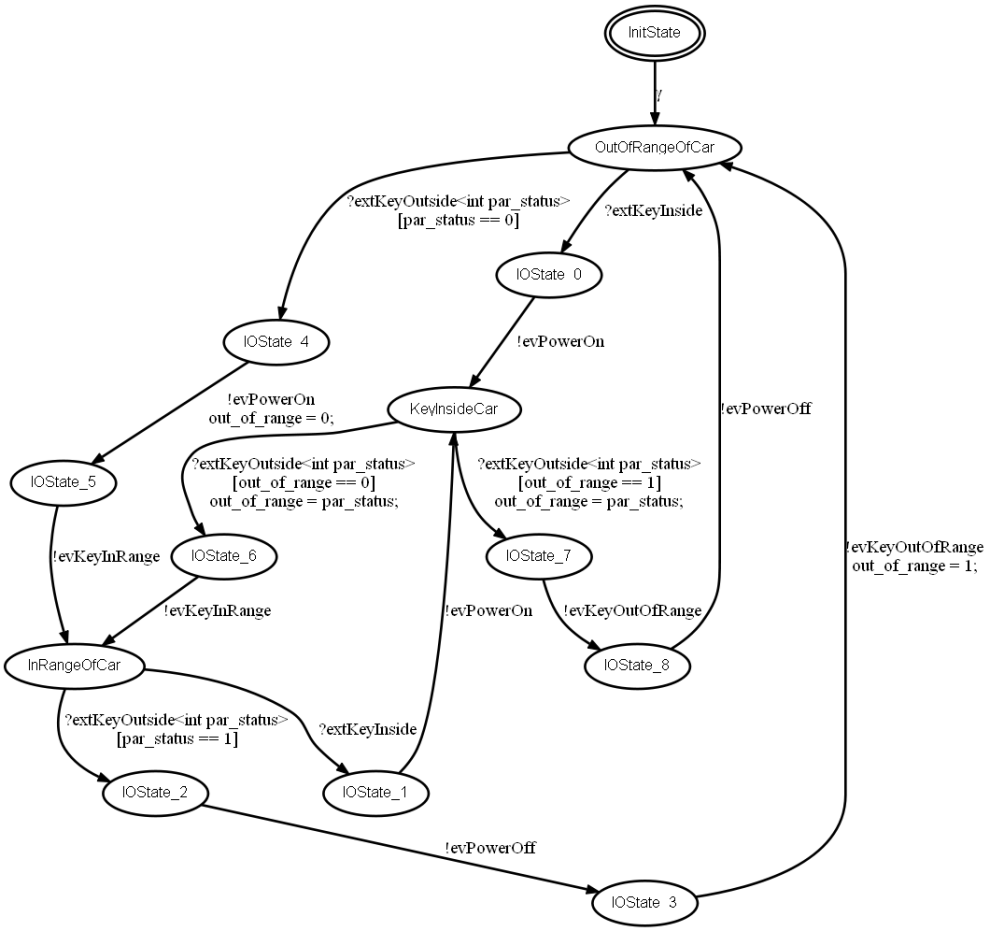


Figure A.2.: The transformed KLD ESTS model based on the UML State Chart depicted in Figure 5.3.

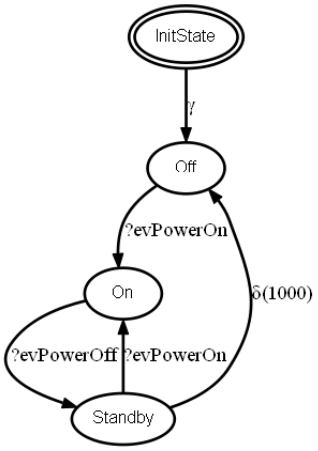


Figure A.3.: The transformed PC ESTS model is based on the UML State Chart depicted in Figure 5.4.

List of Acronyms

ESTS Extended Symbolic Transition System
LTS Labeled Transition System
MBT Model-based Testing
STS Symbolic Transition System
SUT System Under Test
UML Unified Modeling Language
SM State Machine
PCG Program Control Graph
MDE Model Driven Engineering
MDA Model Driven Architecture
MBT Model-based Testing
BET Binary Expression Tree
LTL Linear Temporal Logic
CSP Communicating Sequential Processes
CLP Constraint Logic Programming
OCL Object Constraint Language
BPM Business Process Model
BPMN Business Process Model and Notation
ER Entity-Relationship
FSM Finite State Machine
EPC Event-driven Process Chain
CFC Control Flow Complexity
NA Number of Attributes
NT Number of Transitions
NS Number of States
NTT Number of Transitions by Type
NM Number of Messages
NTG Number of Transitions with a Guard
NTE Number of Transitions with an Effect
MAAD Mean Attribute On Attribute Dependency
MATRD Mean Attribute On Transition Read Dependency
MATWD Mean Attribute On Transition Write Dependency
MADUD Mean Attribute Def-Use Distance
MGC Mean Guard Complexity

MOITD Mean Output To Input Transition Dependency
MOITDD Mean Output To Input Transition Dependency Distance
IPT Independent Path Tree
SUM State Machine Understandability Metric
KAC Keyless Access Controller
KLD Key Location Detector
PC Power Controller
MCC McCabe's Cyclomatic Complexity
CASE computer-aided software engineering
SUT System under Test

Bibliography

[1] *Unified Modeling Language Specification v2.4.1*, Object Management Group (OMG), August 2011.

[2] *Business Process Model and Notation v2.0*, Object Management Group (OMG), January 2011.

[3] A. Gill, *Introduction To The Theory Of Finite-State Machines Assistant Professor of Electrical Engineering*. McGraw-Hill Book Company, Inc., 1962.

[4] S. Kent, "Model Driven Engineering," in *Integrated Formal Methods, Third International Conference, {IFM}*, vol. 2335, 2002, pp. 286–298.

[5] S. J. Mellor, A. Uhl, and D. Weise, "Model-Driven Architecture Models and Metamodels," *OMG white paper*, pp. 290–297, 2000.

[6] L. Apfelbaum and J. Doyle, "Model based testing," *Software Quality Week Conference*, pp. 1–14, 1997. [Online]. Available: http://www.geocities.com/model_based_testing/sqw97.pdf

[7] I. Schieferdecker, "Model-Based Testing," *IEEE Software*, vol. 29, no. 1, pp. 14–18, Jan. 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6111361

[8] I. ISO, "Iso/iec 25010". 2011," *Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models*, 2011.

[9] A. Storch, R. Laue, and V. Gruhn, "Measuring and visualising the quality of models," in *2013 IEEE 1st International Workshop on Communicating Business Process and Software Models: Quality, Understandability, and Maintainability, CPSM 2013*, 2013.

[10] O. I. Lindland, G. Sindre, and A. Solvberg, "Understanding quality in conceptual modeling," *IEEE Software*, vol. 11, pp. 42–49, 1994.

[11] "Findbugs 3.0.0," accessed: 2015-02-17. [Online]. Available: http://findbugs.sourceforge.net/

[12] "Checkstyle 6.3," accessed: 2015-02-17. [Online]. Available: http://checkstyle.sourceforge.net/

[13] "Pmd 5.2.3," accessed: 2015-02-17. [Online]. Available: http://pmd.sourceforge.net/

[14] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, 1999. [Online]. Available: http://books.google.de/books?id=Nmc4wEaLXFEC

[15] K. Vorobyov and P. Krishna, "Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches," in *Proc. SSV*, 2010, pp. 1–7.

[16] C. Schwarzl and B. Peischl, "Static-and dynamic consistency analysis of UML state chart models," *Model Driven Engineering Languages and ...*, pp. 1–15, 2010. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-16145-2_11

[17] J. Lilius and I. Paltor, "vUML: A tool for verifying UML models," *...1999. 14th IEEE International Conference on ...*, 1999. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=802301

[18] J. Küster and J. Stehr, "Towards explicit behavioral consistency concepts in the UML," *...of the 2nd International Workshop on ...*, 2003. [Online]. Available: http://is.uni-paderborn.de/uploads/tx_sibibtex/KuesterSCESM2003.pdf

[19] R. Breu and J. Chimiak-Opoka, "Towards systematic model assessment," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3770 LNCS, pp. 398–409, 2005.

[20] D. Chiorean, M. Paşca, A. Cârcu, C. Botiza, and S. Moldovan, "Ensuring UML Models Consistency Using the OCL Environment," *Electronic Notes in Theoretical Computer Science*, vol. 102, pp. 99–110, Nov. 2004. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1571066104051187

[21] A. Awad and F. Puhlmann, "Structural detection of deadlocks in business process models," *Lecture Notes in Business Information Processing*, vol. 7 LNBIP, pp. 239–250, 2008.

[22] "Ibm rational rose enterprise," accessed: 2015-02-17. [Online]. Available: http://www-03.ibm.com/software/products/en/enterprise

[23] J. Tretmans, "Model based testing with labelled transition systems," *Formal methods and testing*, pp. 1–38, 2008. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-78917-8_1

[24] L. Frantzen, J. Tretmans, and T. Willemse, "A symbolic framework for model-based testing," *...approaches to software testing ...*, pp. 40–54, 2006. [Online]. Available: http://link.springer.com/chapter/10.1007/11940197_3

[25] B. K. Aichernig, "Model-Based Mutation Testing of Reactive Systems," in *Theories of Programming and Formal Methods*, 2013, pp. 23–36.

[26] B. K. Aichernig, E. Jöbstl, and M. Kegele, "Incremental refinement checking for test case generation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7942 LNCS, 2013, pp. 1–19.

[27] C. Schwarzl, "Symbolic Model-based Test Case Generation for Distributed Systems," Dissertation, Graz University of Technology, 2012.

[28] C. Schwarzl and F. Wotawa, "Test case generation in practice for communicating embedded systems," *e & i Elektrotechnik und Informationstechnik*, vol. 128, no. 6, pp. 240–244, Jun. 2011. [Online]. Available: http://link.springer.com/10.1007/s00502-011-0009-5

[29] C. Schwarzl, B. Aichernig, and F. Wotawa, "Compositional random testing using extended symbolic transition systems," *Testing Software and Systems*, pp. 179–194, 2011. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-24580-0_13

[30] F. J. Lucas, F. Molina, and A. Toval, "A systematic review of UML model consistency management," *Information and Software Technology*, vol. 51, no. 12, pp. 1631–1645, Dec. 2009. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0950584909000433

[31] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: A survey," pp. 215–261, 2009.

[32] G. J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997. [Online]. Available: http://www.mendeley.com

[33] O. Grumberg, Y. Meller, and K. Yorav, "Applying software model checking techniques for behavioral UML models," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7436 LNCS, 2012, pp. 277–292.

[34] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 2988, pp. 168–176, 2004. [Online]. Available: http://www.springerlink.com/index/tqa8n61vheno40fm.pdf

[35] F. Fernandes and M. Song, "UML-Checker : An Approach for Verifying UML Behavioral Diagrams," *Journal of Software*, vol. 9, no. 5, pp. 1229–1236, 2014.

[36] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000. [Online]. Available: http://nusmv.fbk.eu/

[37] A. Cimatti and M. Roveri, "Nusmv 1.0: User manual," Technical report, ITC-IRST, Trento, Italy, Tech. Rep., 1998.

[38] S. D. Brookes, C. a. R. Hoare, and a. W. Roscoe, "A Theory of Communicating Sequential Processes," *Journal of the ACM*, vol. 31, no. 3, pp. 560–599, 1984.

[39] M. Goldsmith, B. Roscoe, and P. Armstrong, "Failures-divergence refinement-fdr2 user manual," 2005.

[40] H. Malgouyres and G. Motet, "A UML model consistency verification approach based on meta-modeling formalization," *Proceedings of the 2006 ACM symposium on ...*, pp. 1804–1809, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1141703

[41] J. Jaffar and J.-l. Lassez, "Constraint Logic Programming," *POPL '87 Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 111–119, 1987.

[42] N. Jussien, G. Rochart, and X. Lorca, "Choco: an open source java constraint programming library," *...Contraint Programming (...*, 2008. [Online]. Available: http://hal.archives-ouvertes.fr/docs/00/48/30/90/PDF/choco-presentation.pdf

[43] A. Hanzala and I. Porres, "Consistency of UML class , object and statechart diagrams using ontology reasoners," *Journal of Visual Language and Computing*, vol. 26, pp. 42–65, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jvlc.2014.11.006

[44] C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, and M. Smith, "OWL 2 Web Ontology Language - Structural Specification and Functional-Style Syntax (Second Edition)," *Online*, pp. 1–133, 2012.

[45] R. Shearer, B. Motik, and I. Horrocks, "HermiT : A Highly-Efficient OWL Reasoner," *Complexity*, vol. 432, p. 10, 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.8603&rep=rep1&type=pdf

[46] A. Egyed, "Instant consistency checking for the UML," *Proceedings of the 28th international conference on . . .*, pp. 381–390, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1134339

[47] ——, "UML/analyzer: A tool for the instant consistency checking of UML models," in *Proceedings - International Conference on Software Engineering*, 2007, pp. 793–796. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4222649

[48] A. Reder and A. Egyed, "Computing Repair Trees for Resolving Inconsistencies in Design Models," in *ASE*, 2012, p. 220. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2351676.2351707

[49] W. M. P. Van Der Aalst and a. H. M. Ter Hofstede, "YAWL: Yet another workflow language," *Information Systems*, vol. 30, pp. 245–275, 2005.

[50] W. M. P. Van Der Aalst, "Formalization and verification of event-driven process chains," *Information and Software Technology*, vol. 41, no. January, pp. 639–650, 1999.

[51] V. Gruhn and R. Laue, "Checking Properties of Business Process Models with Logic Programming," in *MSVVEIS*, 2007, pp. 84–93.

[52] O. M. Kherbouche, A. Ahmad, and H. Basson, "Detecting structural errors in BPMN process models," *2012 15th International Multitopic Conference, INMIC 2012*, pp. 425–431, 2012.

[53] M. Browne, E. Clarke, and O. Grümberg, "Characterizing finite Kripke structures in propositional temporal logic," pp. 115–131, 1988.

[54] K. Schneider, *Verification of Reactive Systems: Formal Methods and Algorithms*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[55] S. Onoda, Y. Ikkai, T. Kobayashi, and N. Komoda, "Definition of deadlock patterns for business processes workflow models," in *Systems Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on*. IEEE, 1999, pp. 11–pp.

[56] R. M. Dijkman, M. Dumas, and C. Ouyang, "Formal semantics and analysis of BPMN process models using Petri nets," *Language*, vol. 50, pp. 1–30, 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.3621&rep=rep1&type=pdf

[57] ——, "Semantics and analysis of business process models in BPMN," *Information and Software Technology*, vol. 50, no. 12, pp. 1281–1294, Nov. 2008. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0950584908000323

[58] J. L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 223–252, Sep. 1977. [Online]. Available: http://portal.acm.org/citation.cfm?doid=356698.356702

[59] M. Genero, D. Miranda, and M. Piattini, "Defining metrics for UML statechart diagrams in a methodological way," *Conceptual Modeling for Novel ...*, pp. 118–128, 2003. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-39597-3_12

[60] M. Genero, M. Piattini, and C. Calero, *Metrics for Software Conceptual Models*, M. Genero, M. Piattini, and C. Calero, Eds. Imperial College Press, 2005. [Online]. Available: http://medcontent.metapress.com/index/A65RM03P4874243N.pdf

[61] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, 1976. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1702388

[62] J. Lankford, "Measuring system and software architecture complexity," *2003 IEEE Aerospace Conference Proceedings (Cat. No.03TH8652)*, vol. 8, pp. 8_3849–8_3857, 2003. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1235569

[63] T. Soliman, "Utilizing CK metrics suite to UML models: A case study of Microarray MIDAS software," *Informatics and Systems (...*, 2010. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5461798

[64] M. Hall, "Complexity metrics for hierarchical state machines," *Search Based Software Engineering*, pp. 76–81, 2011. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-23716-4_10

[65] J. Ho Bae, H. Seok Chae, and C. K. Chang, "A metric towards evaluating understandability of state machines:An empirical study," *Information and Software Technology*, vol. 55, no. 12, pp. 2172–2190, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2013.07.011

[66] G. Zhang and M. Hölzl, "A Set of Metrics for States and Transitions in UML State Machines," in *Proceedings of the 2014 Workshop on Behaviour Modelling-Foundations and Applications*. ACM, 2014, p. 2.

[67] K. B. Lassen and W. M. P. van der Aalst, "Complexity metrics for Workflow nets," *Information and Software Technology*, vol. 51, no. 3, pp. 610–626, Mar. 2009. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0950584908001092

[68] J. Cardoso, "Control-flow complexity measurement of processes and Weyuker's properties," *6th International Enformatika Conference*, vol. 8, pp. 213–218, 2005. [Online]. Available: http://eden.dei.uc.pt/~jcardoso/Research/Papers/Oldpaperformat/6th-IEC-2005-CFC-and-Weyker-Properties.pdf

[69] M. Genero, G. Poels, and M. Piattini, "Defining and validating metrics for assessing the understandability of entity-relationship diagrams," *Data and Knowledge Engineering*, vol. 64, pp. 534–557, 2008.

[70] A. S. Vincentelli and A. Pinto, "A complexity metric for concurrent finite state machine based embedded software," *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, no. Sies, pp. 189–195, Jun. 2013. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6601491

[71] V. Gruhn and R. Laue, "Complexity metrics for business process models," *9th international conference on business ...*, pp. 1–12, 2006. [Online]. Available: http://www.researchgate.net/publication/221281564_Complexity_Metrics_for_business_Process_Models/file/d912f5085635ace7b0.pdf

[72] J. Cardoso, "Evaluating workflows and web process complexity," *Workflow Handbook*, vol. 2005, pp. 284–290, 2005.

[73] J. Cardoso and J. Mendling, "A discourse on complexity of process models," *Business process ...*, pp. 117–128, 2006. [Online]. Available: http://link.springer.com/chapter/10.1007/11837862_13

[74] M. H. Halstead, "Elements of software science (operating and programming systems series)," 1977.

[75] S. Henry and D. Kafura, "Software structure metrics based on information flow," *Software Engineering, IEEE Transactions on*, no. 5, pp. 510–518, 1981.

[76] E. Rolón, F. Ruiz, F. García, and M. Piattini, "Applying Software Metrics to evaluate Business Process Models," *CLEI Electronic Journal*, vol. 9, no. 1, 2006.

[77] F. García, M. Piattini, F. Ruiz, G. Canfora, and C. a. Visaggio, "FMESP: Framework for the modeling and evaluation of software processes," *Journal of Systems Architecture*, vol. 52, pp. 627–639, 2006.

[78] W. Khlif, N. Zaaboub, and H. Ben-Abdallah, "Coupling metrics for business process modeling," *WSEAS Transactions on Computers*, vol. 9, no. 1, pp. 31–41, 2010. [Online]. Available: http://www.wseas.us/e-library/transactions/computers/2010/89-144.pdfhttp://www.wseas.us/e-library/conferences/2009/genova/ACS/ACS-32.pdf

[79] B. R. Preiss, *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*, ser. Worldwide Series in Computer Science. Wiley, 2000. [Online]. Available: http://books.google.com.au/books?id=ywpRAAAAMAAJ

[80] J. Bondy and U. Murty, *Graph theory with applications*. Wiley, 1976. [Online]. Available: http://book.huihoo.com/pdf/graph-theory-With-applications/pdf/preface.pdf

[81] D. Lance, R. Untch, and N. Wahl, "Bytecode-based Java program analysis," *... of the 37th annual Southeast regional ...*, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=306382

[82] M. Lam, R. Sethi, J. Ullman, and A. Aho, "Compilers: Principles, Techniques, and Tools," 2006. [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Compilers:+Principles,+Techniques,+and+Tools#0

[83] P. Van Hentenryck and V. Saraswat, "Strategic directions in constraint programming," *ACM Computing Surveys*, vol. 28, no. 4, pp. 701–726, Dec. 1996. [Online]. Available: http://portal.acm.org/citation.cfm?doid=242223.242279

[84] H. F. Ledgard and M. Marcotty, "A genealogy of control structures," *Communications of the ACM*, vol. 18, no. 11, pp. 629–639, Nov. 1975. [Online]. Available: http://portal.acm.org/citation.cfm?doid=361219.361222

[85] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, Jun. 1972. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/0201010

[86] M. Decker, "Simulation of Deterministic and Parallel Execution of Extended Symbolic Transition Systems," Master's Thesis, Graz University of Technology, 2013.