

Masterarbeit

**Statische Analyse mittels Symbolic Execution
von Zustandsautomaten**

Jonas Wielage
Matrikelnummer: 165754
Juli 2020

Prof. Dr. Jakob Rehof
Dr.-Ing. Martin Hentschel

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Software Engineering (LS-14)
<http://ls14-www.cs.tu-dortmund.de>

In Kooperation mit:
itemis AG

Inhaltsverzeichnis

1 Einleitung	1
1.1 Gliederung	2
2 Hintergrund	3
2.1 Yakindu Statechart Tools	3
2.1.1 Aufbau von Yakindu Statechart Tools	4
2.2 Graphentheoretische Grundlagen	8
2.2.1 Definitionen zur Erreichbarkeit	8
2.2.2 Definitionen zu starken Zusammenhangskomponenten	11
2.3 Grundlagen symbolischer Ausführung	12
3 Symbolic Execution	15
3.1 Allgemeiner Ablauf	15
3.2 Vorverarbeitung	15
3.2.1 Vereinfachung von Transitionen	16
3.2.2 Vereinfachung von Expressions	17
3.3 Limitierungen	18
3.3.1 Ausführungsumgebung	19
4 Symbolischer Ausführungsbaum	21
4.1 Knoten	21
4.2 Regeln	23
4.2.1 TransitionRule	24
4.2.2 Regel für die Vereinfachung aktiver Anweisungen	27
4.2.3 Regeln für die Verarbeitung aktiver Anweisungen zu Pfadbedingungen	31
4.2.4 Regeln für die Zyklenelimination	35
5 Pfadexplosion durch Zyklen	37
5.1 Pfadexplosion durch Zyklen in der Fachliteratur	37
5.2 Zyklenelimination	38
5.2.1 Elimination von Transitionszyklen	39

5.2.2	Elimination von Zyklen durch lokale Reaktionen	42
6	Erfüllbarkeitsbestimmung	45
6.1	Kostenfunktion	45
6.2	Wiederverwendung von Erfüllbarkeiten	46
6.3	Kontext-Solving	47
6.4	Unabhängigkeitsoptimierung	47
6.4.1	Grundlagen zur Unabhängigkeit	48
6.4.2	Reduktion durch Unabhängigkeit	49
6.5	SMT-LIB2 Serialisierung	50
6.5.1	Division mit Rest	51
6.6	Solving mit Z3	52
6.7	Solving mit jConstraints	53
6.8	Verbreitung des Erfüllbarkeitsergebnisses	54
6.8.1	Ältesten-Suche im SET	54
6.8.2	Binäre-Suche im SET	55
7	Konstruktion des SET	57
7.1	Konstruktion der Wurzel	57
7.2	Symbolische Programmausführung	58
7.3	Strategie der symbolischen Programmausführung	59
7.4	Terminierung von Pfaden	60
8	Analyse	63
8.1	Erreichbarkeitsanalyse	63
8.2	Weitere Analyse	65
8.2.1	Uninitialisierte Variablen	65
8.2.2	Generierung von Testfällen	65
8.2.3	Weitere Analysemöglichkeiten	66
9	Evaluation	67
9.1	Korrektheit	67
9.2	Performanz	68
9.2.1	Skalierbares Statechart	69
9.2.2	Evaluierung der default Strategie	70
9.2.3	Evaluierung der Transitionszyklenelimination	71
9.2.4	Evaluierung der Unabhängigkeitsoptimierung	74
9.2.5	Evaluierung des Kontext-Solving	74
9.2.6	Evaluierung der Binäre-Suche Ergebnisverbreitung	75
9.2.7	Evaluierung der DFS Konstruktion	76

9.2.8	Evaluierung von jConstraints	77
9.2.9	Evaluierung der Kostenfunktion	79
9.2.10	Evaluierung weiterer Strategien	80
9.3	Verbesserungen zum ursprünglichen Algorithmus	81
9.4	Folgerungen	83
10	Zusammenfassung	85
10.1	Ausblick	86
A	Weitere Informationen	87
	Abbildungsverzeichnis	93
	Literaturverzeichnis	97
	Erklärung	97

Kapitel 1

Einleitung

Symbolische Programmausführung ist eine Programmanalyse, die in den letzten Jahren an Beliebtheit gewonnen hat ([21]). Die Programmausführung wird dabei nicht mit konkreten, sondern symbolischen Eingabewerten ausgeführt. Dies ermöglicht die Analyse aller möglichen Ausführungspfade und damit die Überprüfung, ob bestimmte Eigenschaften durch das Programm verletzt werden. Bekannte Werkzeuge, die symbolische Programmausführung anbieten, sind beispielsweise KeY-Project [24], Klee [9] oder JavaPathFinder [18], wobei sich die Analyse solcher Werkzeuge häufig auf Programmcode wie Java, C bzw. C++ beschränken. Speziell im Kontext von Zustandsautomaten (*Statecharts*) existieren momentan keine passenden Werkzeuge, um diese umfassend zu analysieren ([11]).

In dieser Arbeit wird eine symbolische Programmausführung auf Zustandsautomaten vorgestellt. Die Ergebnisse der Ausführung werden in einem sogenannten symbolischen Ausführungsbaum (*symbolic execution tree, SET*) gespeichert, auf welchem eine Erreichbarkeitsanalyse angeboten wird. Konkret identifiziert die Engine unerreichbare Zustände und Transitionen und zeigt einem Nutzer diese interaktiv während der Modellierung des Statecharts an. Die Engine wurde aufbauend auf dem in [11] vorgestellten Algorithmus entwickelt, aber mit einer komplett neuen Implementierung. Die Ziele bei der Entwicklung waren hauptsächlich Verbesserungen an der Genauigkeit und Ausführungszeit. Weiterhin wurde die Engine mit einer modularen Bauweise entwickelt, sodass Verbesserungen, weitere Optimierungen oder Änderungen an der Strategie der Programmausführung einfach integrierbar und umsetzbar sind. Durch den symbolischen Ausführungsbaum ist gegeben, dass weitere Analysen auf diesem möglich sind, wie beispielsweise eine Testfallgenerierung oder eine Analyse auf Divisionen durch 0.

Im Rahmen dieser Arbeit wurde die Engine in Anbindung an Yakindu Statechart Tools ([17]) und in Kooperation mit der itemis AG entwickelt. Yakindu Statechart Tools ist eine in Eclipse integrierte modulare Entwicklungsumgebung, die zur Modellierung, Simulation und Generierung von ausführbaren Zustandsautomaten verwendet wird.

1.1 Gliederung

In der vorliegenden Arbeit werden in Kapitel 2 zunächst die Grundlagen für das Verständnis der entwickelten symbolischen Programmausführung erläutert. Dabei werden Statecharts im Yakindu Statechart Tools Kontext und graphentheoretische Grundlagen beschrieben. Anschließend wird eine Einleitung in symbolischer Programmausführung gegeben, welche dessen Hauptmerkmale und Probleme anspricht. In Kapitel 3 wird der allgemeine Ablauf der Engine und die Vorverarbeitung von Statecharts besprochen. Zusätzlich werden Limitierungen gegeben, die erläutern, welche Statechart Modelle analysierbar sind. In Kapitel 4 wird das Kernstück der Engine erklärt – der symbolische Ausführungsbaum – welcher alle relevanten Informationen der symbolischen Programmausführung speichert. Zuerst werden die einzelnen Knoten des Baumes und danach die Regeln, welche für die Generierung weiterer Knoten zuständig sind, besprochen. Kapitel 5 beschreibt die Behandlung der Engine von Pfadexplosion durch Zyklen. Dabei wird besprochen, wie die Fachliteratur mit dieser Art von Pfadexplosion umgeht. Anschließend werden zwei Arten von Pfadexplosion im Kontext von Statecharts und die Elimination von Zyklen beschrieben. In Kapitel 6 wird die entwickelte Erfüllbarkeitsbestimmung erklärt. Hierbei werden alle eingesetzten Optimierungen, sowie das Solven und die anschließende Verarbeitung des Ergebnisses beschrieben. Kapitel 7 beschreibt den Ablauf der Konstruktion des SET, genauso wie die einstellbaren Strategien der Engine. In Kapitel 8 wird die Erreichbarkeitsanalyse erläutert, die auf dem konstruierten SET durchgeführt wird. Danach werden weitere Analysemöglichkeiten wie die Testfallgenerierung beschrieben, die auf dem SET möglich sind. Kapitel 9 ist eine anschließende Evaluation der Engine. Hier wird die Korrektheit und Performanz der Analyse bewertet. Abschließend ist in 10 eine Zusammenfassung der Arbeit mit Ausblick aufgeführt.

Kapitel 2

Hintergrund

In diesem Kapitel werden die Grundlagen für das Verständnis der symbolischen Programmausführung erläutert. Einleitend werden Statecharts in Abschnitt 2.1 im Yakindu Statechart Tools Kontext erläutert. In 2.2 werden graphentheoretische Grundlagen definiert. Abschließend wird in 2.3 eine Einleitung zu symbolischer Ausführung gegeben.

2.1 Yakindu Statechart Tools



Abbildung 2.1: Yakindu ScT Logo [17]

Yakindu Statechart Tools (*Yakindu ScT*) ist eine in Eclipse integrierte, modulare Entwicklungsumgebung, welche zur Modellierung, Simulation und Generierung von ausführbaren Zustandsautomaten verwendet wird. Mit Yakindu ScT ist es möglich ereignisgesteuerte Systeme mittels Harel Statecharts zu entwickeln. Aus diesen Statecharts kann Quellcode für verschiedene Hochsprachen wie Java, C++ und weitere generiert werden [17]. Yakindu ScT wird in einer Standard- und einer Professional-Edition angeboten, wobei die letztere weitere Funktionen beinhaltet, welche besonders für den Embedded-Systems Bereich nützlich sind. Für den akademischen Gebrauch kann kostenfrei eine Professional-Lizenz erworben werden, die Standard-Lizenz ist für nicht kommerzielle Projekte kostenfrei.

Harel Statecharts sind eine Darstellungsform von klassischen endlichen Automaten, welche 1984 von David Harel veröffentlicht wurde. Diese erweitert bestehende Notationen klassischer endlicher Automaten, um komplexe und große Systeme handhabbar zu machen. Beispielsweise erlaubt diese Form die Darstellung von Konditionen, Ereignisse oder Entry- und Exit-Aktionen bei Zuständen. Yakindu Statecharts basieren auf Harel Statecharts.

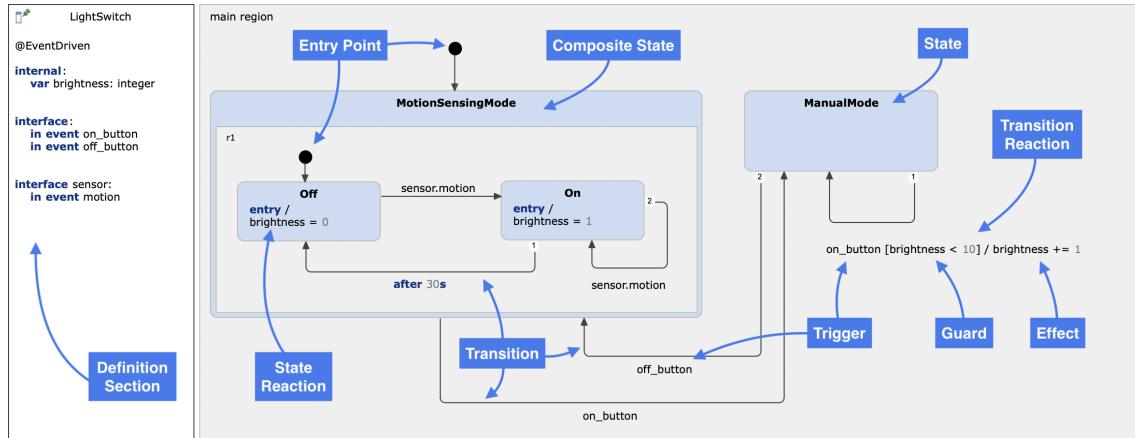


Abbildung 2.2: Beispielhaftes Statechart welches in Yakindu ScT modelliert ist ([17])

Weiterhin sind sie sehr ähnlich aber nicht identisch zu *UML state machines*, welche auf Harel Statecharts basieren. Genauere Informationen über Harel Statecharts sind in [15] und über UML state machines in [8] zu finden. Unterschiede der Yakindu Statechart Implementierung im Gegensatz zu den beiden Modellen stehen in der offiziellen Dokumentation [17].

2.1.1 Aufbau von Yakindu Statechart Tools

Nachfolgend wird ein kurzer Überblick über Yakindu ScT gegeben. Eine vollständige Grammatik der Statechart Elemente ist in den Abbildungen A.1, A.2 und A.3 im Anhang zu finden. Zusätzlich wird auf die offizielle Dokumentation [17] verwiesen, welche eine ausführliche Erklärung aller Elemente und Funktionen von Yakindu ScT enthält.

Abbildung 2.2 zeigt einen Überblick über Yakindu ScT . Links ist die *Definition Section* zu sehen, eine textuelle Modellierung von Variablen, Ereignissen und Operationen. Alle Elemente, welche innerhalb des *Canvas* benutzt werden, müssen hier deklariert werden. Dabei wird festgelegt, ob die Elemente von außerhalb erreichbar sind oder nur innerhalb des Statecharts benutzbar sind. Zudem können mittels Annotationen Eigenschaften in der Ausführung des Statecharts, wie die Taktung, beeinflusst werden.

Rechts ist das *Canvas* zu erkennen, welches eine grafische Modellierung der Statechart-elemente erlaubt. Durch graphische Markierungen wird der Nutzer auf Fehler in der Modellierung aufmerksam gemacht.

Im Beispiel 2.2 wird die integer Variable **brightness** deklariert. In diesem Fall wurde der Variable kein Wert zugewiesen, sodass durch Yakindu anfangs der default Wert 0 gesetzt wird. Es ist möglich selber initiale Werte für die Variablen zu definieren, beispielsweise **var brightness: integer = 5**. Weiterhin werden die Events **on_button**, **off_button** und **motion** deklariert, welche von außerhalb des Statecharts aktiviert werden. Das Event **motion** liegt zudem im **interface sensor**, welches eine Abstraktionsebene zwischen den

hier definierten Elementen darstellt. Insgesamt sind hier noch weitere Typen von Variablen wie beispielsweise `boolean`, `string` oder `real` möglich.

In nachfolgenden Darstellungen von Statecharts in dieser Arbeit wird die Definition Section nicht mit abgebildet. Diese wird implizit durch die Variablen innerhalb des Statecharts definiert. a, b, c, d sind immer numerische, v, w, x, y, z immer boolesche Variablen. evt_i stellen Events dar, welche von außerhalb ausgelöst werden.

Zustand

Zustände (*states*) sind eines der wesentlichen Elemente innerhalb eines Statecharts. Diese können Reaktionen definieren, die beim Eintreten oder Verlassen des Zustandes durchgeführt werden. Zustände sind durch Transitionen untereinander verbunden. Zusätzlich gibt es spezielle Zustände, wie beispielsweise Entry, Final, komposite oder orthogonale Zustände. Entry Zustände markieren den initialen Zustand einer Region. Zustände können zudem aktiv sein, wobei immer mindestens ein Zustand des Statecharts aktiv sein muss.

Im Beispiel 2.2 sind 2 Entry Zustände zu finden. Zusätzlich sind die Zustände `Off`, `On` und `ManualMode` normale Zustände. Der `MotionSensingMode` ist ein kompositer Zustand.

Transition

Eine Transition ist eine gerichtete Kante zwischen zwei Zuständen. Bei Durchführung der Transition wird der Ausgangszustand verlassen und der Zielzustand betreten. Transitionen sind zudem Reaktionen und besitzen Trigger, Guard und Effekte. Wenn ein Zustand mehrere ausgehende Transitionen besitzt, erhalten diese Prioritäten, welche durch den Nutzer veränderbar sind. Bei Transitionsdurchführung wird in der Reihenfolge der Prioritäten geschaut, ob eine Transition durchführbar ist. Ist dies nicht der Fall, wird die nächste Transition betrachtet.

Im Beispiel 2.2 besitzt der Zustand `On` zwei ausgehende Transitionen. Zuerst wird überprüft, ob der Zustand schon mehr als $30s$ aktiv ist. In diesem Fall wird die erste Transition durchgeführt. Andernfalls wird überprüft, ob das Event `motion` vorliegt. Falls dies der Fall ist, wird die zweite Transition durchgeführt. Andernfalls wird keine Transition durchgeführt.

Region

Regionen stellen einen Container für Zustände und Transitionen dar. Durch sie können komposite Statecharts und Hierarchien modelliert werden.

Im Beispiel 2.2 existieren die top-level `main` `region` und die Region `r1` innerhalb des kompositen Zustandes `MotionSensingMode`.

Reaktion

Reaktionen sind eines der Hauptfeatures in Yakindu ScT. Sie treten entweder als Transitionsreaktionen oder als lokale Reaktion eines Zustandes auf. Reaktionen bestehen aus drei optionalen Komponenten, welche in folgender Syntax vereint sind: `trigger[guard]/effects`. Alle Komponenten sind optional, wobei eine Reaktion ohne Trigger und Guard niemals durchgeführt wird, mit Ausnahme der Ausgangstransition des initialen Zustandes.

Trigger definieren, was eintreten muss, damit die Reaktion durchgeführt wird. Mögliche Trigger sind Events oder bestimmte Schlüsselworte, wie zum Beispiel `entry` und `exit`, welche jeweils bei Betreten und Verlassen eines Zustandes ausgelöst werden oder `after`, welches nach einer definierten Zeitspanne ausgelöst wird.

Guard ist ein boolescher Ausdruck, welcher zu `true` ausgewertet werden muss, damit die Reaktion durchgeführt wird.

Effekte sind eine geordnete Liste von Expressions, welche bei der Durchführung der Reaktion ausgeführt werden. Bestimmte Arten von Expressions haben hier keinen Nutzen, wie zum Beispiel eine Addition ohne Zuweisung, sind aber trotzdem erlaubt.

Im Beispiel 2.2 sind mehrere Reaktionen zu finden. `entry / brightness = 0` ist beispielsweise eine lokale Reaktion des Zustandes `Off`. Die Reaktion wird beim Betreten des Zustandes durchgeführt und setzt als Effekt die Variable `brightness` auf den Wert 0. Die Transitionsreaktion `on_button [brightness < 10] / brightness +=1` wird durchgeführt, wenn das Event `on_button` vorliegt und gleichzeitig die Bedingung `brightness < 10` zu `wahr` ausgewertet wird. Anschließend wird der Wert der Variable um eins erhöht. Bei Durchführung der Transition wird der Zustand `ManualMode` verlassen und direkt wieder betreten, da dies eine Selbsttransition ist.

Expression

Ausdrücke (*Expressions*) in Yakindu sind ähnlich wie Ausdrücke in anderen Programmiersprachen. Die Sprache erlaubt boolesche, numerische und bitwise Expressions. Nachfolgend werden die wichtigsten Expressions erläutert. Die vollständige Expression Grammatik ist in Abbildung A.4 und A.5 im Anhang und in [17] zu finden.

Boolesche Expressions erlauben die Darstellung von Expressions, die durch `AND`, `OR` und `NOT` verknüpft sind. Die Operanden sind wiederum boolesche Expressions. Weiterhin werden die Operatoren `<`, `<=`, `>` und `>=` unterstützt, welche zwei numerische Expressions als Operanden besitzen. Die Operatoren `==` und `!=` besitzen zwei gleiche Arten von Operanden.

Numerische Expressions erlauben die Darstellung von Expressions, welche die Operatoren `+`, `-`, `*`, `/` und `%` besitzen. Bei einer numerischen Expression mit diesem Operatoren sind beide Operanden wieder numerische Expressions.

Variablen werden in Yakindu als Expression dargestellt. Je nach Typ sind diese beispielsweise boolesch oder numerisch.

PrimitiveValueExpressions enthalten ein Literal, wie beispielsweise ein `BoolLiteral` oder `IntLiteral`.

AssignmentExpressions weisen einer Variable einen Wert zu. Die linke Seite der Expression wird *Assignee* genannt, die rechte Seite *Assignor*. Assignee und Assignor müssen vom selben Typ sein.

ConditionalExpressions besitzen eine Bedingung, eine `TrueCase`- und eine `FalseCase`-Expression. Wenn die Bedingung zu `true` evaluiert wird, gilt die `TrueCase`-Expression, andernfalls die `FalseCase`-Expression. `ConditionalExpression` können zusätzlich mit `AssignmentExpression` verknüpft werden. Gilt die Bedingung wird der Variable die `TrueCase`-Expression zugewiesen.

SequenceBlockExpressions Für diese Arbeit wurde zusätzlich die `SequenceBlock`-Expression entworfen. Diese verwaltet eine geordnete Liste beliebiger Expressions.

`SequenceBlockExpression` können ineinander verschachtelt werden. Abbildung 2.3 zeigt die Grammatik der Expression. Eine `SequenceBlockExpression` wird als $[expr_1, \dots, expr_n]$ dargestellt.

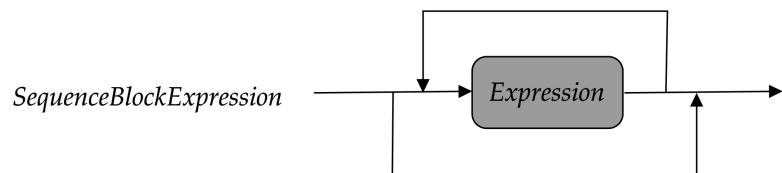


Abbildung 2.3: Grammatik der `SequenceBlockExpression`

Beispiele

Boolesche Expression:

`x || (y < 0); !x`

Numerische Expression:

`a + 5; 5/b`

PrimitiveValueExpression:

`true; 5`

ConditionalExpression:

`x > 5?(a - 1) : (a + 1)`

AssignmentExpression:

`a = b * 2; x = y && (3 > 2)`

Assignment mit Condit.	$a = x > 5?(1) : (-1)$
SequenceBlockExpression:	$[a = a + 1; [\text{true}]; x y]$

2.2 Graphentheoretische Grundlagen

In diesem Abschnitt werden graphentheoretische Grundlagen erläutert. Zuerst werden einige Definitionen bezüglich der Erreichbarkeit von Zuständen und Transitionen beschrieben (2.2.1), danach Definitionen und Folgerungen zu starken Zusammenhangskomponenten (2.2.2).

2.2.1 Definitionen zur Erreichbarkeit

2.2.1.1 Definition (Durchführbarkeit von Transitionen)

Eine Transition t ist genau dann *durchführbar*, wenn

1. der Guard der Transition unter der Variablenbelegung im Ausgangszustand zu `true` evaluiert wird, und
2. die Evaluierung des Guards der Transition und die Ausführung der Effekte der Transition unter der Variablenbelegung im Ausgangszustand keine illegalen Operationen, wie Division durch 0, durchführen, und
3. die Trigger der Transition im aktiven Zustand eintreffen, und
4. der Ausgangszustand erreichbar ist.

2.2.2 Definition (Pfad)

Ein *Pfad* eines Statechart ist eine Sequenz $P = (t_0, t_1, \dots, t_{n-1})$ mit Transitionen t_i , wobei diese auch mehrfach vorkommen dürfen. Der selbe Pfad ist auch durch die Sequenz $PR = (X_0, t_0, X_1, t_1, X_2, \dots, X_{n-1}, t_{n-1}, X_n)$ beschreibbar, mit Zuständen X_i und Transitionen t_i . Hierbei gilt: t_i ist ausgehende Transition von X_i und t_{i-1} ist eingehende Transition von X_i . Da Start- und Endknoten einer Transition eindeutig definiert sind, enthält PR im Vergleich zu P nur redundante Informationen. PR , oder auch P mit den selben Bezeichnungen der zugehörigen Zustände, nennt man auch Pfad von X_0 nach X_n .

2.2.3 Definition (Teilpfad)

Ein *Teilpfad* eines Pfads $P = (t_0, t_1, \dots, t_{n-1})$ ist eine Sequenz $TP_P = (t_i, t_{i+1}, \dots, t_m)$ mit $0 \leq i \leq m \leq n - 1$, welche aus P entsteht, indem eine beliebige Anzahl an Elementen des Anfangs und Endes von P entfernt werden. Die übrig gebliebenen Transitionen sind weiterhin zusammenhängend und der Teilpfad muss mindestens eine Transition enthalten.

2.2.4 Definition (Ausführungszustand)

Ein *Ausführungszustand* ist ein Zustand mit zusätzlich gespeicherter Variablenbelegung, die in diesem Zustand gilt.

2.2.5 Definition (Ausführungspfad)

Ein *Ausführungspfad* bezeichnet einen Pfad, bei welchem die Zustände Ausführungszustände sind. Demnach ist nach jeder Transitionsdurchführung eine Variablenbelegung gespeichert.

2.2.6 Definition (Valider Pfad)

Ein Ausführungspfad ist genau dann *valide*, wenn folgende Bedingungen gelten:

1. Das erste Element ist der Startzustand, und
2. jede Transition ist aus dem vorherigen Ausführungszustand des Pfades durchführbar, und
3. jeder Zustand, außer der erste, entsteht aus der Durchführung der vorherigen Transition.

2.2.7 Definition (Zyklus)

Ein Pfad ist genau dann ein *Zyklus*, wenn der Pfad die Form (t_1, \dots, t_n, t_1) hat, also wenn die erste und letzte Transition der Sequenz identisch sind.

Ein Pfad enthält genau dann einen Zyklus, wenn es einen Teilpfad des Pfades gibt, welcher ein Zyklus ist.

2.2.8 Definition (Zyklusinduzierender Zustand)

Wir nennen einen Zustand X eines Statecharts genau dann *zyklusinduzierend*, wenn ein Zyklus des Statecharts existiert, welcher mit X beginnt. Ein Zustand X mit Selbsttransition t ist offensichtlich zyklusinduzierend mit dem Zyklus $P = (t, t)$.

2.2.9 Definition (Erreichbarer Zustand) Ein Zustand X ist genau dann *erreichbar*, wenn es mindestens einen validen Pfad durch das Statechart gibt, welcher X beinhaltet.

2.2.10 Definition (Erreichbare Transition) Eine Transition t ist genau dann erreichbar, wenn es mindestens einen validen Pfad durch das Statechart gibt, welcher t beinhaltet.

2.2.11 Definition (Erreichbarkeitsanalyse)

Eine *Erreichbarkeitsanalyse* ist eine Analyse, welche für ein gegebenes Statechart entscheidet, welche Zustände und Transitionen bei der Ausführung des Statecharts erreichbar und unerreichbar sind. Ein *false positive* Fehler ist dabei ein Element, welches fälschlicherweise als unerreichbar markiert wurde. Ein *false negative* Fehler bezeichnet ein Element, welches fälschlicherweise als erreichbar markiert wurde.

2.2.12 Definition (Überschattete Transition) Eine Transition t_{low} ist genau dann von einer Transition t_{high} überschattet, falls gilt:

1. t_{low} und t_{high} haben den selben Ausgangszustand, und

2. t_{low} ist niedriger priorisiert als t_{high} , und
3. $\neg(t_{low}.guard \implies t_{high}.guard)$ ist unerfüllbar, und
4. $t_{low}.trigger \subseteq t_{high}.trigger$ gilt, also jeder Trigger, der t_{low} auslöst, muss auch t_{high} auslösen.

Aussagen 3. und 4. lassen sich auch zusammenfassen zu der Aussage:

Wenn t_{low} durchführbar ist, ist auch t_{high} durchführbar.

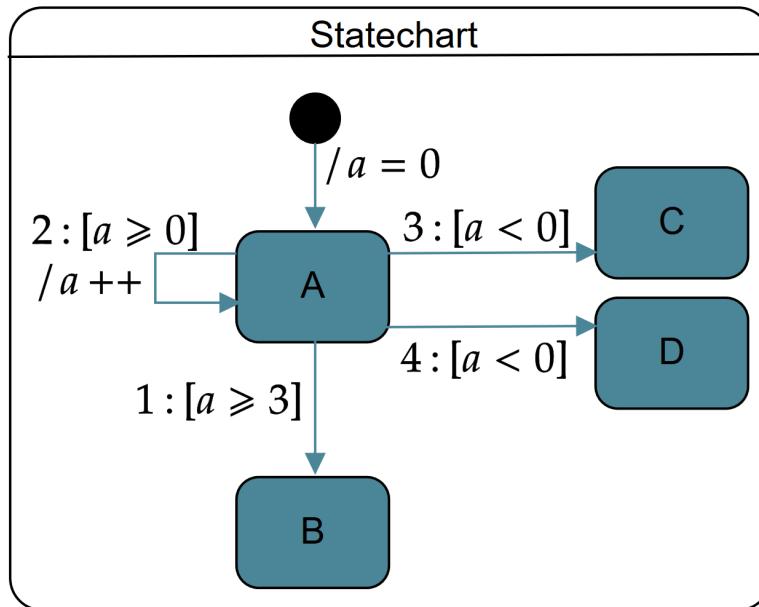


Abbildung 2.4: Beispiel Statechart mit unerreichbaren Zuständen C und D

Beispiel In Abbildung 2.4 ist ein Statechart mit folgenden Eigenschaften abgebildet:

Zyklus:

$$P_{Zyklus} = (A \rightarrow A, A \rightarrow A)$$

valider Pfad:

$$P_{val} = (Entry \rightarrow A, A \rightarrow A, A \rightarrow A, A \rightarrow A, A \rightarrow B)$$

Teilpfad von P_{val} :

$$TP_{P_{val}} = (A \rightarrow A, A \rightarrow B)$$

Kein Teilpfad von P_{val} :

$$(Entry \rightarrow A, A \rightarrow B)$$

invalider Pfad mit Zyklus:

$$P_{inv} = (Entry \rightarrow A, A \rightarrow A, A \rightarrow A, A \rightarrow B)$$

unerreichbarer Zustände:

$$C, D$$

unerreichbare Transitionen:

$$A \rightarrow C, A \rightarrow D$$

zyklusinduzierend:

$$A$$

Ausführungszustand:

$$A \text{ mit } a = 2$$

Überschattete Transitionen:

$$A \rightarrow D$$

2.2.2 Definitionen zu starken Zusammenhangskomponenten

Die folgenden Definitionen beschreiben starke Zusammenhangskomponenten auf Graphen. Weiterhin sind die Definitionen auch auf Statecharts anwendbar, da diese gerichtete Graphen darstellen, bei denen die Zustände die Knoten und die Transitionen die Kanten sind.

2.2.13 Definition (Stark zusammenhängender Graph)

Ein gerichteter Graph $G = (V, E)$ ist genau dann *stark zusammenhängend*, wenn für jeden Knoten innerhalb des Graphen ein Pfad zu jedem anderen Knoten existiert (vgl. [22]).

2.2.14 Definition (Maximal induzierter Teilgraph)

Ein Graph $H = (V_1, E_1)$ heißt genau dann *maximal induzierter Teilgraph* eines Graphen $G = (V, E)$, wenn gilt (vgl. [22]):

1. $V_1 \subseteq V$ und $E_1 \subseteq E$, also dass seine Knoten- und Kantenmenge Teilmengen der jeweiligen Menge von G sind, und
2. $\forall(u, v) \in E : (u \in V_1 \wedge v \in V_1 \implies (u, v) \in E_1)$, also dass E_1 alle Kanten zwischen Knoten aus V_1 enthält, die schon in E vorhanden sind.

2.2.15 Definition (Starke Zusammenhangskomponente)

Ein Teilgraph $H = (V_1, E_1)$ eines gerichteten Graphen $G = (V, E)$ ist genau dann eine *starke Zusammenhangskomponente* (*strongly connected component, scc*), wenn H ein maximal induzierter Teilgraph von G ist, welcher stark zusammenhängend ist (vgl. [22]).

2.2.16 Definition (Zyklusinduzierende starke Zusammenhangskomponenten)

Eine starke Zusammenhangskomponente SCC ist genau dann *zyklusinduzierend*, wenn SCC einen Zustand enthält, welcher zyklusinduzierend ist.

2.2.17 Lemma (Folgerung zu zyklusinduzierenden scc)

Eine starke Zusammenhangskomponente $SCC = (V, E)$ ist genau dann zyklusinduzierend, wenn gilt:

1. $|V| > 1$, oder
2. $V = \{X\}$ und X besitzt eine Selbsttransition.

Zusätzlich gilt offensichtlich, dass jeder Zustand einer zyklusinduzierenden starken Zusammenhangskomponente wieder zyklusinduzierend ist.

Beweis „ \Leftarrow “:

Im ersten Fall enthält V mindestens zwei Zustände X und Y . Da diese Teil von SCC sind, existiert ein Pfad $P_{x \rightarrow y}$ von X nach Y , welcher mit der Transition t beginnt. Aus dem selben Grund existiert zusätzlich ein Pfad $P_{y \rightarrow x}$, welcher von Y nach X geht. Daraus

lässt sich der Zyklus $P_{x \rightarrow y} \cdot P_{y \rightarrow x} \cdot t$ konstruieren und es folgt, dass X und somit SCC zyklusinduzierend ist.

Im zweiten Fall ist X zyklusinduzierend und damit auch SCC .

„ \implies “:

Gilt umgekehrt weder 1. noch 2., so folgt entweder, dass $V = \emptyset$ gilt und damit SCC offensichtlich nicht zyklusinduzierend ist. Andernfalls folgt $V = \{X\}$, wobei X keine Selbsttransition enthält. Wir zeigen, dass X nicht zyklusinduzierend ist und daraus folgt die Aussage. Angenommen es gäbe einen Zyklus Z der in X beginnt. Falls Z die Form $Z = (t_0, t_0)$ besitzt, widerspricht dies der Tatsache, dass X keine Selbsttransition besitzt. Daher muss Z die Form $Z = (t_0, t_1, \dots, t_n, t_0)$ besitzen. Damit gibt es einen Pfad (t_0) von X zu einem Zustand Y , welcher die eingehende Transition t_0 besitzt. Zusätzlich existiert ein Pfad (t_1, \dots, t_n, t_0) von Y nach X . Daraus folgt, dass SCC keine starke Zusammenhangskomponente ist, da aus der Menge $\{X, Y\}$ eine starke Zusammenhangskomponente konstruierbar ist, welche SCC enthält. \square

Beispiele Ein Beispiel für starke Zusammenhangskomponenten ist in Abbildung 2.5 erkennbar. Diese werden durch die Knoten im zweiten Bild dargestellt, die eine Kante zueinander besitzen. Zu erkennen ist, dass jeder Knoten des ursprünglichen Graphen in einer starken Zusammenhangskomponente enthalten ist. Weiterhin ist jeder Knoten außer A zyklusinduzierend. Der Knoten L ist durch die Selbsttransition zyklusinduzierend.

2.3 Grundlagen symbolischer Ausführung

In diesem Abschnitt wird die symbolische Ausführung (*symbolic execution*), wie sie in [21] beschrieben wird, erläutert. Eine konkrete Ausführung eines Programms wird mittels konkreter Eingaben durchgeführt, welches zu einem einzelnen Ausführungspfad führt. Um auf diese Art alle Ausführungspfade zu erreichen, müssten alle konkreten Eingaben des Programms durchgeführt werden, welches häufig durch die Anzahl der Möglichkeiten keine Option ist. Bei dem Versuch mit Kontextwissen die richtigen Eingabekombinationen zu wählen, die alle Ausführungspfade erreichen, kann es leicht passieren, dass Pfade ausgelassen werden.

Im Gegensatz dazu arbeitet die symbolische Ausführung nicht mit konkreten, sondern mit symbolischen Eingabewerten. Mit diesen wird die Programmausführung gestartet, wobei für jede verschiedene Möglichkeit der Ausführung ein neuer Ausführungspfad traversiert wird. Dies erlaubt der Symbolic Execution Engine – dem Programm welches die symbolische Ausführung steuert – mehrere Ausführungspfade gleichzeitig zu traversieren, welche unter verschiedenen konkreten Eingaben zustande gekommen wären. Insgesamt führt die Engine so alle möglichen Ausführungspfade durch. Traditionell verwaltet die Engine für jeden Ausführungspfad folgende Informationen:

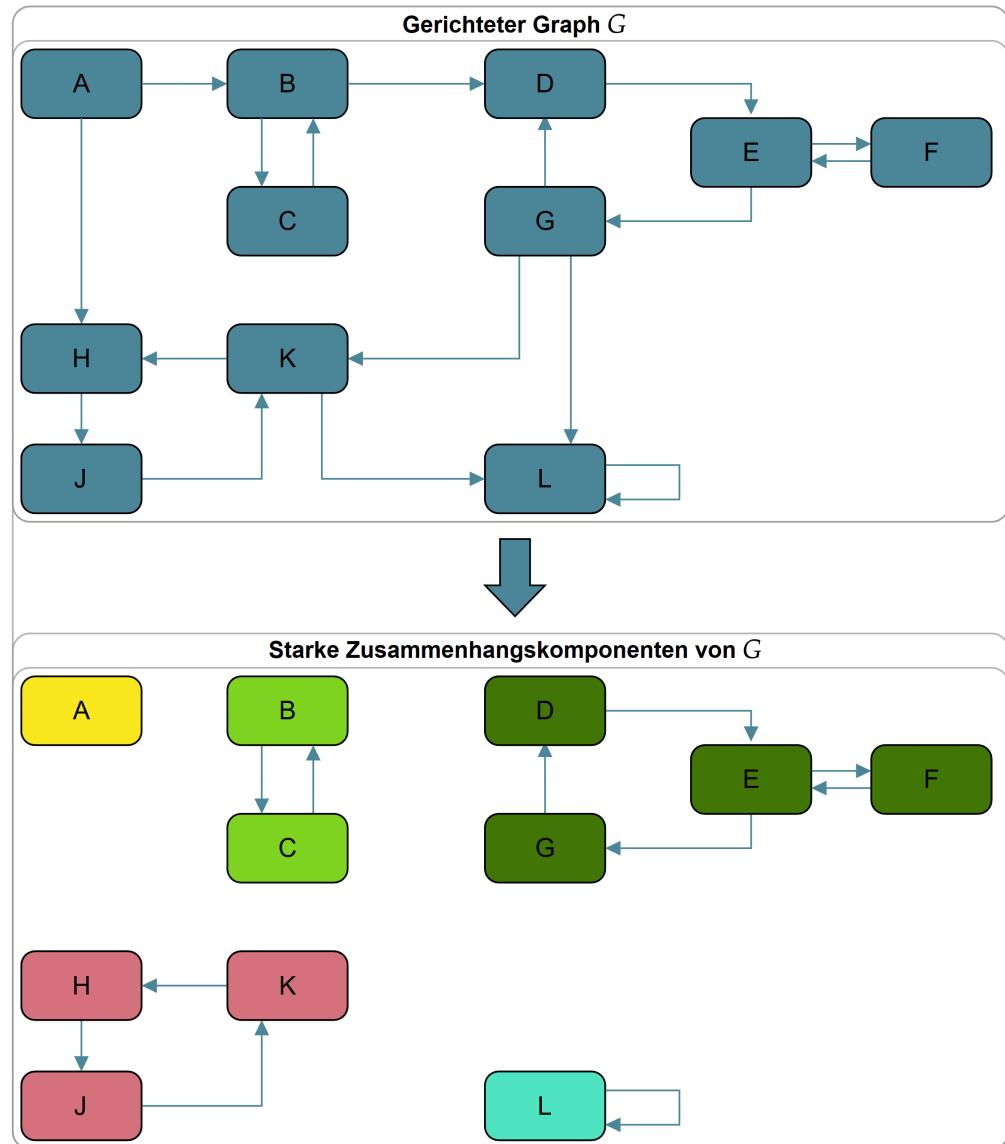


Abbildung 2.5: Beispiel starker Zusammenhangskomponente eines Graphen

- Die **Pfadbedingung**, dargestellt durch eine prädikatenlogische Formel erster Ordnung. Beschreibt, wie die Eingabewerte aussehen, um den aktuellen Zustand zu erreichen. Beinhaltet häufig den *symbolischen Speicher*, der ausdrückt, welche Variablen auf welche symbolischen Ausdrücke bzw. symbolischen Repräsentationen abgebildet werden.
- Die **Anweisung**, welche als nächstes bearbeitet wird und zeigt so den momentanen Stand in der Programmausführung.

Weiterhin wird die Erfüllbarkeit der Pfadbedingung überprüft, um zu schauen, ob der Ausführungspfad valide ist. Dabei wird untersucht, ob ein konkretes Modell der symbolischen Ausdrücke existiert, welches die Pfadbedingung erfüllbar macht. Häufig wird dabei

ein *satisfiability modulo theories* (SMT) *solver* [4] eingesetzt. Diese sind durch den generischen Aufbau ihrer zugrunde liegenden Algorithmen bestens in der Lage, viele verschiedene und komplexe Bedingungen, wie beispielsweise Operationen auf Arrays, zu lösen [21].

Bei symbolischer Ausführung müssen die folgenden Herausforderungen gelöst werden:

Ausführungsumgebung

Agiert die Engine mit einer open oder closed world assumption? Aufrufe von Bibliotheken, Systemfunktionen oder ähnliches können Seiteneffekte erzeugen, welche die Ausführung der Engine beeinflusst. Diese Effekte müssen der Engine zur Verfügung stehen, wobei eine Beachtung aller möglichen Interaktionen unpraktikabel ist.

Pfadexplosion

Wie behandelt die Engine Pfadexplosion? Konstrukte wie Zyklen können die Anzahl der Ausführungswege exponentiell erhöhen. Es ist unwahrscheinlich, dass die Engine alle möglichen Pfade in einer handhabbaren Zeit evaluiert.

Erfüllbarkeitsbestimmung

Wie gut können SMT-Solver mit der Größe und Komplexität von Pfadbedingungen skaliieren?

Speicherverwaltung

Wie speichert die Engine Arrays oder andere komplexe Objekte?

Je nachdem in welchem Kontext die symbolische Ausführung benutzt wird, müssen verschiedene Annahmen getätigt werden, um die oben genannten Herausforderungen zu lösen. Diese beeinflussen die Korrektheit und Vollständigkeit der Ausführung, woraus ein Kompromiss zwischen diesen und der benötigten Zeit entsteht.

In dieser Arbeit wird eine symbolische Programmausführung im Kontext von Statecharts durchgeführt. Nachfolgend werden alle verwalteten Informationen, der Ablauf der Engine und die Annahmen, die getroffen werden, erläutert.

Kapitel 3

Symbolic Execution

In dieser Arbeit wurde eine symbolische Programmausführung auf Zustandsautomaten entwickelt. Die Ergebnisse der Ausführung werden in einem sogenannten symbolischen Ausführungsbaum gespeichert, auf welchem eine Erreichbarkeitsanalyse angeboten wird. Konkret identifiziert die Engine unerreichbare Zustände und Transitionen und zeigt diese einem Nutzer interaktiv während der Modellierung an. Die Engine wurde aufbauend auf dem in [11] vorgestellten Algorithmus, aber mit einer komplett neuen Implementierung entwickelt. Die Ziele bei der Entwicklung waren hauptsächlich Verbesserungen an der Genauigkeit und Ausführungszeit. Weiterhin wurde die Engine mit einer modularen Bauweise entwickelt, sodass Verbesserungen, weitere Optimierungen oder Änderungen an der Strategie der Programmausführung einfach integrierbar und umsetzbar sind.

3.1 Allgemeiner Ablauf

Der allgemeine Ablauf der Ausführung ist in Abbildung 3.1 dargestellt. Zuerst wird die Eingabe, ein Statechart, vorverarbeitet und vereinfacht (3.2). Das entstandene Statechart wird für die Konstruktion des symbolischen Ausführungsbaumes (*symbolic execution tree, SET*) genutzt (4). Dieser wird im nächsten Schritt auf unerreichbare Zustände und Transitionen analysiert (8.1), welche schließlich im ursprünglichen Statechart markiert werden.

In diesem Kapitel werden weiterhin die Vorverarbeitung und die Limitierungen der Ausführung besprochen.

3.2 Vorverarbeitung

Dieser Abschnitt beschreibt die Vorverarbeitung des zu analysierenden Statecharts und die daraus resultierenden Vorbedingungen, die bei der anschließenden Konstruktion des SET im vereinfachten Statechart gelten. Die einzelnen Schritte der Verarbeitung sind in Abbil-

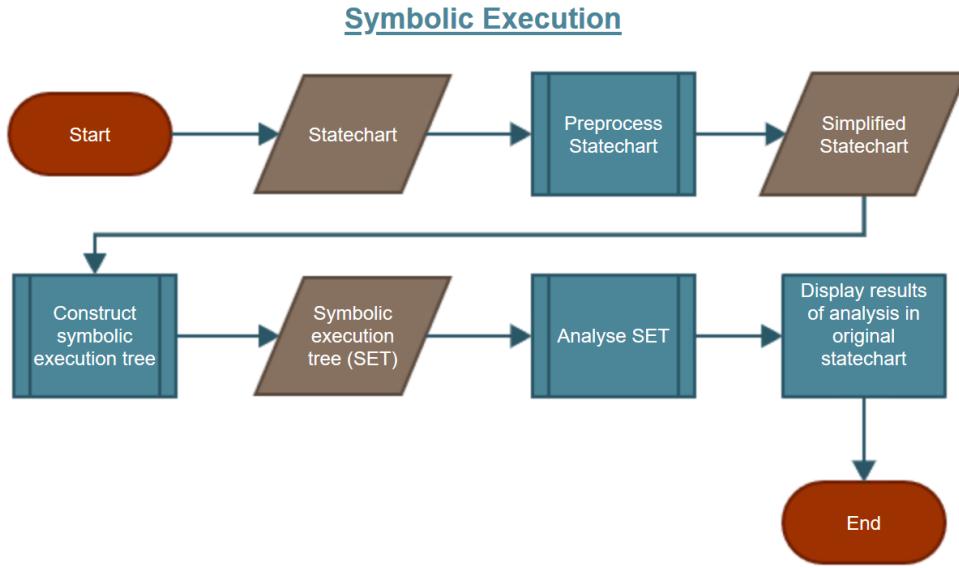


Abbildung 3.1: Allgemeiner Ablauf der symbolischen Ausführung als Flowchart

dung 3.2 zu erkennen und werden nachfolgend kurz erläutert. Eine genaue Verfahrensweise ist nicht Teil dieser Arbeit und wird daher nicht angegeben.

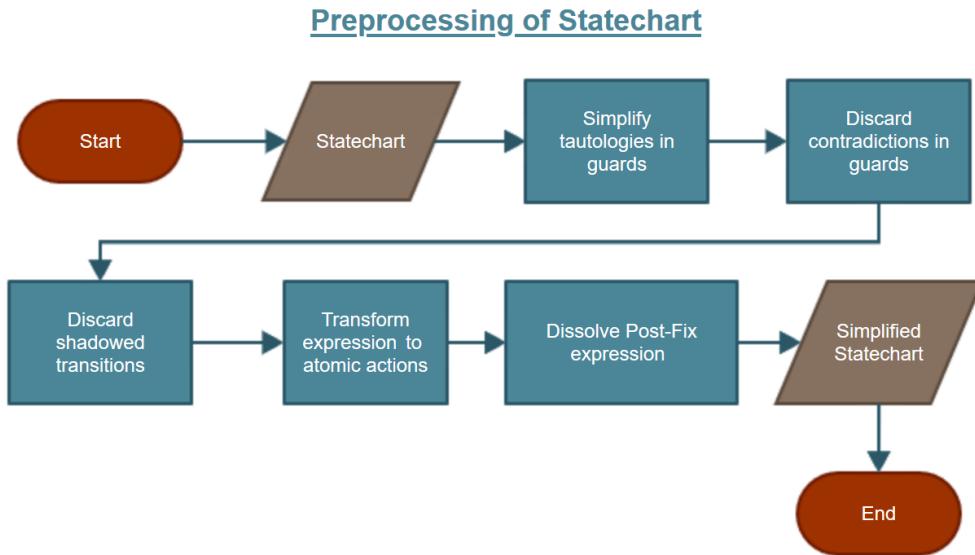


Abbildung 3.2: Ablauf der Vereinfachung des Statecharts dargestellt als Flowchart.

3.2.1 Vereinfachung von Transitionen

Die Vereinfachung von Transitionen umfasst die Schritte *Simplify tautologies in guards*, *Discard contradictions in guards* und *Discard shadowed transitions* des Flowcharts. Guards innerhalb von Reaktionen werden auf zwei Merkmale untersucht. Zum einen wird über-

prüft, ob ein Guard eine Tautologie darstellt. In diesem Fall wird der Guard durch `true` ersetzt. Anschließend wird überprüft, ob ein Guard einen Widerspruch darstellt. Dann wird die komplette Reaktion gelöscht, da diese niemals durchführbar ist. Abschließend werden alle überschatteten Transitionen (definiert in 2.2.12) entfernt, da diese nicht durchführbar sind. Abbildung 3.3 zeigt beispielhaft die Vereinfachung von Transitionen an einem Statechart.

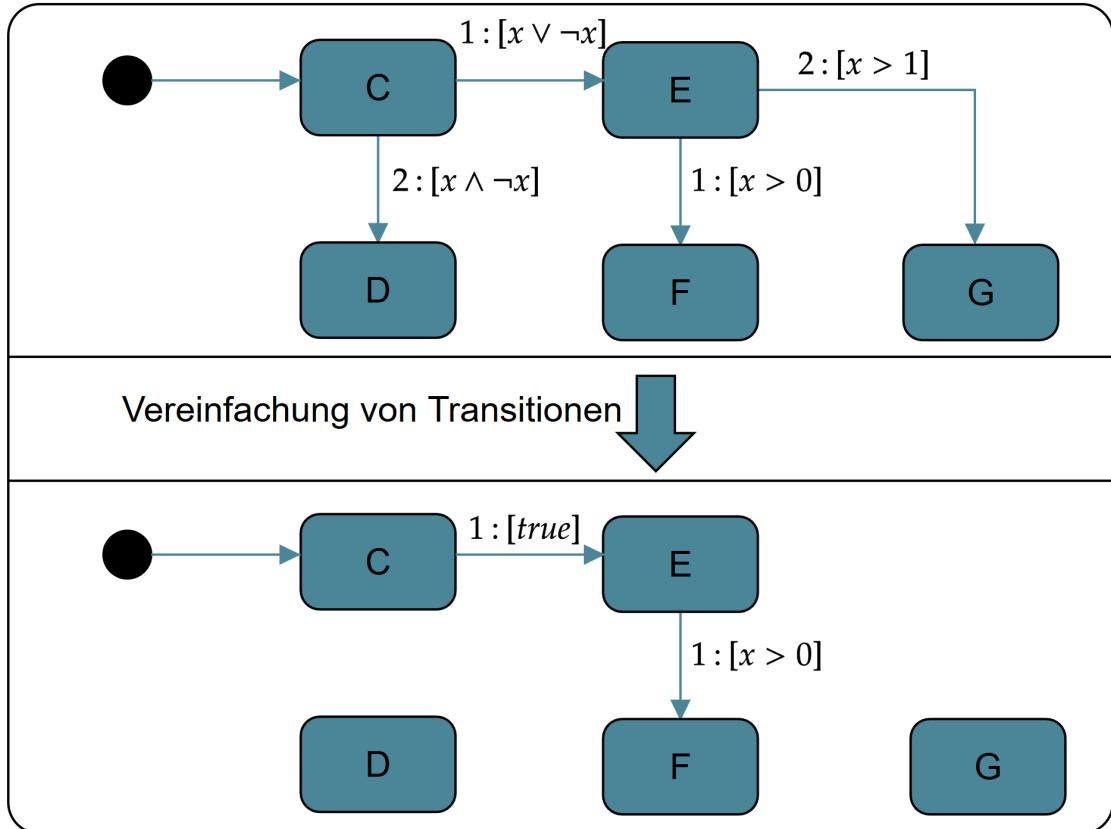


Abbildung 3.3: Beispielhafte Vereinfachung von Transitionen eines Statecharts. Die Zahlen an den Transitionen beschreiben die Prioritäten. Der Guard der Transition $C \rightarrow D$ wird als Widerspruch erkannt und entfernt. Der Guard der Transition $C \rightarrow E$ wird als Tautologie erkannt und zu `true` vereinfacht. Die Transition $E \rightarrow G$ wird von der Transition $E \rightarrow F$ überschattet und entfernt.

3.2.2 Vereinfachung von Expressions

Die nächsten beiden Schritte *Transform expression to atomic actions* und *Dissolve Post-Fix expression* lassen sich als Vereinfachung von Expressions zusammenfassen. Zuerst werden alle Expressions zu atomaren Aktionen transformiert, welche zwei atomare Variablen – Konstanten oder Variablen – mit einem Operator wie z.B. $+$ oder \wedge verbinden. Zusätzlich sind Zuweisungen zu atomaren Aktionen erlaubt. Ausgenommen sind **Conditional-**

Expressions, diese sind erlaubt und werden bei der Konstruktion des SET explizit verarbeitet. Weiterhin werden alle Post-Fix Ausdrücke innerhalb von Expressions aufgelöst und zusätzliche Zuweisungen für diese eingeführt.

In dieser Arbeit wird davon ausgegangen, dass alle Expressions des vereinfachten Statecharts die vorher genannte Form besitzen. Der Ablauf der Transformation wird in dieser Arbeit aber nicht behandelt. In Abbildung 3.4 sind mehrere Beispiele zu erkennen, wie das Ergebnis der Transformation aussehen könnte. Zu beachten ist, dass bisherige Variablennamen gleich bleiben und zusätzliche Variablen und Zuweisungen eingeführt wurden.

1 $a = ((b * c) + 10) * d$ $\left\{ \begin{array}{l} a_1 = b * c \\ a_2 = a_1 + 10 \\ a = a_2 * d \end{array} \right.$	2 $x = \neg((v \wedge w) \vee y)$ $\left\{ \begin{array}{l} x_1 = v \wedge w \\ x_2 = x_1 \vee y \\ x = \neg x_2 \end{array} \right.$
3 $a++$ $\left\{ \begin{array}{l} a = a + 1 \end{array} \right.$	4 $a * = b + c$ $\left\{ \begin{array}{l} a_1 = b + c \\ a = a * a_1 \end{array} \right.$

Abbildung 3.4: Beispiele zur Transformation von Expressions. Links steht jeweils die ursprüngliche Expression, rechts die transformierte. 1 zeigt die Transformation einer Numerischen Expression, 2 die einer Booleschen. Beispiele 3 und 4 zeigen die Auflösung von Post-Fix Ausdrücken

Aus dieser Vorverarbeitung resultiert ein vereinfachtes Statechart, welches nachfolgend genutzt wird, um den SET zu konstruieren.

3.3 Limitierungen

Aufgrund der begrenzten Entwicklungszeit ist die Engine nicht mit allen Statechart Elementen kompatibel. Es wurde einige Features weggelassen, um einen größeren Fokus auf die theoretischen Aspekte der Arbeit, wie die Pfadexplosion und Optimierungen der Erfüllbarkeitsbestimmung, zu ermöglichen. Folgende Elemente sind nicht kompatibel:

- komposite und orthogonale Zustände,
- multiple Regionen,
- History Zustände,
- Exit-, Choice und Synchronisationsknoten,
- after und every Trigger
- real, string und void Variablentypen
- ShiftExpression

- `BitwiseAndExpression`, `BitwiseOrExpression` und `BitwiseXorExpression`
- eingebaute Funktionen: `valueof(event)`, `as`, `active(state)`
- Multiple Trigger einer Reaktion

Einige dieser Elemente, wie zum Beispiel Multiple oder `after` und `every` Trigger werden die Engine nicht zum abstürzen bringen. Trotzdem werden diese nicht unterstützt, da bei ihnen die Korrektheit der Engine nicht überprüft wurde.

3.3.1 Ausführungsumgebung

Ein Hauptproblem symbolischer Ausführung ist die Behandlung der Ausführungsumgebung. In dieser Arbeit wurde sich dafür entschieden, die Analyse externer Funktionen nicht zu unterstützen, welche durch `operations` im Yakindu ScT Kontext umgesetzt werden. Weiterhin wird die Behandlung eingehender Events unterstützt. Hier wird angenommen, dass jedes Event zu jedem Zeitpunkt aktiv sein kann.

Kapitel 4

Symbolischer Ausführungsbaum

In diesem Abschnitt wird der symbolische Ausführungsbaum (*symbolic execution tree, SET*) und seine Bestandteile erläutert. Jeder Pfad des SET beschreibt einen symbolischen Ausführungspfad des Statecharts. Ein Knoten innerhalb des SET spiegelt den derzeitigen Stand der Ausführung im jeweiligen Pfad wieder. Kinderknoten entstehen durch die Anwendung von Regeln (4.2) an einem Knoten. Dabei gibt es unterschiedliche Arten von Regeln, zum Beispiel solche, die Transitionen durchführen und solche, die Expressions verarbeiten und den Pfadbedingungen hinzufügen.

4.1 Knoten

Ein Knoten innerhalb des SET verwaltet für einen Ausführungspfad folgende Informationen:

Pfadbedingung Die Pfadbedingung (*path constraints, PC*) besteht aus einer Liste von Expressions, welche zum derzeitigen Stand des Ausführungspfades gelten. Ein Knoten wird nachfolgend als plausibel bezeichnet, wenn dessen Pfadbedingung erfüllbar ist.

Symbolischer Speicher Der symbolische Speicher beinhaltet die Abbildungen von Variablen auf symbolische Repräsentationen dieser Variablen. Er wird als Liste verwaltet. Die letzte Repräsentation ist immer die aktuellste und wird für die SSA-Transformation 4.2.3 benutzt, bei der die Variablen von Expressions durch ihre symbolische Repräsentation ersetzt werden.

Transitionen Ein Knoten speichert eine Liste von Transitionen, die bis zum derzeitigen Zeitpunkt des Ausführungspfades durchgeführt wurden. Transitionen werden in der Durchführungsreihenfolge gespeichert und können mehrfach vorkommen. Die Liste wird beispielsweise benutzt, um Zyklen innerhalb des Ausführungspfades zu finden.

Aktiver Zustand Ein Knoten speichert den aktiven Zustand zum momentanen Zeitpunkt des Ausführungspfades. Dieser wird beispielsweise benutzt, um die nächsten durchführbaren Transitionen zu berechnen.

Erfüllbarkeit Zusätzlich wird die Erfüllbarkeit der Pfadbedingung des Knotens gespeichert, falls diese bereits bestimmt wurde. An dieser Information ist erkennbar, welche Knoten eines symbolischen Ausführungspfades schon auf Erfüllbarkeit getestet wurden und welche Pfade nicht valide sind. Die Erfüllbarkeit wird für die Erreichbarkeitsanalyse (8.1) und diverse Optimierungen bei der Erfüllbarkeitsbestimmung (6) benutzt.

Aktive Anweisungen Die aktiven Anweisungen sind eine geordnete Liste von Expression, welche vor der nächsten Transitionsdurchführung noch zu Pfadbedingungen verarbeitet werden müssen. Beispielsweise werden die Effekte bei der Durchführung einer Transition erst den aktiven Anweisungen hinzugefügt, wo sie der Reihe nach durch die Anwendung passender Regeln abgearbeitet werden.

Blockierte Reaktionen Um sicher zu stellen, dass symbolische Ausführungspfade endlich sind und dieselbe Transition nicht unendlich oft durchgeführt wird, werden Reaktionen durch die Zykleneliminationen (5) im Kontext der Erreichbarkeitsanalyse blockiert. Die Reaktionen werden von Knoten als eine Liste verwaltet und sind für den jeweiligen symbolischen Ausführungspfad nicht mehr durchführbar.

Elter Ein Knoten speichert eine Referenz seines Elter, wobei ein Knoten nicht mehr als ein Elter besitzt. Dies ermöglicht eine einfache Rückverfolgung der Vorfahren des Knoten.

Kinder Ein Knoten speichert Referenzen auf seine Kindknoten, die durch die Anwendung einer Regel produziert wurden. Die Kindknoten werden als Liste verwaltet.

Angewandte Regel Weiterhin wird die Regel gespeichert, welche auf einen Knoten angewandt wird. Durch diese lassen sich bestimmte Eigenschaften eines Knotens direkt ablesen, z.B. ob ein Knoten mehr als ein Kindknoten besitzen kann, oder ob eine neue Transition durchgeführt wurde.

In Abbildung 4.1 ist beispielhaft ein Knoten dargestellt, indem alle verwalteten Informationen aufgeführt sind. In nachfolgenden Abbildungen von Knoten werden immer nur die Informationen abgebildet, die im jeweiligen Kontext von besonderer Bedeutung sind. Beispielsweise werden diese bei der Konstruktion von Kindknoten verändert.

2	$pc : a_1 = a_0 + 2 \wedge a_1 > 0$	<i>active state : B</i>
	<i>sym. mem : $a \mapsto a_0, a \mapsto a_1$</i>	<i>parent : node 1</i>
	<i>act. statements : $b = a + 3$</i>	<i>children : node 3, node 4</i>
	<i>trans. : $Entry \rightarrow A, A \rightarrow B$</i>	<i>satisfiability : SAT</i>
	<i>rule : DefaultStatementRule</i>	<i>blocked reactions : $A \rightarrow C$</i>

Abbildung 4.1: Beispielhafter Knoten des SET mit allen verwalteten Informationen.

ImmutableList

Einige Informationen, welche als Liste gespeichert werden, sind in der Tiefe des Ausführungsbaumes monoton steigend. Das heißt, dass bei der Generierung von Kindknoten die selbe Liste wie beim Elter verwendet wird und höchstens Elemente an die Liste angehängt werden. Dadurch ist die Liste eines Knoten automatisch eine Teilmenge der gleichen Liste einer seiner Nachfahren. Folgende Informationen fallen in diese Kategorie: Pfadbedingung, Symbolischer Speicher, Transitionen und blockierte Reaktionen. Bei der Implementierung wurden diese Listen durch eine eigens entwickelte `ImmutableList` verwaltet. Diese enthält 2 Attribute, ein Element und eine Referenz auf eine weitere `ImmutableList`. Dies ermöglicht, dass bestehende Listen nicht veränderbar sind und nur weitere Elemente ergänzt werden können.

4.2 Regeln

In diesem Abschnitt werden die Regeln erläutert, welche auf Knoten des SET angewandt werden und für die Generierung von Kindknoten verantwortlich sind. Es kann sein, dass mehrere Regeln für einen Knoten anwendbar sind, schlussendlich wird die Regel durch die Strategie 7.3 ausgewählt. Da die Regeln unterschiedliche Voraussetzungen und Auswirkungen besitzen, ist aber häufig nur eine Regel anwendbar. Die Kriterien, ob eine Regel auf einen Knoten anwendbar ist, lassen sich deterministisch aus den Informationen des Knotens und seiner Vorfahren bestimmen. Beispielsweise gibt es unterschiedliche Regeln, welche die oberste aktive Anweisung verarbeiten. Bei einer `ConditionalExpression` an erster Stelle der aktiven Anweisungen ist die `ConditionalExpressionRule` anwendbar. Falls die Liste der aktiven Anweisungen leer ist, ist die `TransitionRule` anwendbar, welche für die Durchführung von Transitionen verantwortlich ist. Nachfolgend werden alle Regeln näher erläutert, beginnend mit der `TransitionRule`. Danach werden Regeln beschrieben, welche aktive Anweisungen vereinfachen und solche, welche aktive Anweisungen zu Pfadbedingungen verarbeiten. Abschließend werden die Zykleneliminationsregeln vorgestellt.

4.2.1 TransitionRule

Die `TransitionRule` ist für die Durchführung von Transitionen verantwortlich und ist immer dann anwendbar, wenn die aktiven Anweisungen leer sind. Der Anwendungsablauf ist in Abbildung 4.2 dargestellt. Grundsätzlich wird für jede ausgehende, nicht blockierte Transition des aktiven Zustandes ein Kindknoten erstellt, der alle zugehörigen Guards und Effekte beinhaltet. Dies wird in 4.2.1 genauer beschrieben. Abschließend wird weiter in 4.2.1 erläutert, wie die lokalen Reaktionen des aktiven Zustandes verarbeitet werden. Um diese beiden Abläufe zu verstehen, wird zuerst beschrieben, wie Reaktionen allgemein als `ConditionalExpressions` dargestellt und so den aktiven Anweisungen hinzugefügt werden.

Die `TransitionRule` ist auch anwendbar, wenn der aktive Zustand keine ausgehende Transition bzw. lokalen Reaktionen besitzt. In diesem Fall wird kein Kindknoten produziert. Durch die Kriterien der Anwendbarkeit der `TransitionRule` ist immer sicher gestellt, dass bei Blättern eines erfüllbaren symbolischen Ausführungspfades des komplett konstruierten SET die `TransitionRule` angewandt wurde. Genauer wird dies in 7.4 erläutert.

Reaktionen als Expression

Bei Durchführung einer Reaktion wird deren Guard und Effekte den aktiven Anweisungen hinzugefügt. Diese werden je nach Reaktion unterschiedlich behandelt. Wie bereits erwähnt, wird für jede ausgehende Transition des aktiven Zustandes ein Kindknoten erstellt. Wenn beispielsweise der Guard einer Transition nicht erfüllbar ist, muss der zugehörige Kindknoten unplausibel sein, da die Transition nicht durchführbar und damit der Pfad invalide ist. Um dies abzubilden wird eine Transitionsreaktion $\text{trigger}[guard]/expr_1; \dots, expr_n$ als `SequenceBlockExpression` $[guard; expr_1; \dots; expr_n]$ umgewandelt. Weiterhin werden die Expressions $expr_i$, welche Boolesche Expressions sind, entfernt, da diese keine Auswirkungen als Effekte haben und später nicht den Pfadbedingungen hinzugefügt werden dürfen. Insgesamt ergibt sich, dass der Guard durch spätere Regelanwendung den Pfadbedingungen hinzugefügt wird. Diese ist damit unerfüllbar, falls der Guard zu `false` ausgewertet wird.

Anders sieht es bei lokalen Reaktionen aus. Hier soll erreicht werden, dass die Effekte bei Unerfüllbarkeit des Guards nicht durchgeführt werden. Der jeweilige Knoten muss aber trotzdem plausibel bleiben. Daher wird eine lokale Reaktion $\text{trigger}[guard]/expr_1; \dots, expr_n$ als `ConditionalExpression` $\text{guard}? [expr_1; \dots, expr_n] : \text{true}$ umgewandelt. Falls der Guard erfüllbar ist, wird dieser und die `SequenceBlockExpression` $[expr_1; \dots; expr_n]$ den aktiven Anweisungen hinzugefügt. Andernfalls wird der negierte Guard und `true` den aktiven Anweisungen hinzugefügt.

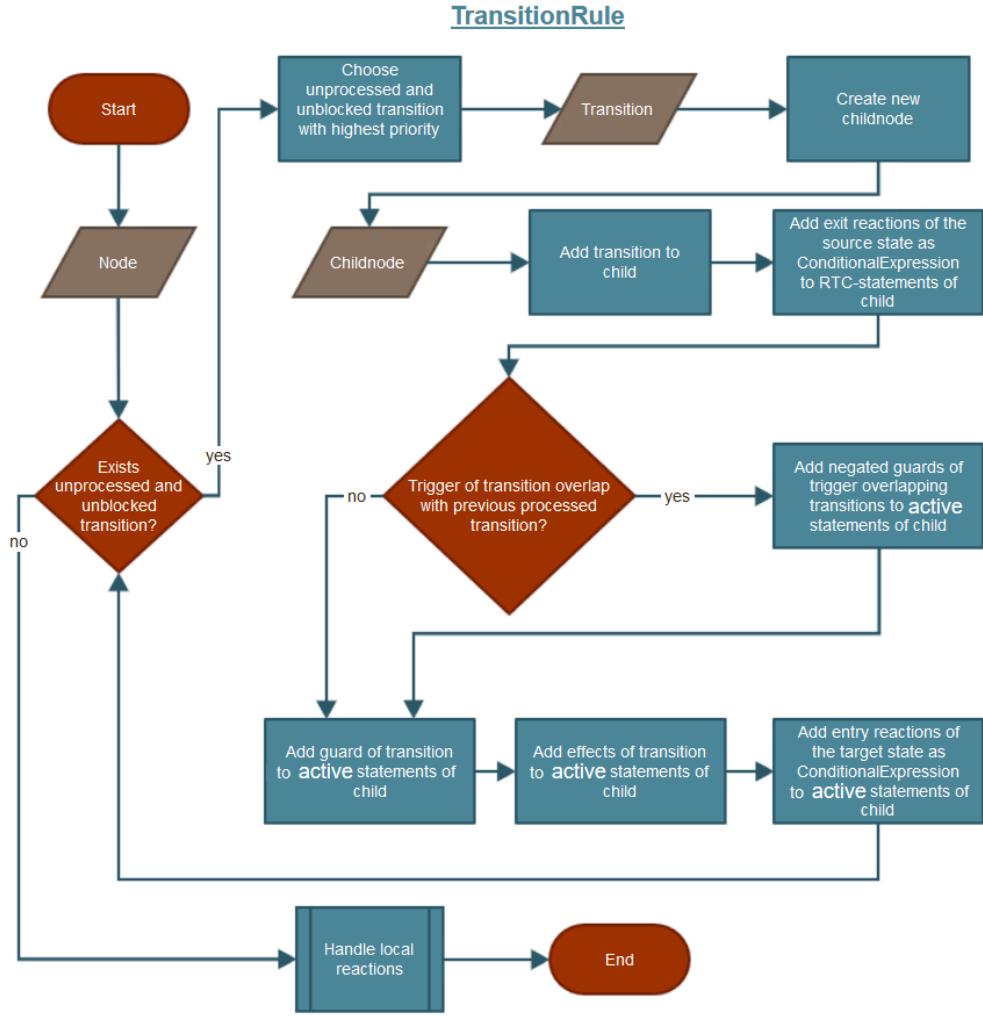


Abbildung 4.2: Ablauf der TransitionRule als Flowchart

Beispielhaft sind diese Expressions in Abbildung 4.3 erkennbar. In Knoten 2 des SET ist die **exit**-Reaktion als **ConditionalExpression** $x?[a = b] : true$ dargestellt. Der Guard und die Effekte der Transition 3 ergibt sich in Knoten 4 als $[a > 0; a = a + 2]$.

Transitionsdurchführung mit Prioritäten

Wie bereits erwähnt, wird für jede ausgehende, nicht blockierte Transition des aktiven Zustandes ein Kindknoten erstellt. Nachfolgend wird der Ablauf der Erstellung eines Kindknoten mit einer Transition t beschrieben, wie er in Abbildung 4.2 erkennbar ist. t wird dabei den Transitionen des Kindes hinzugefügt. Zusätzlich werden die Guards und Effekte, die bei der Durchführung der Transition auftreten, den aktiven Anweisungen hinzugefügt. Dabei ist wichtig, dass die Priorisierung der Transitionen mitberücksichtigt wird.

Zuerst werden mögliche **exit**-Reaktionen des Ausgangszustandes als **ConditionalExpression** umgewandelt zu den aktiven Anweisungen hinzugefügt. Die Umwandlung wur-

de zuvor in 4.2.1 beschrieben. Falls es weitere vom Ausgangszustand ausgehende Transitionen mit einer höheren Priorität und dem selben Trigger wie t gibt, werden dessen negierte Guards den aktiven Anweisungen hinzugefügt. Dadurch werden die Pfadbedingungen der Transitionen unerfüllbar, die durch ihre niedrige Priorität nicht durchführbar sind. Danach wird t als **ConditionalExpression** den aktiven Anweisungen hinzugefügt. Abschließend werden mögliche entry-Reaktionen des Zielzustandes als **ConditionalExpression** umgewandelt ergänzt. Abbildung 4.3 zeigt ein Beispiel der Transitionsdurchführung durch die Anwendung der **TransitionRule**.

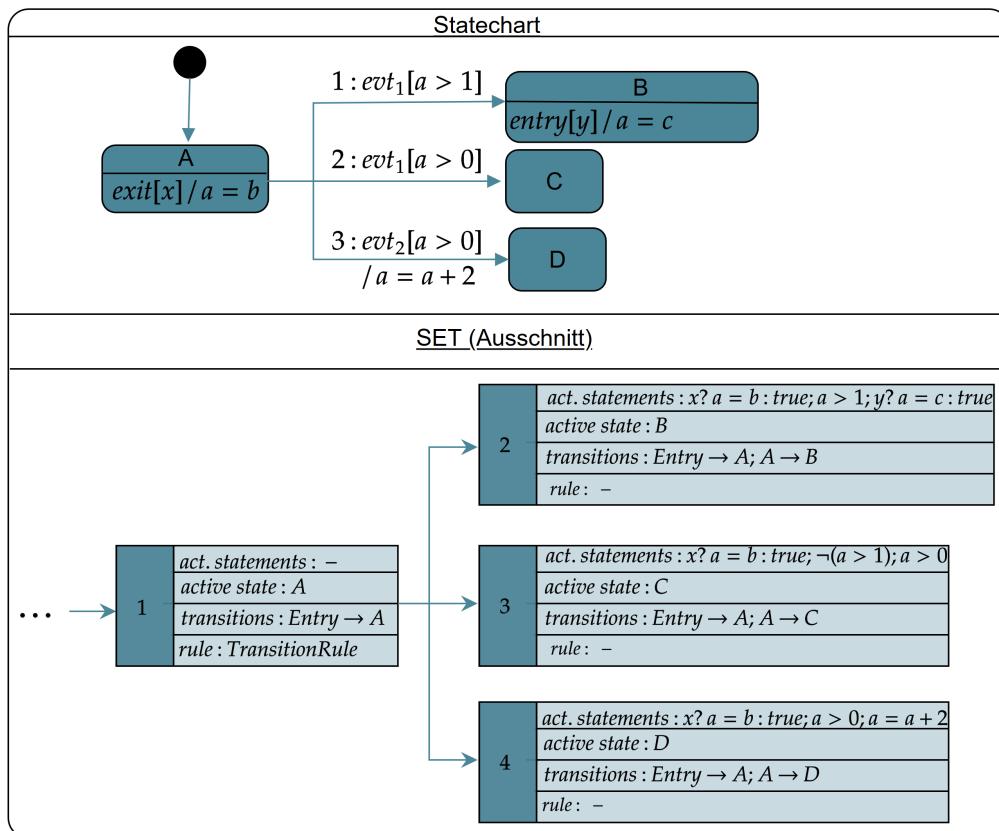


Abbildung 4.3: Beispielhafte Anwendung der **TransitionRule**, welche die Durchführung von Transitionen zeigt. Knoten 2 enthält die exit-und entry-Reaktion als **ConditionalExpression**. Bei Knoten 3 wird der negierte Guard $\neg(a > 1)$ von Transition 1 ergänzt, da die Transitionen 2 und 1 den selben Trigger besitzen. Der Trigger von Transition 3 überlappt mit keinem anderen Trigger.

Durchführung lokaler Reaktionen

Als nächstes wird beschrieben, wie lokale Reaktionen des aktiven Zustandes verarbeitet werden. Der Ablauf ist in Abbildung 4.4 dargestellt. Zuerst werden alle Trigger unblockierter Reaktionen gesammelt. Für jede mögliche Teilmenge von Triggern T wird als nächstes

ein Kindknoten erstellt. Damit wird jede mögliche Kombination an Triggern berücksichtigt, die gleichzeitig vorliegen können. Wie bei den Transitionen werden den aktiven Anweisungen des Knotens die negierten Guards der Transitionen hinzugefügt, die einen Trigger aus T besitzen. Danach werden alle lokale Reaktionen, die durch einen Trigger aus T ausgelöst werden, als **ConditionalExpression** umgewandelt den aktiven Anweisungen hinzufügt. Dieses Vorgehen ist beispielhaft in Abbildung 4.5 erkennbar.

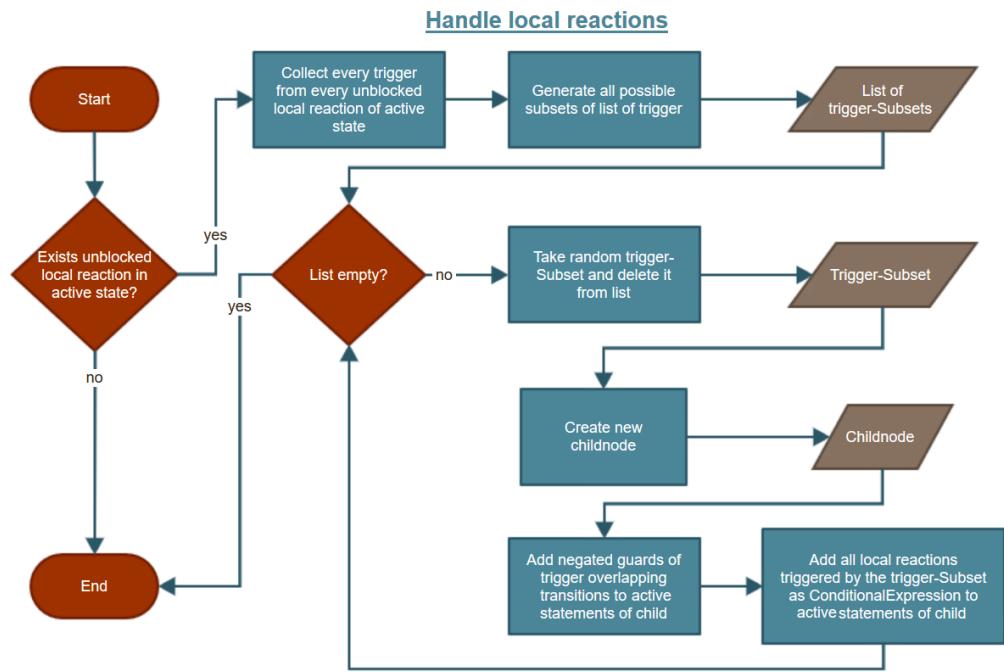


Abbildung 4.4: Ablauf der Durchführung lokaler Reaktionen bei der Anwendung der **TransitionRule** als Flowchart

4.2.2 Regel für die Vereinfachung aktiver Anweisungen

In diesem Abschnitt werden die Regeln betrachtet, welche für die Verarbeitung aktiver Anweisungen zuständig sind. Jede Regel behandelt immer die oberste aktive Anweisung, welche bei den generierten Kindknoten von der Liste entfernt wird. Dabei ist eine Regel immer für bestimmte Instanzen von Expressions zuständig. Beispielsweise ist die **ConditionalRule** anwendbar, wenn die oberste Expression eine Instanz von **ConditionalExpression** ist. Die vereinfachten Expressions werden anschließend an erster Stelle der aktiven Anweisungen hinzugefügt.

ConditionalRule

Die **ConditionalRule** ist dann anwendbar, wenn das oberste Element der aktiven Anweisungen eine **ConditionalExpression** ist. Zur Erinnerung, eine **ConditionalExpression**

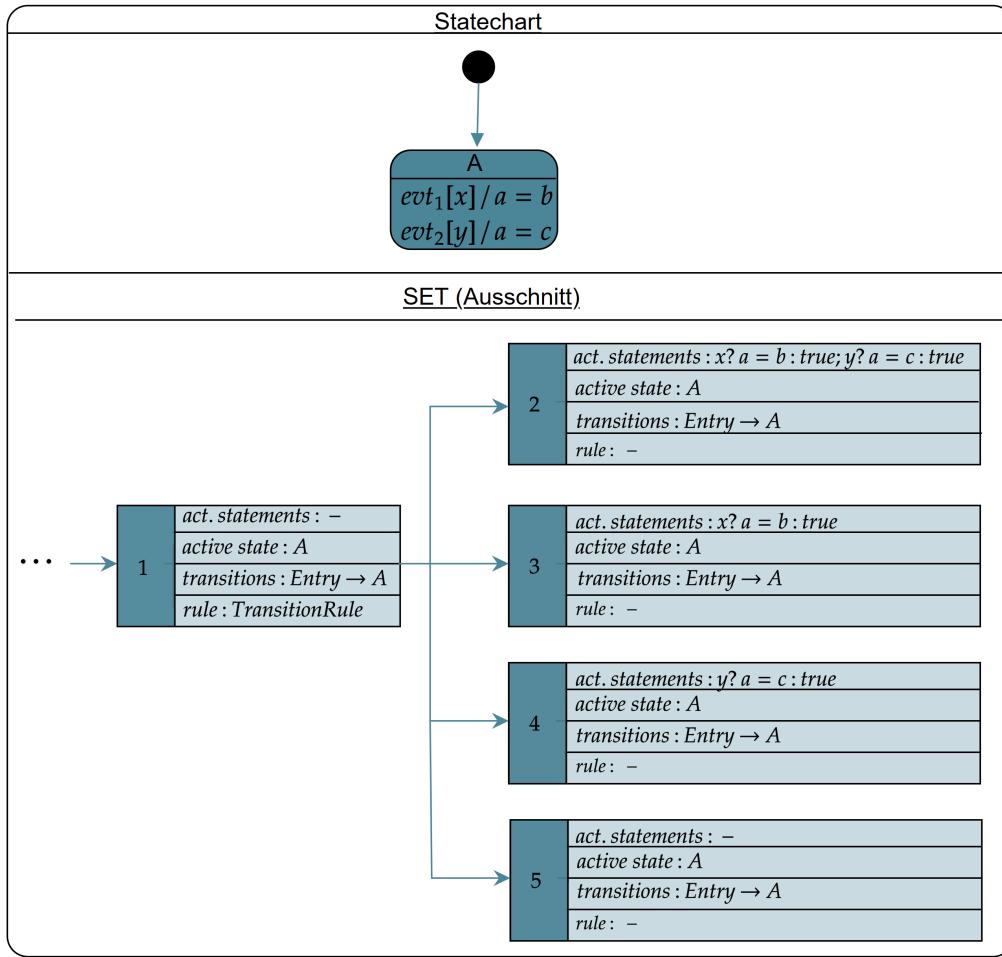


Abbildung 4.5: Beispielhafte Anwendung der `TransitionRule`, welche die Durchführung lokaler Reaktionen zeigt. Für jede Teilmenge der Triggermenge $\{evt_1, evt_2\}$ entsteht ein Kindknoten, bei welchem die lokalen Reaktionen durchgeführt wurden. Beispielsweise ist in Knoten 4 die lokale Reaktion $evt_2[y]/a = c$ durch die Teilmenge $\{evt_2\}$ durchgeführt worden.

besteht aus einer Bedingung, einer `TrueCaseExpression`, die bei Auswertung der Bedingung zu `true` gilt, und einer `FalseCaseExpression`, die bei Auswertung zu `false` gilt.

Bei Anwendung der Regel werden zwei Kindknoten erzeugt. Dem „`true`“-Kind, werden die Bedingung und die `TrueCaseExpression` den aktiven Anweisungen hinzugefügt. Dem „`false`“-Kind wird die negierte Bedingung und die `FalseCaseExpression` hinzugefügt.

Abbildung 4.6 zeigt die Anwendung der Regel mit den relevanten Informationen an den Knoten. Die aktiven Anweisungen Π , die hinter der `ConditionalExpression` $x? y : z$ stehen, bleiben unberührt. Bei den Kindern wird die Expression entfernt und jeweils die zwei Expressions $x; y$ bzw. $\neg x; z$ zu den aktiven Anweisungen hinzugefügt. Wichtig zu erkennen ist, dass die Pfadbedingung φ unverändert bleibt, da diese Regel nur die `ConditionalExpression` in zwei separate Fälle aufteilt und vereinfacht.

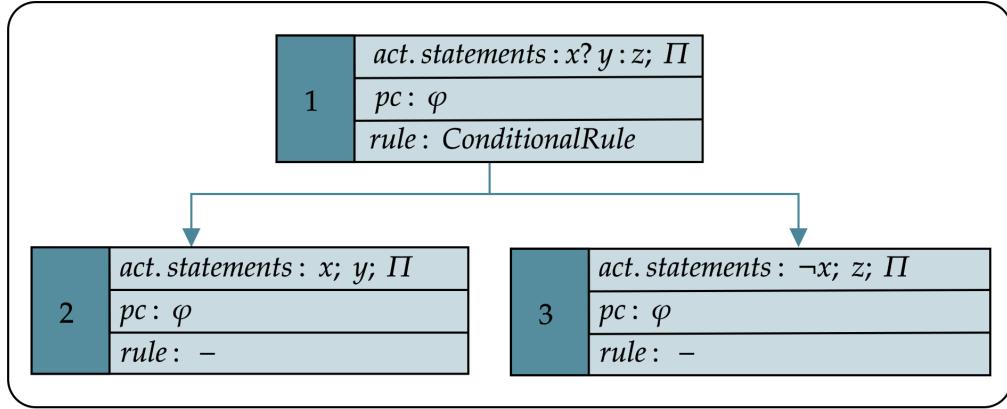


Abbildung 4.6: Anwendung der ConditionalRule mit ConditionalExpression $x?y : z$. Dem „true“-Kind 2 wurde die Bedingung x und TrueCaseExpression y den aktiven Anweisungen hinzugefügt. Dem „false“-Kind 3 die negierte Bedingung $\neg x$ und FalseCaseExpression z .

AssignmentConditionalRule

Die AssignmentConditionalRule ist anwendbar, wenn die oberste Expression der aktiven Anweisungen eine AssignmentExpression ist und gleichzeitig die rechte Seite der Zuweisung, der Assignor, eine ConditionalExpression ist. Wie bei der ConditionalRule werden zwei Kinder erzeugt. Die Expression wird zu zwei Expressions vereinfacht, zur Bedingung und TrueCaseAssignmentExpression bzw. negierter Bedingung FalseCaseAssignmentExpression. Das eine Kind enthält die Expressions im Falle der Auswertung der Bedingung zu `true`, das andere zu `false`.

Die Abbildung 4.7 zeigt die Anwendung der Regel, welche sich an der ConditionalRule orientiert. In diesem Fall wird aber jeweils die Zuweisung $a = y$ bzw. $a = z$ den aktiven Anweisungen hinzugefügt.

SequenceBlockRule

Die SequenceBlockRule ist anwendbar, wenn das erste Element der aktiven Anweisungen eine SequenceBlockExpression ist. Zur Erinnerung, die SequenceBlockExpression ist eine für die Symbolic Execution Engine entwickelte Expression, die eine geordnete Liste von Expressions verwaltet. Bei Anwendung der Regel wird die SequenceBlockExpression von den aktiven Anweisungen im Kindknoten entfernt. Anschließend wird die Liste der Expression der SequenceBlockExpression den aktiven Anweisungen hinzugefügt. Die Anwendung der Regel wird in Abbildung 4.8 dargestellt. Die SequenceBlockExpression $[x; y; z]$ wird entfernt und die darin enthaltenen Expression x, y und z werden der Reihenfolge nach den aktiven Anweisungen hinzugefügt.

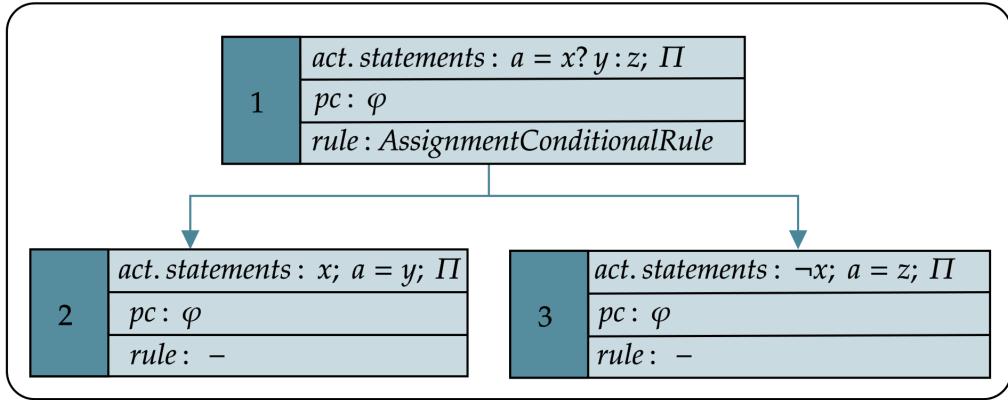


Abbildung 4.7: Anwendung der `AssignmentConditionalRule` mit `ConditionalExpression` als Assignor. Dem „true“-Kind 2 wurde die Bedingung x und `TrueCaseExpression` als Zuweisung $a = y$ den aktiven Anweisungen hinzugefügt. Dem „false“-Kind 3 die negierte Bedingung $\neg x$ und `FalseCaseExpression` als Zuweisung $a = z$.

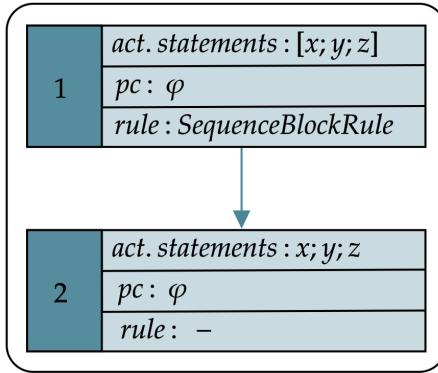


Abbildung 4.8: Anwendungsbeispiel der `SequenceBlockRule`. Die `SequenceBlockExpression` $[x; y; z]$ wird entfernt und die darin enthaltenen Expressions x, y und z werden der Reihenfolge nach den aktiven Anweisungen hinzugefügt.

PrimitiveValueRule

Die `PrimitiveValueRule` ist anwendbar, wenn an oberster Stelle der aktiven Anweisungen eine `PrimitiveValueExpression` liegt. Die Expression wird im Kindknoten gelöscht. Außerdem wird der Pfadbedingung keine Expression hinzugefügt, da eine `PrimitiveValueExpression` keinen Einfluss auf die Erfüllbarkeit des Knotens hat, egal ob diese einen booleschen Wert oder einen numerischen Wert enthält. Eine Ausnahme bildet die Expression mit booleschem Wert `false`. Die resultierende Pfadbedingung $\varphi \wedge \text{false}$ würde immer zu unerfüllbar ausgewertet werden. Daher wird in diesem Fall kein Kindknoten erzeugt und die Erfüllbarkeit des Knotens auf unerfüllbar gesetzt. In Abbildung 4.9 sind zwei beispielhafte Anwendungen der Regel zu erkennen.

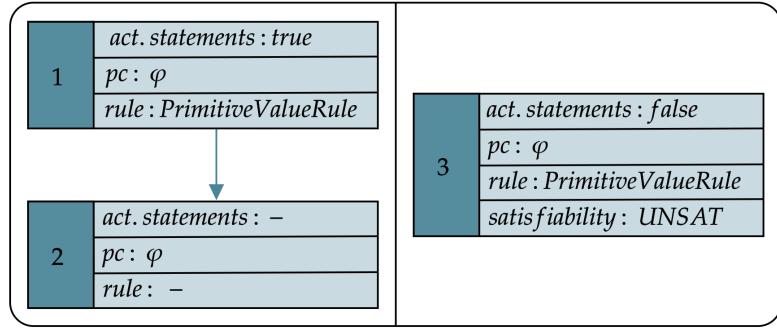


Abbildung 4.9: Anwendungsbeispiel der PrimitiveValueRule. Von Knoten 1 auf 2 wurde die Pfadbedingung nicht verändert. Bei Knoten 3 wird die PrimitiveValueExpression `false` erkannt und der Knoten wird als unerfüllbar gewertet.

4.2.3 Regeln für die Verarbeitung aktiver Anweisungen zu Pfadbedingungen

Diese Art von Regeln verarbeitet die oberste aktive Anweisung und fügt diese anschließend den zu den Pfadbedingungen des generierten Kindknotens hinzu. Im Falle uninitialisierter Variablen werden zusätzliche Expressions generiert um den Variablen default Werte zuzuweisen. Die hinzugefügten Expressions werden vorher in die *static single assignment* (SSA)-Form transformiert.

SSA Transformation

Viele moderne Compiler benutzen heutzutage eine zwischenzeitliche Repräsentation des Programmcodes in SSA-Form. Diese Form setzt voraus, dass jede Variable genau einmal zugewiesen wird und danach nicht mehr verändert wird. Jede Variable kann aber beliebig oft gelesen werden. Die kompakte Darstellung von use-def chains durch diese Form erlaubt eine effizientere Programmanalyse und fördert zudem einfacheres implementieren, testen und debuggen. Beispielsweise ermöglicht es bessere Ergebnisse bei vielen Optimierungen, wie *constant propagation*, *dead code elimination* oder *strength reduction* (vgl. [5]).

Diese Eigenschaften sind für die Erfüllbarkeitsbestimmung sehr hilfreich, da so der Verlauf von Variablenzuweisungen verfolgbar ist. An nachfolgendem Beispiel ist dies leicht erkennbar:

$$\text{Original: } a = b + 2; a > 0; a = a + c; a < 0$$

$$\text{SSA-Form: } a_0 = b_0 + 2; a_0 > 0; a_1 = a_0; a_1 < 0$$

$$\text{Ersetzung ohne SSA-Form: } a = b + 2; b + 2 > 0; a = b + 2 + c; b + 2 + c < 0$$

Die Original Expressions können nicht ohne weiteres durch den SMT-Solver gelöst werden, da beispielsweise $a = a + c$ als unerfüllbar gewertet werden würde. Die SSA-Form löst dieses Problem, indem für jede Zuweisung neue Variablen generiert werden. Weiter-

hin gibt es auch die Möglichkeit der Ersetzung ohne SSA-Form, bei der jede zugewiesene Variable immer durch ihren Assignor ersetzt werden. Dies führt zu langen und unübersichtlichen Pfadbedingungen, welche Optimierungen erschweren. Streng genommen müssten die Variablen b und c auch durch ihre Zuweisung ersetzt werden. Zusätzlich würden dadurch Aktionen entstehen, welche nicht atomar sind. Insgesamt ist diese Variante möglich, fördert aber kompliziertere Pfadbedingungen, welche weniger leicht optimierbar sind.

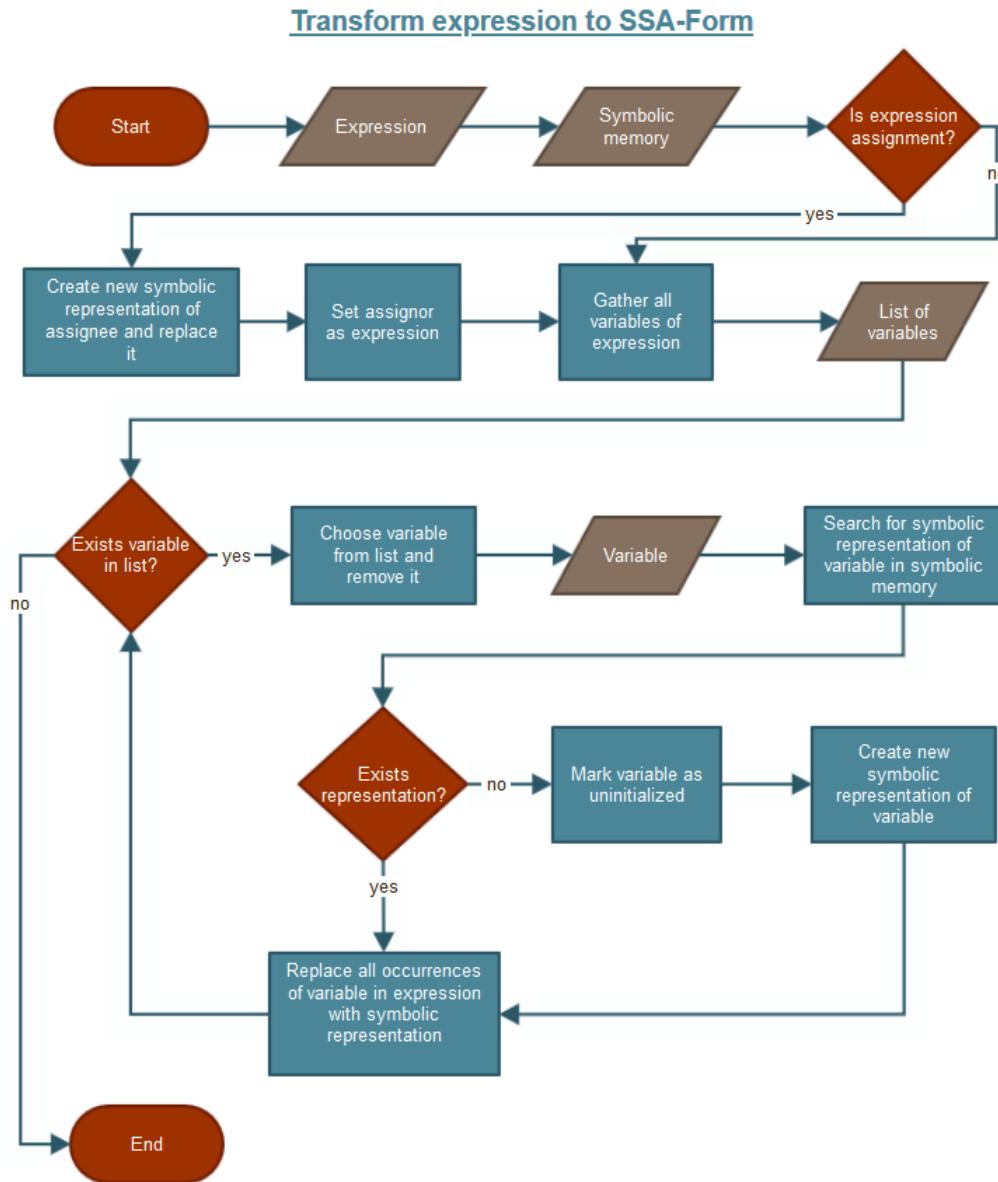


Abbildung 4.10: Ablauf der SSA-Transformation als Flowchart

Der Ablauf der Transformation einer Expression in SSA Form ist in Abbildung 4.10 dargestellt. Wie erkennbar ist, werden **AssignmentExpressions** besonders behandelt. Für den Assignee, die linke Seite der Zuweisung, wird eine neue symbolische Repräsentation

generiert und durch diese ersetzt. Danach wird der Assignor als weiter zu transformierende Expression gesetzt.

Für die Expression werden zuerst alle vorkommenden Variablen gesammelt. Für jede dieser Variable wird geschaut, ob eine symbolische Repräsentation im symbolischen Speicher existiert. Falls ja, wird die Variable durch die aktuellste ersetzt. Andernfalls wird eine neue symbolische Repräsentation für diese erstellt und alle Vorkommen der Variable in der Expression durch diese ersetzt. Gleichzeitig wird die Variable als uninitialisiert markiert. Dies ermöglicht eine spätere Erkennung und Zuweisung des default Wertes, sowie eine Analyse der uninitialisierten Variablen im Statechart (siehe 8.2.1).

Da immer die aktuellste symbolische Repräsentation gewählt wird, und bei einem neuen assignment direkt eine neue Repräsentation erzeugt wird, transformiert der Algorithmus die Expression in die SSA-Form. Abschließend wird der Ablauf an einem Beispiel verdeutlicht. Angenommen wir hätten folgende Situation:

Expression:	$b = b * c$
Symbolischer Speicher:	$b \mapsto b_0; b \mapsto b_1$

Nach der Transformation mit zusätzlichem default-Wert für c :

Expression:	$b_2 = b_1 * c_0$
Neue sym. Repr.:	$b \mapsto b_2; c \mapsto c_0$
Default-Wert Zuweisung:	$c_0 = 0$

Nachfolgend werden die Regeln vorgestellt, welche diese Transformation benutzen.

AssignmentRule

Die **AssignmentRule** ist anwendbar, wenn die oberste Expression der aktiven Anweisungen eine **AssignmentExpression** ist und der Assignor gleichzeitig weder eine **Conditional-Expression** noch eine **NumericalMultiplyDivideExpression** ist. Die Zuweisung wird in die SSA-Form transformiert und anschließend den Pfadbedingungen hinzugefügt. Die neu generierte symbolische Repräsentation des Assignees wird dem symbolischen Speicher hinzugefügt. Bei nicht initialisierten Variablen innerhalb des Assignors werden der Pfadbedingung zusätzlich generierte Expressions hinzugefügt, die diesen Variablen default Werte zuweisen. Die zugehörigen Abbildungen werden dem symbolischen Speicher ergänzt.

Die Abbildung 4.11 beschreibt die Anwendung der Regel. Alle Variablen werden durch ihre korrespondierende symbolische Repräsentation ersetzt, welche dem symbolischen Speicher hinzugefügt wird. Da c uninitialisiert ist, erkennbar an der fehlenden Abbildung im symbolischen Speicher, wird c_0 der default Wert 0 zugewiesen.

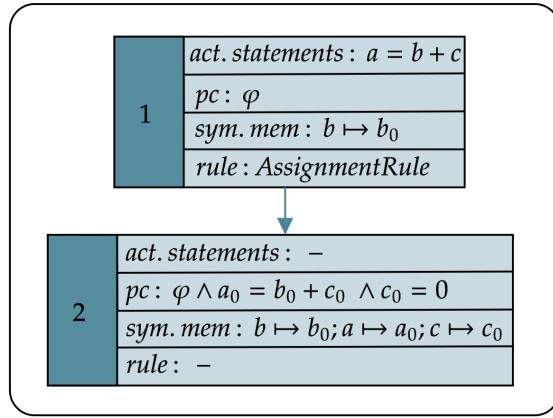


Abbildung 4.11: Beispielhafte Anwendung der `AssignmentRule`, bei der die Variable c uninitialisiert ist.

AssignmentDivisionRule

Die `AssignmentDivisionRule` ist anwendbar, wenn an erster Stelle der aktiven Anweisungen eine `AssignmentExpression` ist und gleichzeitig die rechte Seite der Zuweisung, der Assignor, eine `NumericalMultiplyDivideExpression` mit Divisionsoperator ist. Bei Anwendung der Regel entstehen zwei Kindknoten. In beiden Fällen wird die Expression in die SSA-Form transformiert und den Pfadbedingungen hinzugefügt. Zusätzlich wird der Pfadbedingung des ersten Kindes die Bedingung $divisor \neq 0$ hinzugefügt. Dem zweiten Kind wird dementsprechend die Negation hinzugefügt, also die Bedingung $divisor == 0$. Abbildung 4.12 zeigt die Anwendung der Regel. In Knoten 2 ist der Divisor ungleich 0, in Knoten 3 wird er als 0 gewertet.

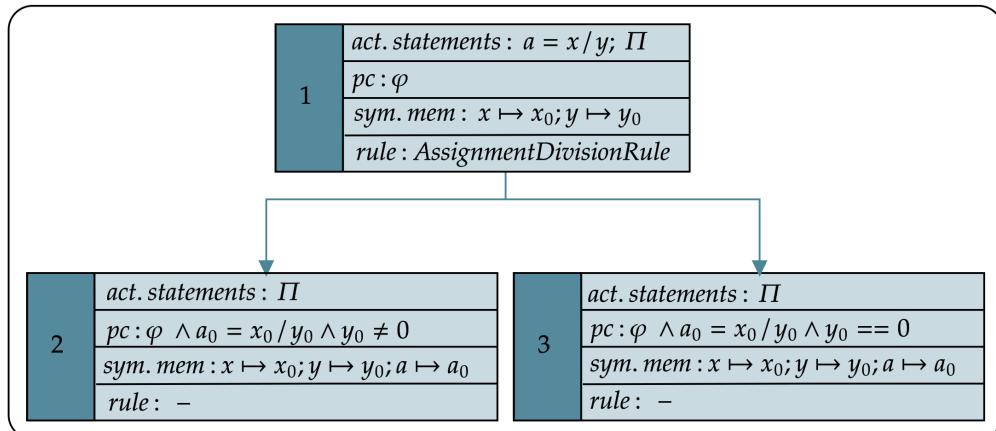


Abbildung 4.12: Anwendung der `AssignmentDivisionRule`. Knoten 2 zeigt den Fall, dass der Divisor ungleich 0 ist. In Knoten 3 wird der Divisor als 0 gewertet.

DefaultStatementRule

Die `DefaultStatementRule` deckt alle Expressions ab, die von der Engine unterstützt werden und von keiner anderer aktiven Anweisungsregel verarbeitet werden. Genauer gesagt sind dies folgende Expression: `LogicalAndExpression`, `LogicalOrExpression`, `LogicalNotExpression`, `LogicalRelationExpression`, `NumericalAddSubtractExpression`, `NumericalMultiplyDivideExpression` ohne Divisionen, `NumericalUnaryExpression` und `UnaryExpression`. Die Expression wird in die SSA Form transformiert und anschließend der Pfadbedingung hinzugefügt. Falls die Expression uninitialisierte Variablen enthält, werden diese wie bei der `AssignmentRule` behandelt. Für jede Variable wird der Pfadbedingung eine Expression hinzugefügt, die dieser den default Wert zuweist. Zusätzlich werden die neu generierten symbolischen Repräsentationen, falls diese existieren, dem symbolischen Speicher angehängt. Insgesamt funktioniert diese Regel ähnlich wie die `AssignmentRule`, außer dass keine neue symbolische Repräsentation für den Assinee generiert wird.

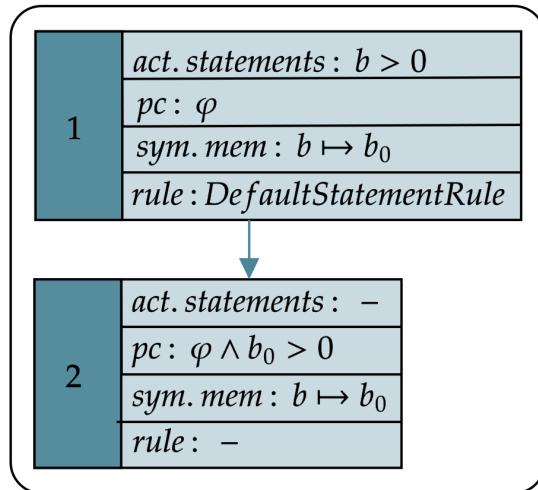


Abbildung 4.13: Beispielhafte Anwendung der `DefaultStatementRule`, bei der eine `LogicalRelationExpression` verarbeitet wird.

4.2.4 Regeln für die Zyklenelemination

In diese Kategorie fallen die beiden Regeln `TransitionCycleEliminationRule` und `LocalReactionCycleEliminationRule`. Diese sind dafür verantwortlich unendliche Ausführungs-pfade im SET zu unterbrechen, indem bestimmte Reaktionen in dem symbolischen Aus-führungspfad blockiert werden. Um trotzdem die Korrektheit der Analyse zu garantie-ren, wird gezielt Wissen über Variablen vergessen, die in blockierten Reaktionen veränderbar sind. Dadurch kann es möglich sein, dass bestimmte Zustände und Transitionen des Statecharts *false negativ* als erreichbar markiert werden, obwohl diese eigentlich unerreichbar sind. Diese Einschränkung wird für die Endlichkeit des SET in Kauf genom-

men, da eine Analyse sonst unmöglich wäre. Die `TransitionCycleEliminationRule` verhindert dabei, dass unendlich oft die gleiche Transition durchgeführt wird, während die `LocalReactionCycleEliminationRule` verhindert, dass unendlich oft lokale Reaktionen innerhalb eines Zustandes durchgeführt werden. Beide Regeln sind im Kontext der Erreichbarkeitsanalyse entwickelt worden und sind nicht allgemeingültig für andere Analysen einsetzbar. Eine detaillierte Beschreibung der Zyklenelimination ist im nächsten Kapitel 5 zu finden.

Kapitel 5

Pfadexplosion durch Zyklen

Dieses Kapitel beschreibt die Pfadexplosion durch Zyklen, welche eine der größten Herausforderungen symbolischer Programmausführung ist. Zuerst wird die Behandlung von Zyklen in der Fachliteratur besprochen und eine Begründung dafür, warum in dieser Arbeit ein anderer Ansatz gewählt wurde. Anschließend werden die Zykleneliminationen erläutert, die durch die Regeln `TransitionCycleEliminationRule` und `LocalReactionCycleEliminationRule` (4.2.4) umgesetzt werden. Die `TransitionCycleEliminationRule` dämmt unendliche symbolische Ausführungspfade ein, welche durch Zyklen bei der Durchführung von Transitionen hervorgerufen werden. Die `LocalReactionCycleEliminationRule` adressiert unendliche symbolische Ausführungspfade, die durch wiederholte Durchführung von lokalen Reaktionen eines Zustandes auftreten.

5.1 Pfadexplosion durch Zyklen in der Fachliteratur

Pfadexplosion ist bei traditioneller symbolischer Ausführung auf Programmcode meistens auf Zyklen und Funktionsaufrufe zurückzuführen [21]. Daher adressieren viele vielversprechende Ansätze der Fachliteratur diese Hauptprobleme. Da gerade die Behandlung von Zyklen für diese Arbeit interessant ist, wurden einige Ansätze näher betrachtet. Schließlich wurde jedoch der Ansatz der Zyklenelimination weiterentwickelt, der im ursprünglichen Algorithmus ([11]) vorgestellt wurde. Dies liegt zum einen daran, dass Ansätze der Fachliteratur zwar relativ gut mit simplen Zyklen zureckkommen [21], aber Schwierigkeiten bei der Analyse von komplexeren haben. Simple Zyklen sind solche, die keine weiteren Abzweigungen innerhalb ihres Körpers haben. Dagegen besitzen komplexe Zyklen weitere Abzweigungen, wie beispielsweise Unterzyklen oder if-Blöcke.

Zum anderen wurde in dieser Arbeit der Umfang für die Integration der Verfahren in die Statechart Umgebung betrachtet. Daher wurde entschieden, dass die Verfahren zwar interessant sind und Potential haben, aber bei der Entwicklung des Prototypen keine hohe

Priorität besitzen. Vielmehr war es wichtig eine funktionierende und zuverlässige Zyklusbearbeitung zu entwickeln, als die effizienteste, die vielleicht am Ende durch die begrenzte Entwicklungszeit nicht funktioniert. Dennoch bietet sich hier ein Ansatzpunkt, an dem die Engine verbessert werden kann.

Ein vielversprechender Ansatz, der grob betrachtet wurde, ist die Invariantenberechnung. Dabei werden Bedingungen ermittelt, welche vor und nach der Ausführung jeder Zykliteration erfüllt sein müssen. Generell besteht hier die Schwierigkeit eine Invariante zu bestimmen, die nützlich für die weitere Berechnung ist. Gerade bei komplexeren Zyklen sind diese Varianten häufig zu schwach. Dadurch können sie die Auswirkungen der Zyklen nicht ausreichend beschreiben und die symbolische Ausführung hat keinen Mehrwert durch die Invariante gewonnen (vgl. [25]). Hier könnte sich im Vorfeld die Komplexität von Zyklen bestimmen lassen. Damit ließen sich zumindest für die simpleren Zyklen aussagekräftige Varianten berechnen.

Verfahren der Fachliteratur besitzen zudem ein weiteres Problem, welches im Kontext von Statecharts bewältigt werden muss. Die meisten Verfahren gehen davon aus, dass Zyklen einen Einstiegs- und einen Ausstiegspunkt besitzen und basieren ihre Analyse darauf. Beispielsweise adressiert Proteus [25] das Problem komplexerer Zyklen, diese müssen aber fest definierte Ein- und Ausgänge besitzen. Wie in Abbildung 5.1 zu erkennen ist, sind Statecharts leicht konstruierbar, welche diese Bedingung nicht besitzen. Ein Ansatzpunkt für eine Verbesserung wäre eine Transformation der Zyklen, sodass diese nur einen Ein- bzw. Ausgang besitzen.

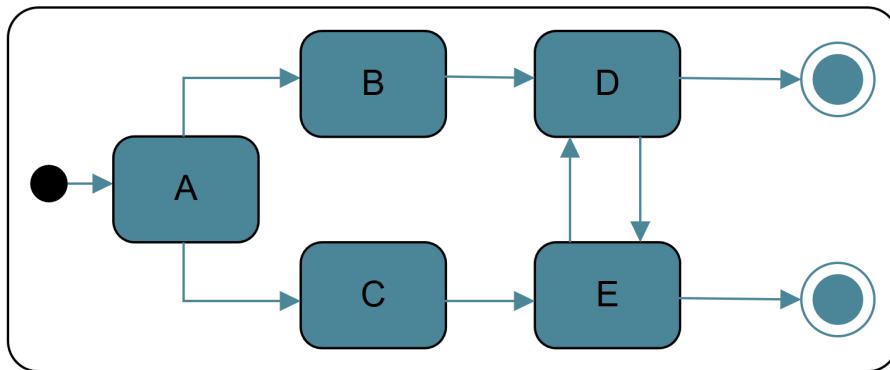


Abbildung 5.1: Statechart mit mehreren Ein- und Ausgängen des Zyklus $D \rightarrow E \rightarrow D$.

5.2 Zyklelenimation

Im folgenden wird die in der Arbeit umgesetzte Zyklelenimationen besprochen. Diese wurden speziell für die Erreichbarkeitsanalyse entwickelt und treffen daher einige Annahmen über Variablenbelegungen, die nicht ohne weiteres für andere Analysen sinnvoll sind. Bei der symbolischen Analyse und insbesondere der Konstruktion des SET ist es aber

problemlos möglich die Zykleneliminationen ein- und auszuschalten, indem die jeweiligen Regeln ausgeschaltet werden.

Durch die Eliminationen werden die durch den Zyklus veränderbaren Variablen auf neue symbolische Repräsentationen gesetzt. Dadurch besitzen diese keine Bedingungen zu vorherigen Expressions, wodurch praktisch das Wissen über diese Variable gelöscht wird. Es kann vorkommen, dass in der späteren Analyse Zustände und Transitionen als erreichbar markiert werden, obwohl sie es nicht sind. Diese Einschränkung wird in dieser Arbeit in Kauf genommen, um die Pfadexplosion einzudämmen und einen endlichen SET zu konstruieren.

5.2.1 Elimination von Transitionszyklen

Dieser Abschnitt erläutert die `TransitionCycleEliminationRule`, welche für die Elimination von Transitionszyklen verantwortlich ist. Dazu wird beschrieben, wann die Regel anwendbar ist. Abschließend werden die Effekte der Anwendung erläutert. Für die Durchführung dieser Elimination wird das Wissen aus den starken Zusammenhangskomponenten des Statecharts benutzt, welches nachfolgend beschrieben wird.

Starke Zusammenhangskomponenten

Um einen Zyklus zu eliminieren, müssen alle zugehörigen Zustände und Transitionen bekannt sein. Dafür werden die starken Zusammenhangskomponenten (2.2.2) des Statecharts durch den von Gabow [13] entwickelten Algorithmus berechnet. Die Bibliothek *jGraphT* [20] wurde für die Implementierung verwendet. Mit den starken Zusammenhangskomponenten lässt sich für jeden Zustand bestimmen, ob dieser zyklusinduzierend (2.2.16) ist. Zusätzlich wird für jede zyklusinduzierende starke Zusammenhangskomponente bestimmt, welche Reaktionen zum Zyklus gehören. Zum einen sind das alle lokalen Reaktionen der Zustände der Komponente und zum anderen alle Transitionen, die diese Zustände untereinander verbinden.

Aus diesen Reaktionen können weiterhin die veränderbaren Variablen des induzierten Zyklus ermittelt werden. Dazu werden alle Assignees der `AssignmentExpression` von Effekten der Reaktionen gesammelt.

Für das Statechart aus Abbildung 5.2 ergibt sich beispielsweise die starke Zusammenhangskomponente $\{A, B\}$. Die zugehörigen Reaktionen sind die Transitionen $A \rightarrow B$ und $B \rightarrow A$. Lokale Reaktionen besitzt keiner der zugehörigen Zustände. Die einzige veränderbare Variable ist a , da die Transition $A \rightarrow B$ die `AssignmentExpression` $a = a * 2$ als Effekt besitzt.

Anwendbarkeit

Die Anwendbarkeit der `TransitionCycleEliminationRule` kann je nach gewählter Strategie auf zwei verschiedene Arten erfolgen. In beiden Fällen muss die vorherige Transitionsdurchführung abgeschlossen sein, also die Liste aktiver Anweisungen leer sein.

Die erste Strategie *iterativ* argumentiert über die tatsächlich durchgeführten Transitionen des symbolischen Ausführungspfades. Hier wird überprüft, ob die zuletzt durchgeführte Transition insgesamt häufiger als ein durch die Strategie festgelegter Schwellenwert traversiert wurde.

Die zweite Strategie arbeitet präventiv und basiert die Anwendbarkeit darauf, ob der aktive Zustand des Knotens zyklusinduzierend ist. Dieses lässt sich einfach durch die Berechnung der starken Zusammenhangskomponenten bestimmen. Demnach wird hier direkt auf die Möglichkeit eines Zyklus reagiert und die Elimination angewandt.

```

1 input: node: Knoten
2
3 if node has active statements then:
4   return false
5
6 // preventive strategy
7 if applicability is based on preventive strategy then:
8   return node.activeState is cycle inducing
9
10 // iterative strategy
11 else if applicability is based on iterative strategy then:
12   $transitionToCheck ← node.transitions.last
13   $threshold ← get threshold from strategy
14   if node.transitions contains $transitionToCheck at least $threshold-times
15     then:
16       return true
17   else
18     return false

```

Listing 5.1: Anwendbarkeit der Transitionszyklenelimination

Anwendung

Das Ziel der Transitionszyklenelimination ist, dass der Zyklus von allen Nachfahren des Knotens in dem Ausführungspfad des SET nicht mehr betreten wird. Um dies zu erreichen werden alle Reaktionen, die zum Zyklus gehören, in allen Kindknoten blockiert. Zum einen sind das alle Transitionen, die zwei Zustände des Zyklus verbinden. Zum anderen sind dies alle lokalen Reaktionen dieser Zustände. Ausgenommen sind hierbei `Exit`-Reaktionen, da diese durch die Blockierung der Transitionen innerhalb des Zyklus nur durchgeführt werden, wenn der Zyklus verlassen wird.

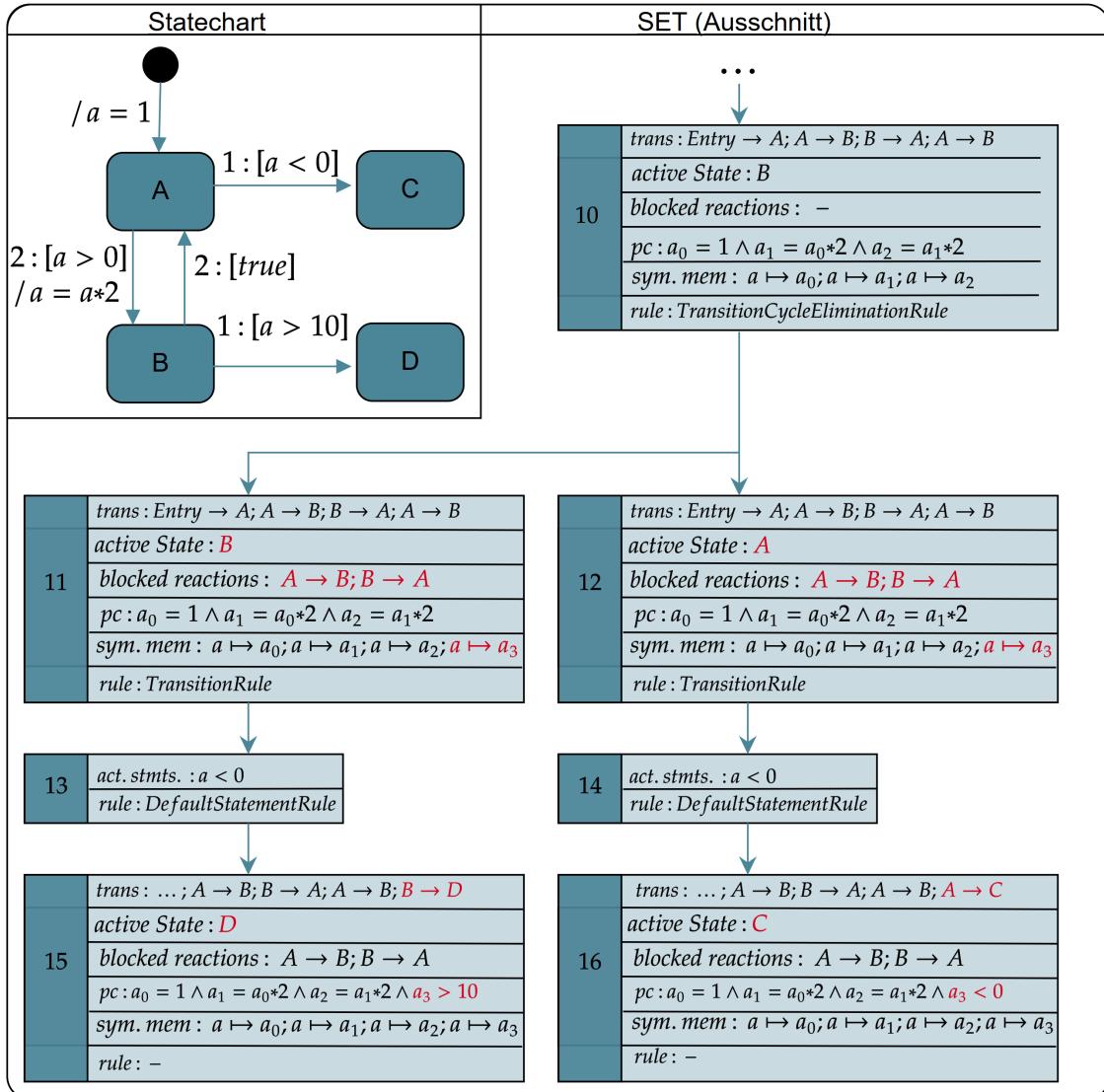


Abbildung 5.2: Beispielhafte Anwendung der `TransitionCycleEliminationRule` an einem Statechart. Knoten 10 ist nach zweimaliger Durchführung der Transition $A \rightarrow B$ entstanden. Nach Anwendung der Regel wird für die, durch die Transition veränderbare, Variable a eine neue symbolische Repräsentation erzeugt. Bei Anwendung der `TransitionRule` in Knoten 11 und 12 entsteht durch die blockierte Reaktion kein Kindknoten für den anderen Zustand des Zyklus. Knoten 15 und 16 sind durch die Anwendung der Regel beide plausibel und der SET ist nicht unendlich.

Mit der Elimination des Zyklus entspricht der symbolische Ausführungspfad keiner tatsächlichen Ausführung des Statecharts. Diese Einschränkung ist speziell für die Erreichbarkeitsanalyse 8.1 entwickelt worden. Wie zuvor erklärt, bedingt die Elimination des Zyklus, dass die dort enthaltenen Reaktionen und damit die zugehörigen Expressions nicht mehr durchgeführt werden. Um dies auszugleichen wird das Wissen über die in den Expressions veränderbaren Variablen vergessen. Genauer gesagt erhalten alle Variablen eine neue

symbolische Repräsentation im symbolischen Speicher, die Assinee einer Expression von einer blockierten Reaktion sind. Dadurch besitzen diese keine Bedingungen zu vorherigen Expressions, wodurch praktisch das Wissen über diese Variable gelöscht wird.

Zuletzt werden die Kinderknoten generiert. Die Idee ist, einen Kindknoten für jeden Ausgang aus dem Zyklus zu erstellen. Daher wird für jeden Zustand des Zyklus, welche eine Transition mit Zielzustand außerhalb des Zyklus besitzen, ein Kind erstellt. Gleichzeitig wird dieser Zustand als der aktive Zustand des Kindknoten gesetzt. Wie bereits erwähnt, werden in den Kindknoten alle Reaktionen des Zyklus bis auf `Exit`-Reaktionen blockiert.

Um sicher zu stellen, dass durch die Blockierung der Reaktionen bei der Erreichbarkeitsanalyse Transitionen nicht fälschlicherweise als unerreichbar markiert werden, wird die Analyse im späteren alle blockierten Transitionen und die Zustände des Zyklus als erreichbar markieren.

Abbildung 5.2 zeigt die Anwendung der Regel an einem Statechart. Durch das Beispiel sind die Vor- und Nachteile der Regel erkennbar. Vorteilhaft ist, dass der SET endlich ist und die Transitionen $A \rightarrow B, B \rightarrow A$ nicht unbegrenzt durchführbar sind. Zusätzlich wird die Pfadexplosion dadurch eingeschränkt, dass der Zyklus nicht weiter traversierbar ist. Nachteilig ist, dass der Knoten 16 durch die Anwendung plausibel ist, da die zu dem Zeitpunkt für a geltende symbolische Variable a_3 keine Einschränkungen besitzt.

5.2.2 Elimination von Zyklen durch lokale Reaktionen

Wie bei der `TransitionRule` erwähnt, ergeben sich Kindknoten durch die Durchführung lokaler Reaktionen eines Zustandes. Durch wiederholte Durchführung ergibt sich hieraus ein unendlicher Ausführungspfad. Um dies zu vermeiden wird dieses Verhalten durch die `LocalReactionCycleEliminationRule` eliminiert.

Anwendbarkeit

Diese Regel ist genau dann anwendbar, wenn durch die letzten zwei Anwendungen der `TransitionRule` keine Transitionen, sondern jeweils lokale Reaktionen durchgeführt wurden. Zusätzlich muss die Liste aktiver Anweisungen leer sein.

Anwendung

Durch die Anwendung der Regel entsteht genau ein Kindknoten. Diesem werden zum einen alle lokalen Reaktionen des Zustandes als blockierte Reaktionen hinzugefügt, um eine weitere Durchführung dieser zu verhindern. Falls zuvor eine Reaktion erfolgreich durchgeführt wurde, also eine Reaktionsbedingung zu `true` ausgewertet wurde, wird das Wissen über alle veränderbaren Variablen, die in den Expressions aller lokaler Reaktionen des aktiven Zustands sind, vergessen. Dies funktioniert ähnlich wie bei `TransitionCycleEliminationRule`. Genauer gesagt erhalten alle Variablen, die Assinee einer Expression von einer blockierten

Reaktionen sind eine neue symbolische Repräsentation im symbolischen Speicher. Dadurch besitzen diese keine Bedingungen zu vorherigen Expressions, wodurch praktisch das Wissen über diese Variable gelöscht wird. Falls die Variablen nicht eliminiert werden und der Ausführungspfad nochmal denselben Zustand betritt, ist es kein Problem, dass die Reaktion blockiert wurde. In diesem Fall wird die `TransitionCycleEliminationRule` anwendbar sein und so die passenden Variablen eliminieren.

Zu beachten ist, dass bei der vorherigen `TransitionCycleEliminationRule` die Variablen aller lokalen Reaktionen eliminiert werden, wodurch diese auch die `LocalReactionCycleEliminationRule` beinhaltet.

Abbildung 5.3 zeigt die Anwendung der Regel an einem Statechart. Durch das Beispiel sind die Vor- und Nachteile der Regel erkennbar. Vorteilhaft ist, dass der SET endlich ist und die Reaktion r_1 nicht unbegrenzt durchführbar ist. Nachteilig ist, dass der Knoten 14 durch die Anwendung plausibel ist, da die zu dem Zeitpunkt für a geltende symbolische Variable a_3 keine Einschränkungen besitzt.

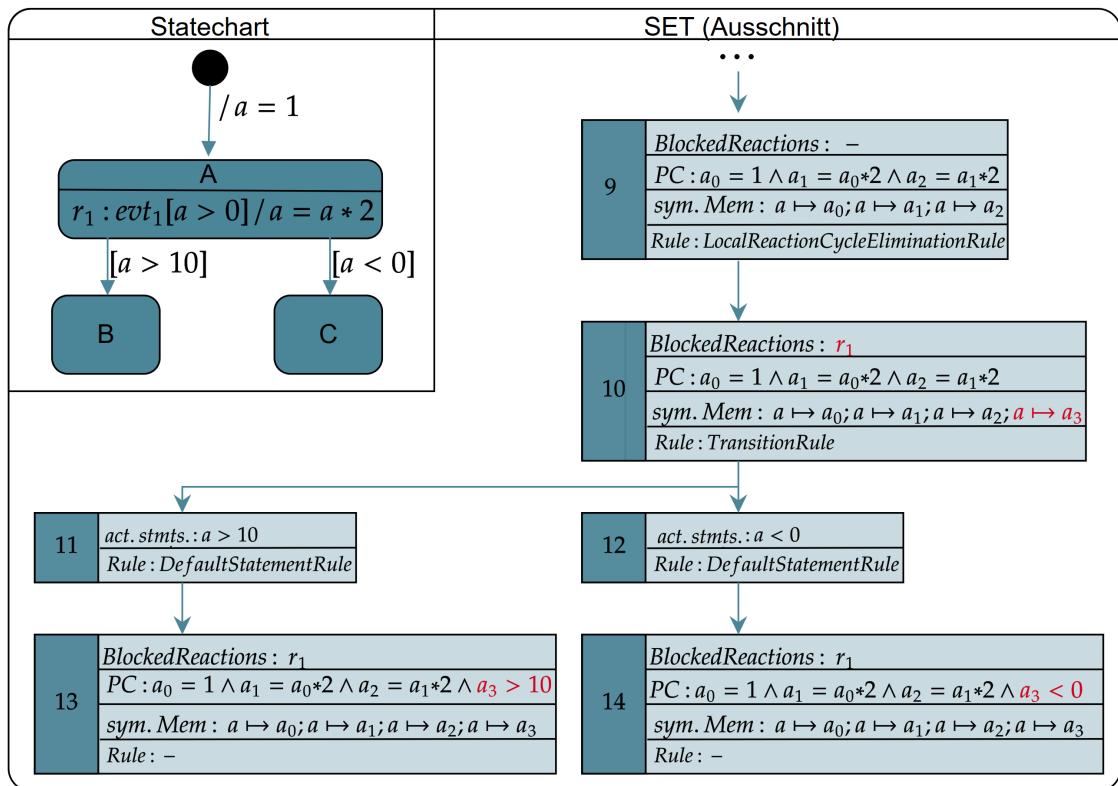


Abbildung 5.3: Beispielhafte Anwendung der `LocalReactionCycleEliminationRule` an einem Statechart. Knoten 9 ist aus der zweimaligen Durchführung der Reaktion r_1 entstanden. Nach der Anwendung der Regel wird für die in r_1 zugewiesene Variable a eine neue symbolische Repräsentation erstellt. Bei Anwendung der `TransitionRule` in Knoten 10 entsteht durch die blockierte Reaktion kein Kindknoten für r_1 . Knoten 13 und 14 sind durch die Anwendung der Regel beide plausibel und der SET ist nicht unendlich.

Kapitel 6

Erfüllbarkeitsbestimmung

Dieses Kapitel beschreibt den Ablauf der Erfüllbarkeitsbestimmung von Pfadbedingungen der Knoten und den dort eingesetzten Optimierungen. Diese dienen zum einen dazu die Größe und Komplexität der Pfadbedingung zu reduzieren. Zum anderen wird die Gesamtanzahl der Aufrufe des SMT-Solvers verringert. Dabei wird zuerst die Kostenfunktion der Erfüllbarkeitsbestimmung erläutert. Danach wird eine Erkenntnis über die Wiederverwendung von Erfüllbarkeitsergebnissen bewiesen, welche für die folgenden Optimierungen benutzt wird. Der komplette Ablauf der Bestimmung, dargestellt in Abbildung 6.1, beginnt mit dem Versuch, die Erfüllbarkeit des Knotens über simples Kontextwissen mittels Kontext-Solving 6.3 zu bestimmen. Wenn dies nicht erfolgreich ist, wird die Pfadbedingung durch die Unabhängigkeitsoptimierung 6.4 reduziert und anschließend in das international anerkannte SMT-LIB2 Format konvertiert 6.5. Je nach gewählter Strategie wird die Bedingung danach mit dem SMT-Solver Z3 6.6 oder mithilfe der jConstraints Bibliothek 6.7 evaluiert. Abschließend wird das Erfüllbarkeitsergebnis verarbeitet und im SET verbreitet 6.8.

6.1 Kostenfunktion

Dieser Abschnitt erläutert die entwickelte Kostenfunktion der Erfüllbarkeitsbestimmung. Die Idee dahinter ist die Reduzierung der Gesamtanzahl der Aufrufe des SMT-Solvers, indem Knoten nicht direkt überprüft werden. Vielmehr wird erst im späteren Verlauf des symbolischen Ausführungspfades getestet, ob der Pfad erfüllbar ist. Zum einen werden nur Knoten auf Erfüllbarkeit getestet, die keine aktiven Anweisungen besitzen. Zum anderen ist durch die Strategie festgelegt, dass eine gewisse Mindestanzahl an Abzweigungen im symbolischen Ausführungsbaum nach der letzten Erfüllbarkeitsbestimmung existieren muss. Dies lässt sich beispielsweise an Abbildung 6.5 sehen. Angenommen Knoten 1 wäre auf Erfüllbarkeit überprüft worden und die Mindestanzahl an Verzweigungen wäre 2. Dann

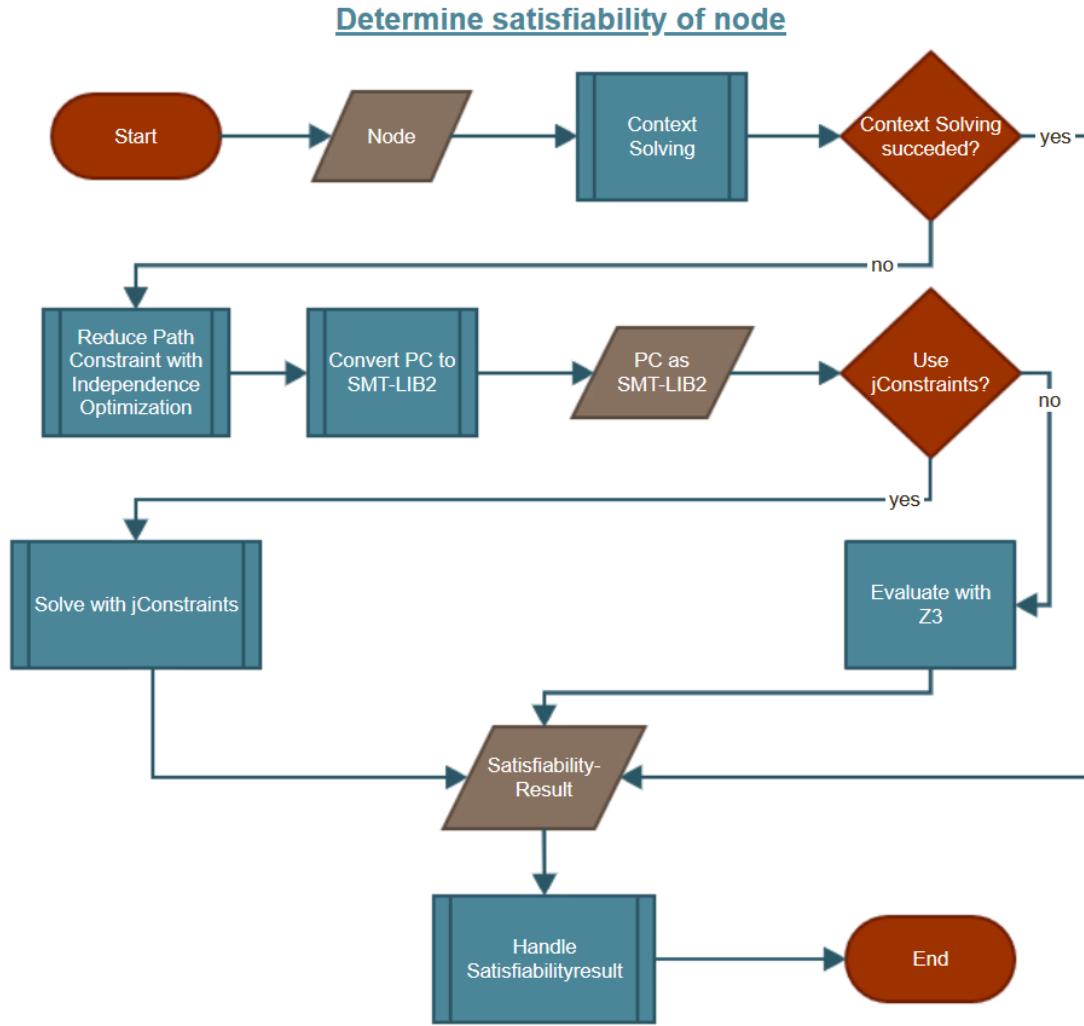


Abbildung 6.1: Ablauf der Erfüllbarkeitsbestimmung als Flowchart

würde erst in Knoten 5 eine Erfüllbarkeitsbestimmung erfolgen, da zwischen 1 und 5 zwei Verzweigungen liegen.

Um trotzdem eine korrekte Konstruktion des SET zu garantieren, werden die Erfüllbarkeitsergebnisse im SET verbreitet. Wie genau diese Weitergabe funktioniert, wird später im Abschnitt 6.8 erläutert.

6.2 Wiederverwendung von Erfüllbarkeiten

In folgendem Abschnitt wird eine Erkenntnis zur Wiederverwendung von Erfüllbarkeiten, die während der Bestimmung verwendet wird, erläutert.

6.2.1 Lemma (Endlichkeitssatz) *Der Endlichkeitssatz für die Aussagenlogik [12] (nachfolgend Endlichkeitssatz) besagt, dass eine Menge Σ aussagenlogischer Formeln genau dann erfüllbar ist, wenn jede endliche Teilmenge $\Gamma \subseteq \Sigma$ erfüllbar ist.*

6.2.2 Lemma (Wiederverwendung von Erfüllbarkeiten)

Seien n und m Knoten eines SET, π_m und π_n Pfadbedingung der jeweiligen Knoten.

Zusätzlich ist m Vorfahre von n . Dann gilt:

- a) $\pi_m \subseteq \pi_n$
- b) π_n ist erfüllbar $\implies \pi_m$ ist erfüllbar
- c) π_m ist unerfüllbar $\implies \pi_n$ ist unerfüllbar

Beweis (Wiederverwendung von Erfüllbarkeiten)

- a) Folgt aus der Konstruktion, da n Nachfahre von m ist und bei der Generierung von Kindern die Pfadbedingung erweitert wird, oder gleich bleibt. Bestehende Bedingungen der Bedingung werden bei der Kindergenerierung niemals gelöscht oder verändert.
- b) Folgt aus dem Endlichkeitssatz und a).
- c) Folgt aus dem Endlichkeitssatz und a). □

6.3 Kontext-Solving

In diesem Abschnitt wird Kontext-Solving erläutert, welches versucht, die Erfüllbarkeit eines Knotens durch simples Kontextwissen zu bestimmen. Wie in Abbildung 6.2 dargestellt, wird zuerst der jüngste Vorfahre bestimmt, dessen Erfüllbarkeit schon bekannt ist. Wenn dieser Vorfahre unerfüllbar ist, gilt die Unerfüllbarkeit durch die Wiederverwendung von Erfüllbarkeiten auch für den Knoten. Andernfalls wird die Pfadbedingung des Knotens mit der des Vorfahrens verglichen. Wenn diese identisch sind, ist der Knoten erfüllbar, da der Vorfahre erfüllbar ist. Mögliche Nachfahren werden nicht auf bekannte Erfüllbarkeit überprüft. Falls dort relevante Informationen zu diesem Knoten bekannt sind, würden diese bereits bei der jeweiligen Erfüllbarkeitsbestimmung hier in diesen Knoten eingetragen worden sein. Das wird später bei der Verbreitung von Erfüllbarkeitsergebnissen in 6.8 erläutert.

Kontext-Solving funktioniert zwar nicht in allen Fällen, kann aber trotzdem die Last des SMT-Solvers reduzieren. Wenn beispielsweise durch die letzten durchgeföhrten Transitionen keine Expressions zu den Pfadbedingungen hinzugefügt wurden, kann durch das Kontext-Solving schnell die Erfüllbarkeit bestimmt werden.

6.4 Unabhängigkeitsoptimierung

Als nächstes wird eine weitere wichtige Optimierung beschrieben, welche auf die *constraint independence optimization* aus EXE [10] aufbaut. Ziel dieser Optimierung ist die Reduk-

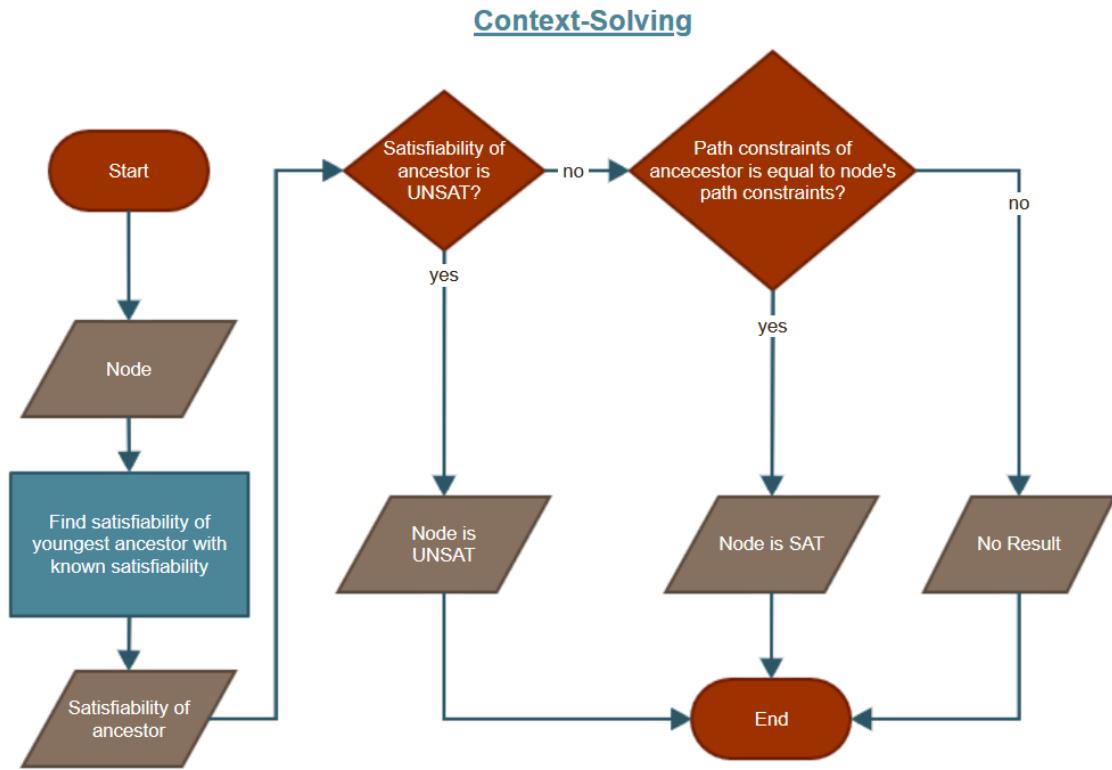


Abbildung 6.2: Ablauf des Kontext-Solvings als Flowchart

tion der Größe und Komplexität der Pfadbedingung. Um dies zu erreichen werden zwei Prinzipien ausgenutzt.

6.4.1 Grundlagen zur Unabhängigkeit

Zum einen wird die vorherige Erkenntnis über die Wiederverwendung von Erfüllbarkeiten 6.2 eingesetzt. Die zu bestimmende Pfadbedingung φ wird in die Teilmengen $\varphi = \varphi_{sat} \cup \varphi_{new}$ aufgeteilt. φ_{sat} ist dabei die größte Teilmenge der Pfadbedingung, für welche die Erfüllbarkeit bekannt ist. Diese Teilmenge lässt sich anhand des jüngsten Vorfahren mit bekannter Erfüllbarkeit ablesen. Die Teilmenge ist sicher erfüllbar, andernfalls wäre φ durch Kontext-Solving als unerfüllbar bestimmt worden. Die Teilmenge φ_{new} ergibt sich als $\varphi_{new} = \varphi \setminus \varphi_{sat}$.

Zum anderen wird die Tatsache benötigt, dass eine Pfadbedingung φ häufig in mehrere Teilmengen φ_i unabhängiger Pfadbedingungen unterteilbar ist.

Definition (Unabhängige Pfadbedingungen) Zwei Pfadbedingungen π_1 und π_2 gelten als unabhängig, falls sie disjunkte Variablenmengen besitzen. Beispielsweise lässt sich die Pfadbedingung

$$\pi : a = 0 \wedge b = 0 \wedge c = 0 \wedge c \geq a$$
in

$$\pi_1 : a = 0 \wedge c = 0 \wedge c \geq a \text{ und } \pi_2 : b = 0$$

unabhängiger Pfadbedingungen unterteilen. π_1 besitzt die Variablenmenge $\sigma_1 = \{a, c\}$, π_2 besitzt die Variablenmenge $\sigma_2 = \{b\}$ und es gilt $\sigma_1 \cap \sigma_2 = \emptyset$.

Um zu zeigen, dass φ erfüllbar ist, muss schließlich nur gezeigt werden, dass alle unabhängigen Teilmengen φ_i erfüllbar sind. Falls alle φ_i erfüllbar sind, existiert für jede Teilmenge φ_i ein erfüllbares Modell, welches jeder Variable aus φ_i einen konkreten Wert zuweist. Da alle Variablenmengen der einzelnen Teilmengen disjunkt sind, bildet die Vereinigung aller Modelle ein erfüllbares Modell für φ .

Da φ_{sat} und φ_{new} potentiell abhängige Pfadbedingungen sind, ist die Erfüllbarkeitsbestimmung von φ_{new} allein nicht ausreichend um die Erfüllbarkeit von φ zu zeigen. Aus diesem Grund werden die beiden Unterteilungen von φ nachfolgend zusammengesetzt:

6.4.1 Lemma *Sei φ eine Pfadbedingung mit einer Unterteilung in untereinander unabhängige Pfadbedingungen φ_i mit $\varphi = \bigcup_{i \in \{1, \dots, n\}} \varphi_i$. Weiterhin existiert eine Unterteilung $\varphi = \varphi_{sat} \cup \varphi_{new}$, mit der erfüllbaren Teilmenge φ_{sat} .*

Dann ist φ genau dann erfüllbar, wenn $\bigcup_{i \in K} \varphi_i$ mit $K = \{i \in \{1, \dots, n\} \mid \varphi_i \cap \varphi_{new} \neq \emptyset\}$ erfüllbar ist.

Beweis „ \implies “:

Angenommen, φ ist erfüllbar. Damit sind nach dem Endlichkeitssatz für Aussagenlogik alle Teilmengen von φ erfüllbar. Da nach Voraussetzung $\forall i \in \{1, \dots, n\} : \varphi_i \subseteq \varphi$ gilt, folgt die Implikation.

„ \Leftarrow “:

Angenommen $\bigcup_{i \in K} \varphi_i$ mit $K = \{i \in \{1, \dots, n\} \mid \varphi_i \cap \varphi_{new} \neq \emptyset\}$ ist erfüllbar. Aus dem Endlichkeitssatz der Aussagenlogik folgt, dass φ_i für $i \in K$ erfüllbar ist.

Es reicht zu zeigen, dass φ_i für $i \in (\{1, \dots, n\} \setminus K)$ erfüllbar ist. Da $\varphi = \bigcup_{i \in \{1, \dots, n\}} \varphi_i$ eine Unterteilung in unabhängige Teilmengen ist, folgt damit die Erfüllbarkeit von φ .

Sei also $\psi \in \{\varphi_i \mid (i \in \{1, \dots, n\} \setminus K)\}$. Aus der Definition von K folgt, dass $\psi \cap \varphi_{new} = \emptyset$ gilt. Aus $\varphi = \varphi_{sat} \cup \varphi_{new}$ und $\psi \subseteq \varphi$ folgt, $\psi \subseteq \varphi_{sat}$. Durch den Endlichkeitssatz folgt die Erfüllbarkeit von ψ und damit die Behauptung. \square

6.4.2 Reduktion durch Unabhängigkeit

Wie zuvor gezeigt, reicht die Erfüllbarkeit der Vereinigung aller unabhängigen Teilmengen von φ zu zeigen, die eine Teilmenge von φ_{new} sind.

Die Berechnung der unabhängigen Teilmengen wird, wie in EXE ([10]), durch einen Graphen bestimmt, dessen Knoten die Variablen von φ sind. Eine Kante existiert zwischen zwei Knoten, wenn es eine Bedingung innerhalb der Pfadbedingung gibt, die beide Variablen beinhaltet. Anschließend werden die zusammenhängenden Knoten durch einen

Graphalgorithmus bestimmt. In dieser Arbeit wurde dafür eine einfache Breitensuche verwendet, die aus jGraphT [20] übernommen wurde. Danach werden alle Bedingungen zusammengefügt, die zu der gleichen zusammenhängenden Knotenmenge korrespondieren. Abschließend werden nur die Teilmengen an Bedingungen betrachtet, welche mindestens eine Bedingung aus φ_{new} enthalten.

Die Optimierung liefert im schlimmsten Fall komplett φ , sodass die gleiche Pfadbedingung wie die vor der Optimierung überprüft wird. Im besten Fall sind φ_{new} und φ_{sat} unabhängige Pfadbedingungen, sodass nur φ_{new} auf Erfüllbarkeit geprüft werden. Beispielsweise arbeitet die Optimierung gut mit der Zyklenelimination zusammen, wie in Abbildung 5.3 erkennbar ist. Wenn die Erfüllbarkeit für Knoten 9 mit der Pfadbedingung $a_0 = 1 \wedge a_1 = a_0 * 2 \wedge a_2 = a_1 * 2$ bekannt ist, muss für die Erfüllbarkeit der Knoten 13 und 14 jeweils nur die Erfüllbarkeit von $a_3 > 10$ bzw. $a_3 < 0$ gezeigt werden, da die Variablenmengen $\{a_0, a_1, a_2\}$ und $\{a_3\}$ disjunkt sind.

6.5 SMT-LIB2 Serialisierung

In diesem Abschnitt wird die Serialisierung der Pfadbedingung in das SMT-LIB2 Format [3] beschrieben. SMT-LIB2 ist ein international anerkannter Standard, welcher als gemeinsame Ein- und Ausgabesprache vieler SMT-Solver, wie zum Beispiel Z3 [19] oder SMTInterpol [7], benutzt wird. Die Serialisierung verläuft größtenteils unkompliziert. Jede Expression der Pfadbedingung wird in eine *assertion* in SMT-LIB2 transformiert. Der größte Unterschied hierbei ist die *prefix*-Notation in SMT-LIB2. Zusätzlich muss jede verwendete Variable als null-stellige Funktion deklariert werden. In Listing 6.1 ist die Serialisierung einer beispielhaften Pfadbedingung in SMT-LIB2 dargestellt.

```

1 /* Ursprüngliche Pfadbedingung bestehend aus 2 Expressions: */
2 (x > 0 && y); (x ≠ 3 || ¬y)
3
4 /* Pfadbedingung in SMT-LIB2: */
5 // Deklaration der Variablen
6 (define-fun x () Int)
7 (define-fun y () Bool)
8 // Bedingungen als assertions
9 (assert (and (> x 0) y))
10 (assert (or (not (= x 3)) (not y)))

```

Listing 6.1: Serialisierung einer beispielhaften Pfadbedingung in SMT-LIB2

Mit diesem Vorgehen ist es möglich, fast alle unterstützten Expressions in SMT-LIB2 direkt zu übersetzen. Den einzigen Unterschied bildet die Division. Während der Entwicklung der Engine ist aufgefallen, dass die Division in SMT-LIB2 anders als in Yakindu definiert ist. Yakindu verwendet dabei die Division aus Java. Da SMT-LIB2 als Hilfe bei

der Erfüllbarkeitsbestimmung von Pfadbedingungen innerhalb Yakindu benutzt wird, muss sich die Division genauso verhalten. Um das zu gewährleisten wurden Sonderregeln für die Division mit Rest eingeführt.

6.5.1 Division mit Rest

Der Unterschied der Division liegt beim Auf- bzw. Abrunden des Ergebnisses. In Yakindu wird standardmäßig nur der Integeranteil gewertet, also zur 0 gerundet. SMT-LIB2 verhält sich anders, da dort bei einer Division x/y immer dann abgerundet wird, wenn der Nenner y positiv ist und immer dann aufgerundet wird, wenn y negativ ist.

Das führt dazu, dass das Ergebnis der Division $(-1)/3$ in Yakindu 0, aber in SMT-LIB2 -1 ist. Um das Verhalten von SMT-LIB2 anzupassen wurde die Division durch eine eigene, *yakDiv* (Listing 6.2) ersetzt. Diese addiert bzw. subtrahiert je nach Vorzeichen der Eingabe eine zusätzliche 1 um die Rundung an Yakindu anzupassen.

```

1 input: numerator x: Integer, denominator y: Integer
2 output: result of division: Integer
3
4 if (x >= 0) or (x % y == 0) then
5   return x / y
6 else if y > 0 then
7   return (x / y) + 1
8 else
9   return (x / y) - 1

```

Listing 6.2: Algorithmus der *yakDiv* - Division

Beweis Um zu beweisen, dass die Division damit äquivalent zu der von Yakindu ist, betrachten wir nachfolgend alle mögliche Fälle:

1. Fall: $\mathbf{y} = \mathbf{0}$

Yakindu liefert einen Fehler und würde die Pfadbedingung als unerfüllbar auswerten. SMT-LIB2 liefert durch die bereits erwähnte Nullteilungsprüfung auch unerfüllbar zurück.

2. Fall: $\mathbf{y} \neq \mathbf{0} \wedge \mathbf{x} \equiv \mathbf{0} \pmod{\mathbf{y}}$

In diesem Fall muss das Ergebnis nicht gerundet werden und beide Divisionen sind somit identisch.

3. Fall: $\mathbf{x} = \mathbf{0} \wedge \mathbf{y} \neq \mathbf{0}$

Beide Divisionen liefern 0 als Ergebnis.

4. Fall: $\mathbf{x} > \mathbf{0} \wedge \mathbf{y} > \mathbf{0} \wedge \mathbf{x} \not\equiv \mathbf{0} \pmod{\mathbf{y}}$

Da $(x/y) > 0$ gilt, wird das Ergebnis in Yakindu abgerundet. *yakDiv* führt die SMT-LIB2 Division durch und da $y > 0$ gilt, wird das Ergebnis auch abgerundet.

5. Fall: $x > 0 \wedge y < 0 \wedge x \not\equiv 0 \pmod{y}$

Da $(x/y) < 0$ gilt, wird das Ergebnis in Yakindu aufgerundet. yakDiv führt die SMT-LIB2 Division durch und da $y < 0$ gilt, wird das Ergebnis auch aufgerundet.

6. Fall: $x < 0 \wedge y > 0 \wedge x \not\equiv 0 \pmod{y}$

Da $(x/y) < 0$ gilt, wird das Ergebnis in Yakindu aufgerundet. Laut Definition liefert yakDiv $x/y + 1$ zurück. Da $y > 0$ gilt, wird x/y in SMT-LIB2 abgerundet und es ergibt sich $\lfloor x/y \rfloor + 1 = \lceil x/y \rceil$, da $x \not\equiv 0 \pmod{y}$ gilt.

7. Fall: $x < 0 \wedge y < 0 \wedge x \not\equiv 0 \pmod{y}$

Da $(x/y) > 0$ gilt, wird das Ergebnis in Yakindu abgerundet. Laut Definition liefert yakDiv $x/y - 1$ zurück. Da $y < 0$ gilt, wird das x/y in SMT-LIB2 aufgerundet und es ergibt sich $\lceil x/y \rceil - 1 = \lfloor x/y \rfloor$, da $x \not\equiv 0 \pmod{y}$ gilt.

Abschließend ist in Listing 6.3 ein Beispiel erkennbar, welches die Bedingung $z = x/y$ als SMT-LIB2 String dargestellt.

```

1 // Deklaration der Variablen
2 (define-fun x () Int)
3 (define-fun y () Int)
4 (define-fun z () Int)
5 // Division z = x / y
6 (assert (= z
7   (ite (or (>= x 0) (= 0 (mod x y))) (div x y) (ite (> y 0) (+ (div x y) 1)
8     (- (div x y) 1))))
9 ))
```

Listing 6.3: Beispiel einer Division in SMT-LIB2

6.6 Solving mit Z3

Je nach gewählter Strategie wird die Erfüllbarkeit der SMT-LIB2 Bedingung direkt mit Z3 bestimmt. Dazu wird ein Prozess gestartet, der die Bedingung über die Konsole an Z3 schickt und aus welchem das Ergebnis ablesbar ist. Wichtig anzumerken ist, dass für die gesamte Erfüllbarkeitsbestimmung immer der gleiche Prozess benutzt wird. Ein wiederholtes beenden und wiederaufbauen des Prozesses dauert zu lange. Z3 bietet dafür die Möglichkeit, durch die Befehle `push` und `pop`, Bedingungen hinzuzufügen und zu löschen, sodass der gleiche Prozess verwendet werden kann.

Momentan wird immer die komplette Pfadbedingung entfernt. Eine Verbesserungsmöglichkeit wäre, nur Teile der Pfadbedingung zu löschen. Dies ließe sich beispielsweise gut mit der DFS Konstruktionsstrategie (7.3) verbinden, indem sich jeweils immer die Generation des Knotens der Pfadbedingung gemerkt wird. Anhand dieser könnten die zu löschen Teile der Pfadbedingung bestimmt werden.

Durch die Umwandlung der Pfadbedingung in SMT-LIB2 und die modulare Bauweise der Engine ist es zudem problemlos möglich einen anderen SMT-Solver, wie SMTInterpol, zu verwenden.

6.7 Solving mit jConstraints

Dieser Abschnitt beschreibt die verwendete Bibliothek *jConstraints* [16]. Diese ist eine Abstraktionsebene für Constraint-Solver in Java und bietet einen einheitlichen Zugriff auf verschiedene SMT-Solver wie Z3 oder SMTInterpol [7] an. Weiterhin werden nützliche Algorithmen und Werkzeuge für die Arbeit mit logischen Ausdrücken bereitgestellt, welche durch eine interne Objektstruktur abgebildet werden. Ein Beispiel für einen Algorithmus ist die Konstantenfaltung, welche für viele Nutzer von SMT-Solvern nützlich ist, aber häufig neu implementiert wird.

Durch die dort implementierten Optimierungen und einfache Anbindung an einen SMT-Solver wurde die Bibliothek für die Erfüllbarkeitsbestimmung benutzt. Weiterhin hat diese Arbeit dazu beigetragen einen Fehler in der *jConstraints* Optimierung **Numeric-Simplification** zu finden und zu beheben. *jConstraints* ist zusätzlich interessant, da die Bibliothek aktiv weiterentwickelt wird und die dort angebotenen Optimierungen in der Anzahl und Effizienz somit verbessert werden. In der Arbeit wird *jConstraints* in Verbindung mit *jConstraints-z3* [1] benutzt, welche die Schnittstelle zwischen *jConstraints* und Z3 bereitstellt.

Nachfolgend wird der Ablauf der Erfüllbarkeitsbestimmung aus Abbildung 6.3 erläutert. Zuerst wird die Pfadbedingung in das *jConstraints* eigene Format umgewandelt und es entsteht ein **SMTProblem** Objekt. Dann werden alle Bedingungen des Problems zu einer großen Konjunktion zusammengefasst, da *jConstraints* diese Darstellung bevorzugt. Anschließend wird die Pfadbedingung durch die **NumericSimplification** optimiert. Diese arbeitet ähnlich wie die *Implied-Value-Concretization* aus KLEE [9], indem alle Variablen, für die ein Wert in Abhängigkeit zu einer anderen Variablen ohne konkreten Wert existiert, durch diesen Wert ersetzt werden. Dadurch ist es möglich die Komplexität der Pfadbedingung zu reduzieren und einige Bedingung komplett zu entfernen. Als nächstes wird geschaut, ob in dem **SMTProblem** noch freie Variablen existieren, also solche, die keinen konkreten Wert besitzen. Ist dies nicht der Fall, kann die Evaluation ohne SMT-Solver direkt in *jConstraints* gelöst werden. Andernfalls wird das optimierte Problem an Z3 weitergegeben und auf Erfüllbarkeit getestet. Der Test auf freie Variablen erlaubt daher die Anzahl der Probleme zu reduzieren, die der SMT-Solver löst. Dies ist besonders interessant, da die untersuchten Pfadbedingungen häufig konkrete Variablenwerte besitzen.

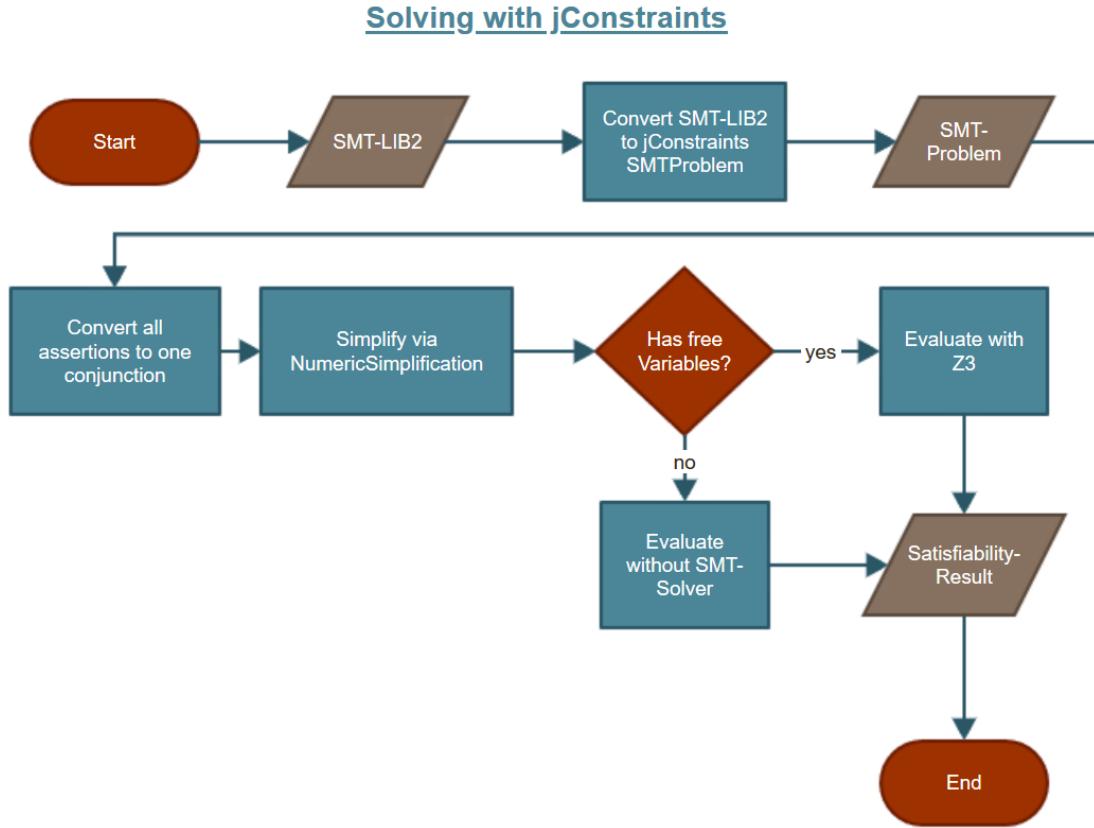


Abbildung 6.3: Ablauf der Erfüllbarkeitsbestimmung mittels jConstraints als Flowchart

6.8 Verbreitung des Erfüllbarkeitsergebnisses

Der letzte Schritt der Erfüllbarkeitsbestimmung ist die Verarbeitung des Ergebnisses. Wie in 6.1 erläutert, wird nicht jeder Knoten direkt auf Erfüllbarkeit überprüft. Daher muss das Ergebnis an diese Knoten weitergegeben werden.

6.8.1 Ältesten-Suche im SET

In Abbildung 6.4 ist erkennbar, wie sich das Erfüllbarkeitsergebnis anschließend im SET ausbreitet. Zuerst wird dieses in den Knoten selbst eingetragen. Im Fall, dass das Ergebnis erfüllbar lautet, werden alle Vorfahren, die keine eingetragene Erfüllbarkeit besitzen, als erfüllbar markiert. Dies ist durch die Wiederverwendung der Erfüllbarkeiten aus 6.2 möglich.

Andernfalls ist aus demselben Grund bekannt, dass alle Nachfahren auch unerfüllbar sind, wenn der Knoten es selber ist. Daher werden diese als unerfüllbar markiert. Um den Knoten zu finden, an dem die Pfadbedingung unerfüllbar wurde, wird der älteste Vorfahre des Knotens gesucht, der unerfüllbar ist. Diese Information wird für die anschließende Analyse benötigt. Wenn es Vorfahren gibt, die keine bekannte Erfüllbarkeit besitzen, wird

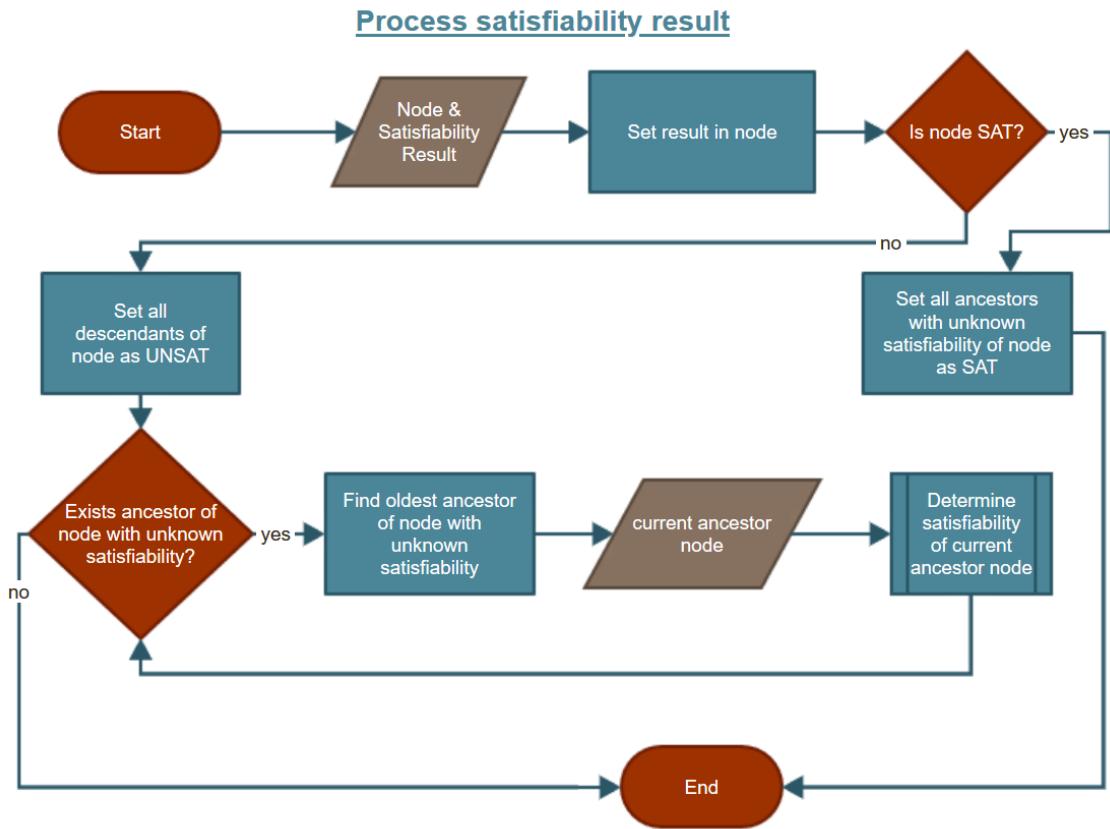


Abbildung 6.4: Ablauf der Verbreitung der Erfüllbarkeit als Flowchart

der älteste von ihnen auf Erfüllbarkeit getestet. Hierbei durchläuft der Knoten die komplette Erfüllbarkeitsbestimmung, welche in Abbildung 6.1 dargestellt ist und in diesem Kapitel erläutert wurde. Genauso wird das Ergebnis durch den SET verbreitet. Diese Suche geschieht solange, bis kein Vorfahre ohne bekannte Erfüllbarkeit mehr existiert. Da nach und nach für jeden Knoten die Erfüllbarkeiten bestimmt werden und sich Erfüllbarkeit in den Vorfahren und Unerfüllbarkeit in den Nachfahren ausbreitet, terminiert dieser Algorithmus.

In Abbildung 6.5 ist ein Beispiel der Verbreitung zu erkennen.

6.8.2 Binäre-Suche im SET

Eine weitere Strategie, die für die Verbreitung entwickelt wurde, ist die binäre Suche. Der einzige Unterschied zu der Ältesten-Suche besteht dabei, dass nicht der älteste unbekannte Knoten auf Erfüllbarkeit getestet wird. Stattdessen wird der Knoten auf Erfüllbarkeit getestet, der in der Mitte zwischen dem ältesten unbekannten und ältesten bekannten Knoten liegt, wobei zum jüngeren Knoten gerundet wird. Im Beispiel 6.5 würde der Knoten 3 als erstes auf Erfüllbarkeit überprüft werden, da die Knoten 2 und 3 die Mitte von 1 und 5 bilden und 3 jünger als 2 ist.

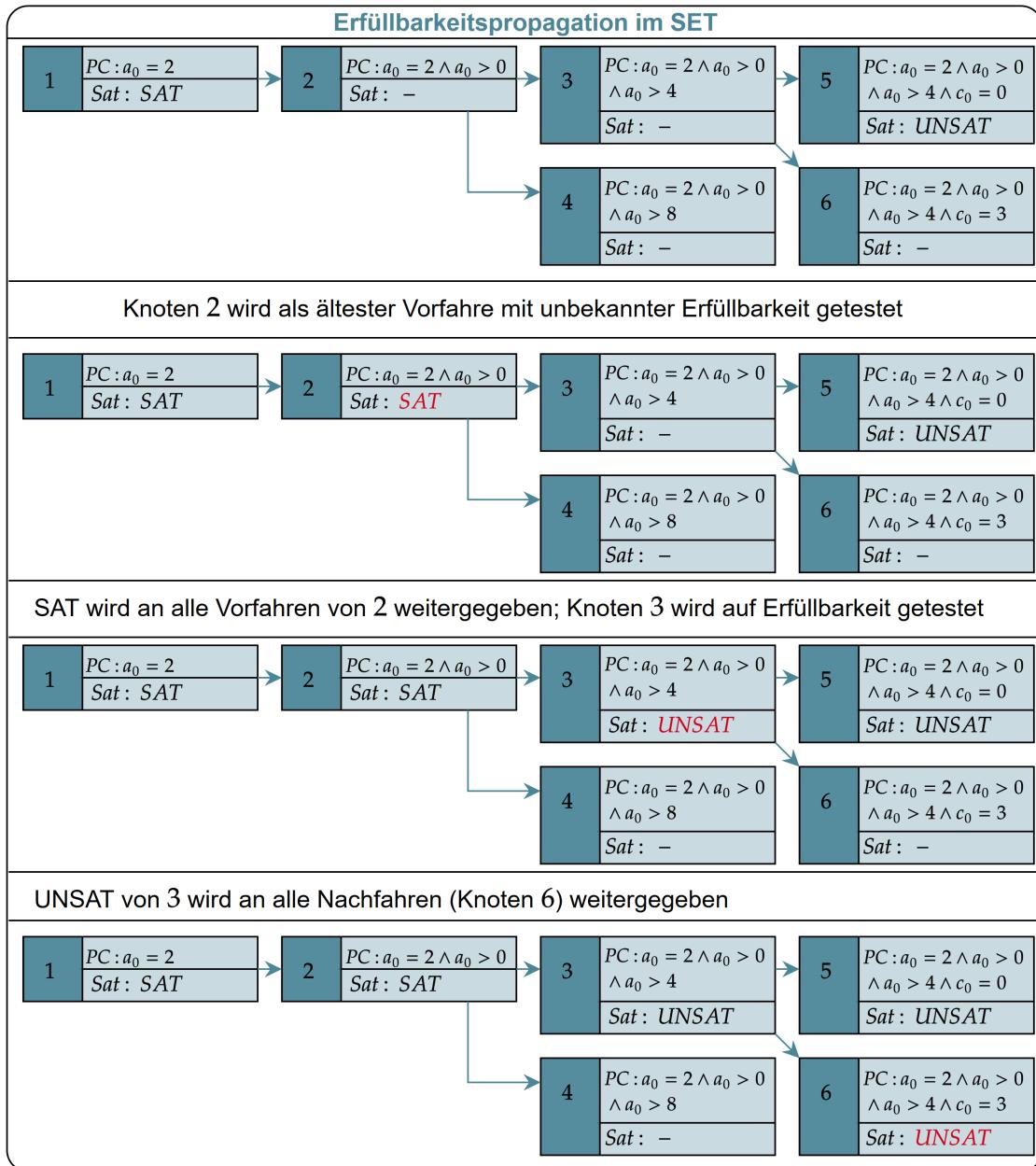


Abbildung 6.5: Beispielhafte Verbreitung eines Erfüllbarkeitsergebnisses im SET. Knoten 5 ist hierbei der Startknoten der Verbreitung. Zu beachten ist, dass die Erfüllbarkeit von Knoten 4 am Ende immer noch unbekannt ist.

Durch die Erfüllbarkeitsbestimmung und insbesondere der Verbreitung der Erfüllbarkeit ist nach der Konstruktion des SET für jeden Knoten die Erfüllbarkeit bekannt. Diese Information wird bei der späteren Erreichbarkeitsanalyse verwendet.

Kapitel 7

Konstruktion des SET

In diesem Kapitel wird der Ablauf der Konstruktion des SET beschrieben. Dazu zählen die Konstruktion der Wurzel (7.1), der allgemeine Ablauf (7.2) und die Strategie der symbolischen Ausführung (7.3). Abschließend wird die Terminierung der Ausführungspfade erläutert.

7.1 Konstruktion der Wurzel

Die Wurzel des SET wird aus der Definition Section des Statecharts konstruiert. Dazu wird eine Liste von aktiven Anweisungen aus den Variablen Deklarationen erstellt, welcher der Variable den initialen Wert aus der Definition Section zuweist. Da Variablen in YAKINDU ohne initialen Wert einen default Wert erhalten, werden diese erstmal ignoriert und nicht den aktiven Anweisungen hinzugefügt. Falls diese „uninitialisierten“ Variablen später referenziert werden, wird nachträglich der default Wert bei der SSA-Transformation (4.2.3) eingetragen. Diese Verschiebung des Problems erlaubt eine Analyse (8.2.1) von uninitialisierten Variablen ohne weiteren großen Aufwand. Da die Pfadbedingung der Wurzel leer ist, ist diese plausibel und es wird **SAT** als Erfüllbarkeit eingetragen. Der aktive Zustand der Wurzel ist der Entry-Zustand der Hauptregion des Statecharts.

Abbildung 7.1 zeigt beispielhaft, wie die Wurzel des SET aussieht.

Statechart		SET-Wurzel											
<pre>definition - section : var a : integer = 0 var b : integer var x : boolean = true event evt1</pre>		1	<table border="1"><tr><td>act. stmts : a = 0; x = true</td><td>sym. mem : -</td></tr><tr><td>satisfiability : SAT</td><td>parent : -</td></tr><tr><td>active state : Entry</td><td>children : -</td></tr><tr><td>trans. : -</td><td>pc : -</td></tr><tr><td>blocked reactions : -</td><td>rule : -</td></tr></table>	act. stmts : a = 0; x = true	sym. mem : -	satisfiability : SAT	parent : -	active state : Entry	children : -	trans. : -	pc : -	blocked reactions : -	rule : -
act. stmts : a = 0; x = true	sym. mem : -												
satisfiability : SAT	parent : -												
active state : Entry	children : -												
trans. : -	pc : -												
blocked reactions : -	rule : -												

Abbildung 7.1: Beispielhafte Wurzel des SET. Die Variablen *a* und *x* werden als aktive Anweisung hinzugefügt.

7.2 Symbolische Programmausführung

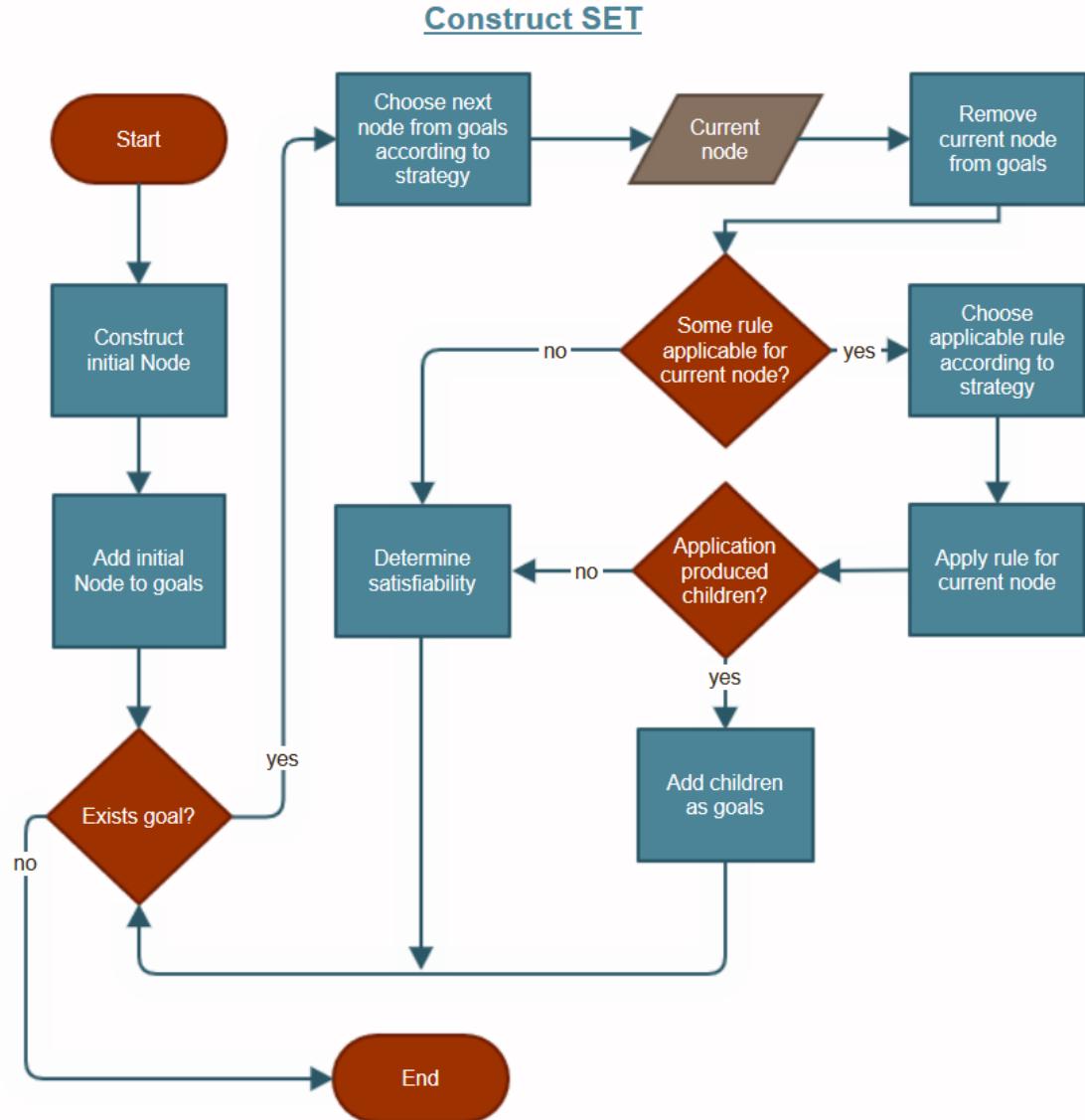


Abbildung 7.2: Ablauf der Konstruktion des SET als Flowchart

Dieser Abschnitt beschreibt den allgemeinen Ablauf der symbolischen Programmausführung bzw. der Konstruktion des SET. Dieser wird in Abbildung 7.2 dargestellt. Zuerst wird die Wurzel des SET erzeugt und den *goals*, einer Liste aller noch zu bearbeitenden Knoten, hinzugefügt. Insgesamt werden solange weitere goals verarbeitet, bis es keine mehr gibt. Während der Ausführung werden immer wieder neue Knoten erstellt, die den goals zugefügt werden.

Als nächstes wird ein goal anhand der Strategie ausgewählt und aus der Liste entfernt. Für diesen Knoten wird nun überprüft, welche Regeln anwendbar sind. Da auch mehrere Regeln anwendbar sein können, wird eine durch die Strategie gewählt und angewandt.

Wenn die Anwendung der Regel Kinderknoten produziert, werden diese den goals hinzugefügt. Im Falle, dass keine Kinder produziert werden oder dass keine Regel anwendbar ist, wurde ein Blatt im finalen SET gefunden, für welches die Erfüllbarkeitsbestimmung forciert wird. Der Fall, dass keine Regel anwendbar ist, sollte ausgeschlossen sein, wird aber sicherheitshalber trotzdem aufgeführt, falls das Statechart beispielsweise nicht unterstützt Expressions enthält. Durch die Anwendbarkeit der Regeln und im speziellen der `TransitionRule` ist gesichert, dass die letzte angewandte Regel die `TransitionRule` ist und keine weiteren Kinder produziert werden. Genauer wird dies in Abschnitt 7.4 erläutert. Abschließend, wenn keine goals mehr vorhanden sind, ist die Konstruktion des SET abgeschlossen und dieser ist bereit für die Analyse (Kapitel 8).

7.3 Strategie der symbolischen Programmausführung

Die symbolische Programmausführung kann mit verschiedenen Strategien ausgeführt werden. So ist es beispielsweise möglich verschiedene Optimierungen ein- bzw. auszuschalten. Nachfolgend werden die verschiedenen Möglichkeiten benannt.

Auswahl der anzuwendenden Regel Durch die Strategie wird entschieden, welche Regel angewandt wird, wenn mehrere möglich sind. Hier wurde sich für nur eine Reihenfolge entschieden:

```
TransitionCycleEliminationRule > LocalReactionCycleEliminationRule >
    TransitionRule > Regeln aktiver Anweisungen
```

Die `TransitionCycleEliminationRule` hat eine höhere Priorität als die `LocalReactionCycleEliminationRule`, da durch erste auch die zweitere mit angewandt wird. Daher macht es keinen Sinn die `LocalReactionCycleEliminationRule` zuerst anzuwenden, da zusätzlich danach auch die `TransitionCycleEliminationRule` angewandt werden könnte (Kapitel 5). Die `TransitionRule` hat eine niedrige Priorität als die Eliminationen, da die Eliminationen unendliche Transitionsdurchführungen verhindern sollen und sonst keine Wirkung hätten. Die Regeln zu aktiven Anweisungen haben die niedrigste Priorität, da diese sich gegenseitig ausschließen und auch mit den vorher genannten Regeln nicht gleichzeitig anwendbar sind.

Konstruktionsrichtung des SET Durch die Strategie wird entschieden, welcher Knoten als nächstes aus den goals verarbeitet wird. Hier wurden die zwei Ansätze *BFS* und *DFS* entwickelt.

- **BFS** (*breadth-first-search*) wählt den Knoten mit dem frühesten Erstellungszeitpunkt. So wird der SET Ebene für Ebene in die Breite aufgebaut.

- **DFS** (*depth-first-search*) wählt den Knoten mit dem spätesten Erstellungszeitpunkt. Mit dieser Strategie wird jeder Ausführungspfad im SET nacheinander konstruiert.

Momentan bringt keine der Konstruktionsrichtungen einen Performanzvorteil. Mit weiteren Optimierungen, wie zum Beispiel verbesserter `push` und `pop` Benutzung von Z3, wäre hier aber ein Vorteil mit DFS zu erreichen.

Transitionszyklenelimination Bei der Transitionszyklenelimination `TransitionCycleEliminationRule` (5) wurden verschiedene Ansätze entwickelt, wann die Regel anwendbar ist. Zum einen wurde ein präventiver Ansatz entwickelt, der die Elimination bei der Möglichkeit eines Zyklus startet. Die iterativen Ansätze starten die Elimination nach wiederholter Durchführung der selben Transition. Die Anzahl der Durchführungen ist durch die Strategie einstellbar. Die Elimination kann zudem auch ausgeschaltet werden. Hier ist aber nicht für jedes Statechart garantiert, dass die Analyse ohne Abbruch terminiert.

Kostenfunktion Durch die Strategie wird die Anzahl der Abzweigungen nach der letzten Bestimmung im symbolischen Ausführungspfad festgelegt, die vor der Erfüllbarkeitsbestimmung existieren müssen (6.1).

Kontext-Solving Durch die Strategie kann die Kontext-Solving Optimierung ein- bzw. ausgeschaltet werden (6.3).

Unabhängigkeitsoptimierung Durch die Strategie kann die Unabhängigkeitsoptimierung ein- bzw. ausgeschaltet werden (6.4).

SMT-Solver Durch die Strategie kann die Wahl des SMT-Solvers eingestellt werden. Es besteht die Möglichkeit zwischen nativem Z3 (6.6) und jConstraints (6.7).

Ergebnisverbreitung Durch die Strategie kann die Reihenfolge der Ergebnisverbreitung (6.8) festgelegt werden. Es besteht die Wahl zwischen oldest-search und binary-search.

Zeitlimit Durch die Strategie kann ein Zeitlimit der symbolischen Programmausführung eingeführt werden. Nach Ablauf der Zeit bricht die Ausführung ab. Dieses kann im Falle ausgeschalteter Transitionszyklenelimination nützlich sein.

7.4 Terminierung von Pfaden

Abschließend wird die Terminierung der symbolischen Ausführungspfade beschrieben. Aktive Anweisungen werden durch die korrespondierenden Regeln verarbeitet. Diese Regeln sind so aufgebaut, dass die Anweisungen solange verarbeitet werden, bis die Liste leer ist.

Bei Knoten ohne aktive Anweisungen gibt es zwei Fälle. Der aktive Zustand besitzt entweder ausgehende Transitionen, oder lokale Reaktionen, oder keins von beidem. Falls er keine besitzt, werden bei der Anwendung der `TransitionRule` keine Kindknoten generiert, was eine Terminierung des Ausführungspfades zur Folge hat. Andernfalls werden in den Kindknoten entweder lokale Reaktionen oder eine Transition durchgeführt. Da Statecharts nur endliche viele Elemente besitzen, werden diese früher oder später wiederholt durchgeführt. Die unendliche Durchführung der jeweiligen Reaktionen wird aber durch die Zykluselimination verhindert. Weiterhin bewirkt die Blockierung der jeweiligen Reaktionen, dass beim Durchführungsversuch durch die `TransitionRule` keine Kindknoten entstehen. Insgesamt werden Ausführungspfade daher früher oder später mit einer abschließenden `TransitionRule` terminieren.

Durch die Erfüllbarkeitsbestimmung sicher gestellt, dass nur Knoten auf Plausibilität überprüft werden, für welche die Erfüllbarkeit der Pfadbedingung unbekannt ist. Weiterhin wird bei jedem Durchlauf der Bestimmung mindestens eine unbekannte Erfüllbarkeit bestimmt und eingetragen. Dadurch wird die Plausibilität schlussendlich für jeden Knoten bekannt sein und terminieren.

Zusätzlich kann durch die Strategie ein Zeitlimit eingestellt werden, welches einen Abbruch der Ausführung zur Folge hat.

Kapitel 8

Analyse

Dieses Kapitel erläutert die Erreichbarkeitsanalyse für Zustände und Transitionen, die auf dem vorher konstruierten SET durchgeführt wird. Zusätzlich werden weitere Analysen aufgezählt, die auf dem SET durchführbar sind.

8.1 Erreichbarkeitsanalyse

Die Erreichbarkeitsanalyse bestimmt die Erreichbarkeit aller Zustände und Transitionen des Statecharts. Dafür wird der zuvor konstruierte SET analysiert. Wie in 5 erläutert wurde, wird der SET mithilfe der Zyklenelemination konstruiert. Diese ist notwendig um einen endlichen und analysierbaren SET zu erhalten. Gleichzeitig wird durch die Elimination Wissen über Variablen vergessen, wie in Abbildung 5.2 erkennbar ist. Dadurch wird der Zustand C fälschlicherweise als erreichbar markiert. Diese Einschränkung wird bewusst in Kauf genommen sodass es insgesamt möglich ist, dass *false negative* Fehler in der Erreichbarkeitsanalyse auftreten. Hingegen werden *false positive* Fehler nicht geduldet. Ein unerreichbar markierter Pfad ist auf jeden Fall unerreichbar.

Abbildung 8.1 zeigt den Ablauf der Erreichbarkeitsanalyse auf dem SET. Alle Knoten werden jeweils einmal durchlaufen. Insgesamt werden aber nur die Knoten näher betrachtet, deren angewandte Regel die `TransitionRule` oder eine der Zykleneliminationen ist. Diese enthalten alle notwendigen Informationen über die Erreichbarkeit.

Bei einem erfüllbaren Knoten mit `TransitionRule` ist bekannt, dass keine aktiven Anweisungen mehr existieren. Das heißt, dass die letzte Transition komplett abgeschlossen ist und ihre Effekte und ihr Guard in der Pfadbedingung verarbeitet wurden. Daher ist der komplette Transitionspfad inklusive der letzten Transition valide bzw. durchführbar. Genauso ist der aktive Zustand des Knotens erreichbar. Daher werden die letzte Transition und der aktive Zustand des Knotens als erreichbar markiert. Es reicht in diesem Fall nur die letzte Transition zu markieren, da alle zuvor traversierten Transitionen durch einen weiteren Knoten mit `TransitionsRule` als erreichbar markiert werden.

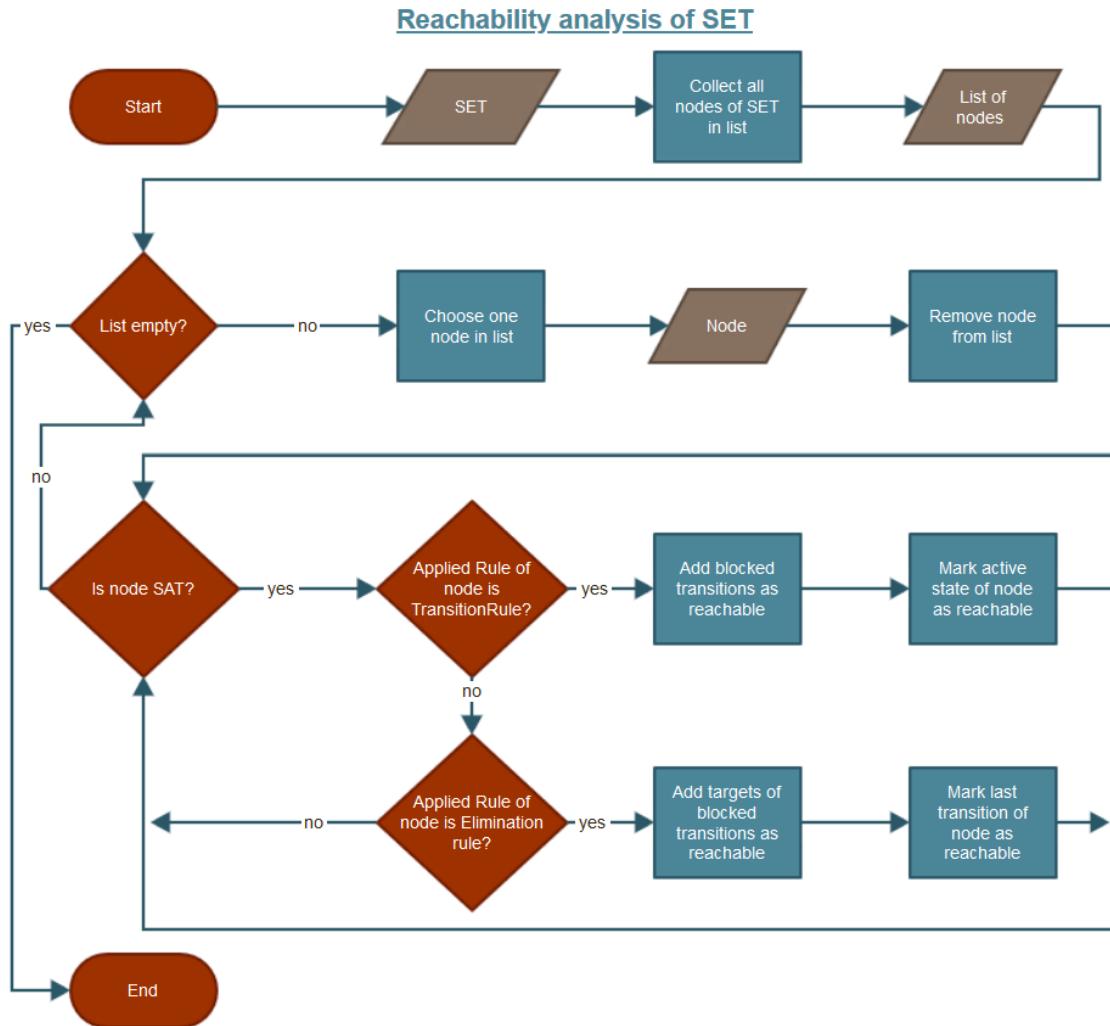


Abbildung 8.1: Ablauf der Erreichbarkeitsanalyse als Flowchart

Weiterhin müssen Knoten mit `TransitionCycleEliminationRule` oder `LocalReactionCycleEliminationRule` betrachtet werden. Bei der Elimination wird angenommen, dass alle Zustände und Transitionen innerhalb des Zyklus erreichbar sind. Da diese aber nicht zwingend traversiert werden, ist es möglich, dass diese niemals aktiver Zustand oder letzte Transition eines Knotens mit `TransitionRule` sind. Daher werden alle blockierten Transitionen und deren Zielzustände eines erfüllbaren Knotens mit Eliminationsregel als erreichbar markiert. Da nur Transitionen innerhalb der Zustände des Zyklus blockiert werden, sind keine zusätzlichen *false positive* Fehler außerhalb des Zyklus durch diesen Schritt möglich.

Insgesamt ist anzumerken, dass durch die Analyse auch Zustände berücksichtigt werden, die keine ausgehenden Transitionen besitzen, da die letzte angewandte Regel jedes Ausführungspfades die `TransitionRule` ist.

8.2 Weitere Analyse

In diesem Abschnitt werden weitere Möglichkeiten der SET Analyse beschrieben. Diese existieren nicht in der Implementierung, geben aber einen Eindruck darüber, welche Analysen mit welchem Mehraufwand realisierbar sind.

8.2.1 Uninitialisierte Variablen

Eine Möglichkeit ist die Analyse von uninitialisierten Variablen. Diese werden in Yakindu standardmäßig ignoriert und bei Referenzierung mit einem default Wert initialisiert. Trotzdem kann es sinnvoll sein, dem Benutzer darüber eine Meldung zu geben, oder für andere Domänen außerhalb von Yakindu interessant sein.

Die Analyse ist hierbei recht simpel. Dazu werden die Regeln betrachtet, welche aktive Anweisungen zu Pfadbedingungen verarbeiten (4.2.3). Bei der Anwendung dieser Regeln wird die jeweilige verarbeitete Expression in die SSA-Form transformiert und die referenzierten Variablen durch symbolische Repräsentation aus dem symbolischen Speicher ersetzt. Falls keine passende symbolische Repräsentation im Speicher existiert, ist die Variable uninitialisiert. In diesem Zuge werden der Pfadbedingung zusätzliche Expressions hinzugefügt, die der uninitialisierten Variable einen default Wert zuweisen. Der Assignee ist hierbei ausgenommen. Zusätzlich wird dem symbolischen Speicher eine Abbildung der Variable zu einer neuen symbolischen Repräsentation hinzugefügt.

Eine Analyse würde den SET durchgehen und jeden Kindknoten einer solchen Regel betrachten. Falls der Kindknoten einen neuen Eintrag im symbolischen Speicher besitzt, ist diese Variable vorher uninitialisiert gewesen. Ausnahme bildet der Kindknoten eines AssignmentRuleknotens. Bei diesem muss geschaut werden, ob zusätzlich zum assignee eine neue Variablenabbildung dem symbolischen Speicher hinzugefügt wurde.

Beispielhaft ist dies in Abbildung 4.11 erkennbar. Bei dem Kindknoten 2 werden zwei Einträge in den symbolischen Speicher hinzugefügt. Der Eintrag $a \mapsto a_0$ gehört zu dem Assignee der `AssignmentExpression`. Der Eintrag $c \mapsto c_0$ ist zusätzlich hinzugefügt worden und signalisiert die uninitialisierte Variable c .

8.2.2 Generierung von Testfällen

Eine weitere interessante Analysemöglichkeit ist die Testfallgenerierung, welches die benötigte Zeit für die Testphase des Statecharts reduzieren kann. Der SET ermöglicht die Betrachtung aller möglichen Ausführungspfade des Statecharts. Zusätzlich kann für alle erfüllbaren Pfade ein Modell der Variablenbelegung durch den SMT-Solver berechnet werden. Dadurch lassen sich leicht Eingaben für Unit-Tests generieren, die alle möglichen Ausführungspfade durchlaufen, wodurch eine gute *code-coverage* erreicht wird. Durch die Möglichkeit, die maximale Anzahl an Zyklenerationen in der Strategie der SET Konstruktion

festzulegen, lässt sich eine *loop-k coverage* einführen. Je nach gewünschtem Analyseverhalten ist es möglich die Zyklenelimination dahingehend zu verändern, dass Ausführungspfade nach k -Zykleniterationen abgebrochen werden und das Wissen über Variablen nicht vergessen wird. Damit würden die durch die Zyklenelimination erzeugten *false negative* Fehler umgangen, *false positive* Fehler wären aber möglich.

In der Fachliteratur existieren für diesen Anwendungsfall einige Publikationen, die dieses Thema behandeln, wie zum Beispiel [23] und [2]. Häufig wird die Testfallgenerierung in Verbindung mit dynamischer symbolischer Ausführung durchgeführt, bei der eine konkrete Ausführung parallel zur symbolischen Ausführung durchlaufen wird. Nähere Informationen zum State of the Art von dynamischer symbolischen Ausführung und Testfallgenerierung finden sich in [6]. Ein Tool, welches diese Generierung umsetzt, ist beispielsweise DART [14].

8.2.3 Weitere Analysemöglichkeiten

Auf dem SET sind noch viele weitere Analysen möglich. Beispielsweise ließen sich nicht gelesene Variablen einfach durch die Betrachtung des symbolischen Speichers der Blätter analysieren. Statecharts, die eine Division durch 0 nicht ausschließen, lassen sich durch die Betrachtung der **DivisionRule**- Knoten und dessen Kinder analysieren.

Einige Analysen lassen sich direkt auf dem vorhandenen SET durchführen, während andere Änderungen der Strategie oder eine Erweiterung des Regelwerks erfordern. Wie hier dargestellt wurde, bietet der SET und die symbolische Ausführung insgesamt weitreichende Möglichkeiten Statecharts auf verschiedenste Arten zu analysieren.

Kapitel 9

Evaluation

Dieses Kapitel beschreibt die abschließende Evaluation der entwickelten Engine. Zuerst wird dazu die Korrektheit der Ergebnisse der Erreichbarkeitsanalyse besprochen. Anschließend wird die Performanz verschiedener Strategien evaluiert. Danach wird die entwickelte Engine mit dem ursprünglichen Algorithmus aus [11] verglichen. Abschließend wird ein Fazit dieser Evaluation gegeben.

9.1 Korrektheit

Über den Zeitraum der Entwicklung wurde die Erreichbarkeitsanalyse, mit zugehöriger SET Konstruktion, ausgiebig integrativ getestet. Hierzu wurden Statecharts konstruiert, die bestimmte Fehlerklassen und Grenzfälle widerspiegeln und die einzelnen Komponenten der Engine testen. Dabei wurde kontrolliert, dass für diese Statecharts die Erreichbarkeit korrekt analysiert wurde. Insgesamt hat sich die Analyse als stabil und zuverlässig erwiesen und es wurden keine *false positive* Fälle gefunden. Die Code-Coverage der Tests betrug insgesamt 85%.

Beispielhaft wird hier ein Statechart vorgestellt, an welchem die Korrektheit von einigen Komponenten, wie der default Wert Initialisierung, Transitionszyklenelimination, Elimination lokaler Reaktionen, Priorisierung von Transitionen und die Division getestet wurde. Dieses ist in Abbildung 9.1 zu sehen.

Zum einen wurde das Statechart mittels der default Strategie analysiert. Hier wurden korrekterweise Zustand F , sowie Transitionen $A \rightarrow F, D \rightarrow F, (G \rightarrow F)_1$ und $(G \rightarrow F)_2$ als unerreichbar markiert. Ohne Transitionszyklenelimination wurde zudem erkannt, dass Zustand N und Transition $C \rightarrow N$ unerreichbar sind. Nachfolgend wird eine kleine Erklärung gegeben, warum die Elemente unerreichbar sind:

- **A → F:** b ist uninitialisiert und wird daher der default Wert 0 zugewiesen.
- **D → F:** $D \rightarrow E$ hat eine höhere Priorität und wird im Zustand D durchgeführt, da der Guard im Ausführungszustand D immer `true` ist.

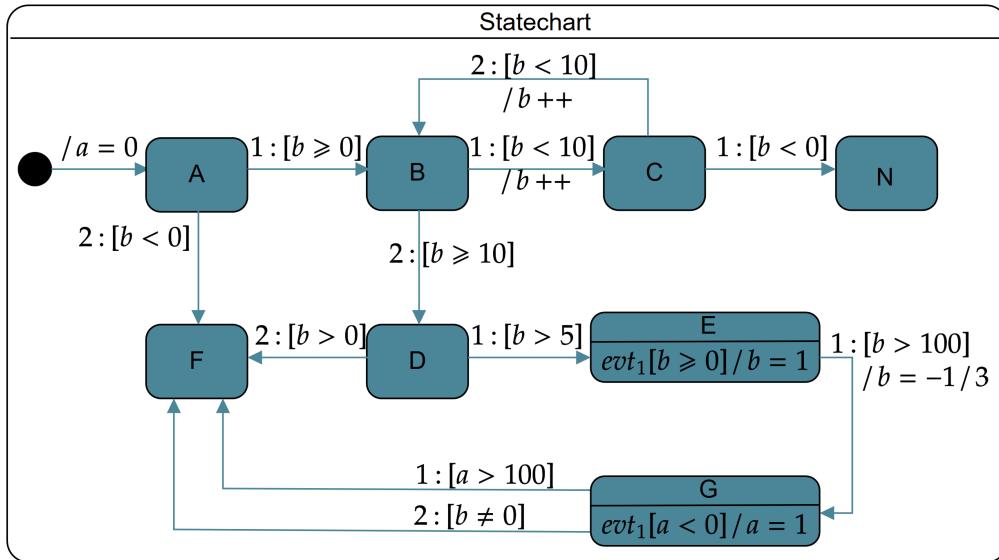


Abbildung 9.1: Beispielhaftes Statechart an dem die Korrektheit evaluiert wurde.

- (**G → F**)₁: Die lokale Reaktion in **G** wird niemals durchgeführt, daher wird die lokale Reaktion nicht eliminiert. Dass Wissen über die Variable **a** wird nicht vergessen. Zu Anfang des Statecharts wurde $a = 0$ zugewiesen.
- (**G → F**)₂: Die Division in **E → G** wird als Java-Division evaluiert, daher ist danach $b == 0$ gegeben.
- **F**: Alle eingehenden Transitionen von **F** sind nicht erreichbar.
- **C → D**: Ohne Transitionszyklenelimination wird das Wissen über **b** nicht vergessen. Mit Elimination schon, daher ist in diesem Fall die Transition erreichbar.
- **N**: Ohne Transitionszyklenelimination ist die eingehende Transition nicht erreichbar, mit Elimination schon.

9.2 Performanz

In diesem Abschnitt wird die Performanz der Engine bezüglich einzelner Strategien an einem skalierbaren Statechart evaluiert. Da diese Evaluation an einem synthetischen Statechart stattfindet, ist dies kein ausschöpfendes Performance-Review. Vielmehr soll ein Überblick über die verschiedenen Strategien gegeben werden. Die Strategien, bei denen beispielsweise eine Optimierung ausgeschaltet wurde, werden dazu mit einer default Strategie verglichen. Um die Performanz zu messen wurden pro Strategien 4 Durchläufe der Engine gemacht. Der erste Durchlauf dient als Warmup, die Zeit wurde als Mittelwert der nachfolgenden 3 Messungen genommen. Die Engine lief auf einem Lenovo ThinkPad T440p mit IntelCore i7-4710MQ @2.5Ghz und 16GB Arbeitsspeicher. Zuerst wird das Statechart vorgestellt.

9.2.1 Skalierbares Statechart

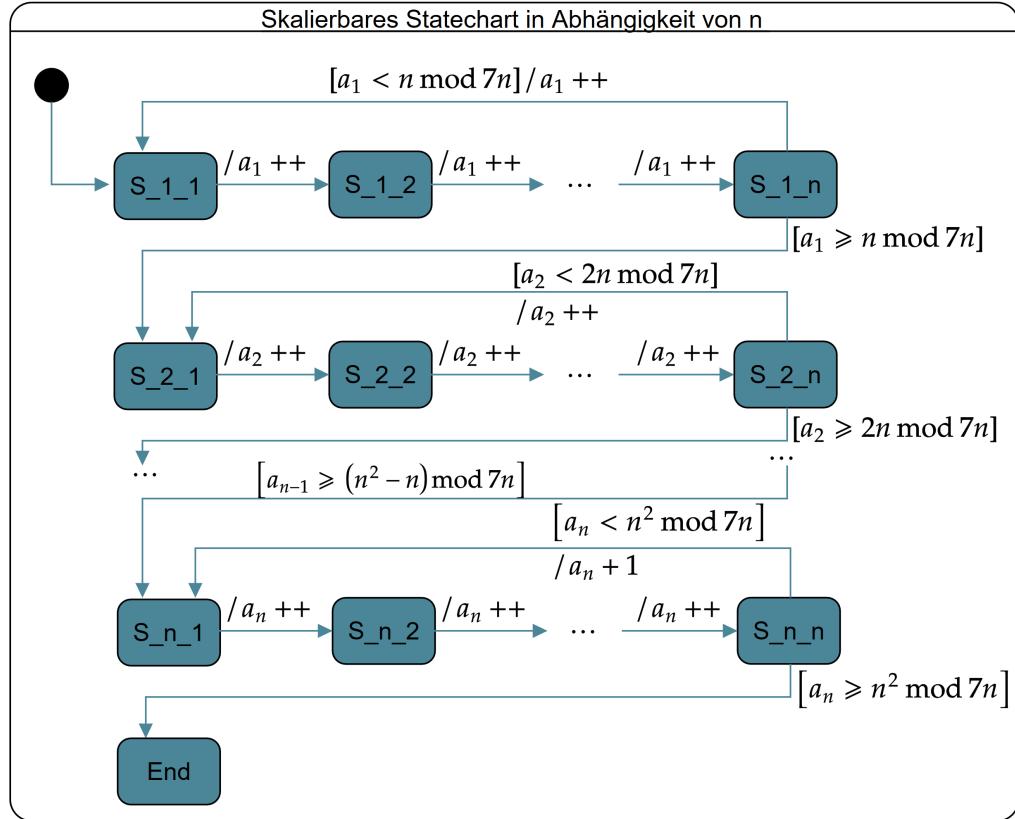


Abbildung 9.2: Skalierbares Statechart in Abhängigkeit des Parameters n .

Um die Performanz der Engine zu evaluieren wurde ein skalierbares Statechart entworfen. Der Aufbau des Statecharts ist in Abbildung 9.2 zu erkennen. Das Statechart wird in Abhängigkeit zum Parameter n konstruiert und besitzt n Zyklen mit jeweils n Knoten. Zusammen mit dem Entry und Endzustand besitzt das Statechart $2 + n^2$ Knoten und $1 + n * (n + 1)$ Kanten. Pro Zyklus wird eine Variable generiert, die nur innerhalb des Zyklus verwendet wird. Alle Variablen werden anfangs auf 0 gesetzt. Die Knoten in einem Zyklus sind ringförmig angeordnet und es gibt immer einen Ein- und Ausgangsknoten. Jede Kante innerhalb eines Zyklus erhöht die jeweilige Variable um eins. Der Ausgangsknoten eines jeden Zyklus besitzt eine Kante zum Eingangsknoten des selben und eine Kante zum Eingangsknoten des nächstens Zyklus, wobei die Kanten jeweils mit einem Guard versehen sind. Die Guards sind so gewählt, dass der i -te Zyklus ($i \bmod 7$) + 1-mal durchlaufen wird. Dabei ist die modulo Berechnung im Guard vorab in Java geschehen und wird nicht durch die Engine gelöst. Beispielsweise würde der 1. und 8. Zyklus 2 mal durchlaufen, der 7. Zyklus nur einmal. Dies ermöglicht, dass die verschiedenen Transitionszykleneliminationen unterschiedlich viele Zyklen eliminieren. Zusätzlich ist das Statechart so gewählt, dass jeder

Zyklus automatisch verlassen wird, also die Engine auch bei ausgeschalteter Elimination terminiert.

9.2.2 Evaluierung der default Strategie

size	default strategy	
	solving time	overall time
2	0.0007	0.0055
4	0.0017	0.02
8	0.0112	0.0919
16	0.0451	0.332
32	0.2758	1.6631
64	2.056	11.2853

Tabelle 9.1: Performanz der default Strategie in Sekunden

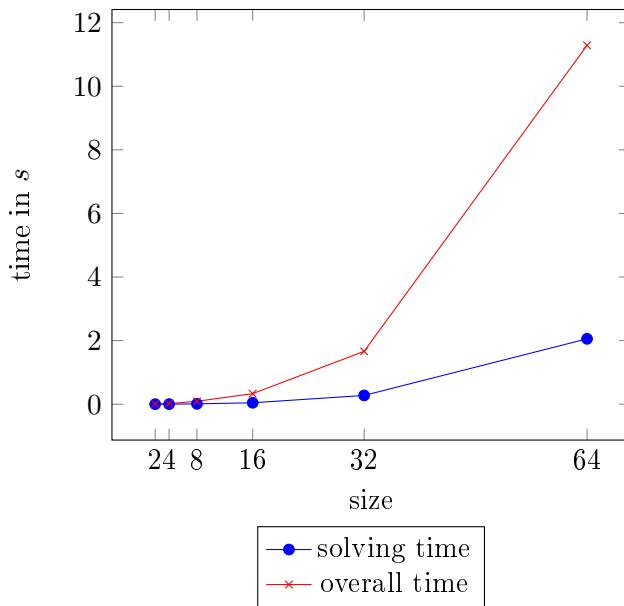


Abbildung 9.3: Graphische Visualisierung der default Strategie in Sekunden (siehe Tabelle 9.1)

Zuerst wurde das Statechart mittels einer default Strategie evaluiert. Diese lautet:
 Transitionszyklenelimination = iterativ 1, Kostenfunktion = 30, Kontext-Solving = on,
 Unabhängigkeitsoptimierung = on, SMT-Solver = natives Z3, Konstruktion = BFS,
 Zeitlimit = 1800s

Anzumerken ist, dass die default Strategie nicht die schnellste Strategie ist. Im späteren Teil dieses Kapitels wird der Grund dafür noch erläutert.

Die Performance ist in Tabelle 9.1 und Abbildung 9.3 dargestellt. Zu erkennen ist, dass die Zeit mit steigender Größe exponentiell ansteigt. Von Größe 16 auf 32 wurde die Zeit ca.

verfünffacht, von 32 auf 64 ca. versiebenfacht. Weiterhin ist der Anteil der Solving-Zeit zur Gesamt-Zeit mit steigender Größe leicht angestiegen. Bei einer Größe von 2 liegt der Anteil bei $\sim 10\%$, bei einer Größe von 64 bei $\sim 17\%$. Insgesamt arbeitet die Engine aber sehr schnell. Bei einer Größe von 32, also 1026 Knoten, dauert die Analyse unter 2s. Da Nutzer erfahrungsbedingt häufig kleinere Statecharts modellieren, ist dies mehr als ausreichend.

9.2.3 Evaluierung der Transitionszyklenelimination

size	default strategy		default w/ iterative 5 elim.	
	solving time	overall time	solving time	overall time
2	0.0007	0.0055	0.0014	0.0099
4	0.0017	0.02	0.0088	0.0925
8	0.0112	0.0919	0.0398	0.7067
16	0.0451	0.332	0.2375	3.0757
32	0.2758	1.6631	2.4239	12.4378
64	2.056	11.2853	24.8289	147.2434

size	default w/ preventive elim.		default w/o elimination	
	solving time	overall time	solving time	overall time
2	0.0005	0.0008	0.0009	0.0081
4	0.0007	0.0013	0.0047	0.077
8	0.0009	0.0022	0.0017	0.6769
16	0.0018	0.0048	0.0932	2.8673
32	0.0038	0.0129	1.7448	21.066
64	0.0072	0.0367	19.8491	190.9537

Tabelle 9.2: Vergleich der Performanz der Transitionszyklenelimination in Sekunden

In diesem Abschnitt wird die Transitionszyklenelimination evaluiert. Dazu wird das Statechart mit den verschiedenen Modi – iterativ mit einer Transitionswiederholung, iterative mit fünf Transitionswiederholungen, präventiv und ohne Zyklenelemination – der Elimination analysiert, der Rest der Strategie ist mit der default Strategie identisch. Die Performance ist in Tabelle 9.2 und Abbildung 9.5 dargestellt. Erkennbar ist, dass die präventive Strategie mit Abstand am schnellsten ist. Die default Strategie benötigt bei einer Größe von 64 insgesamt 307-mal so lange. Iterativ 5 und ohne Zyklenelemination brauchte jeweils 13-mal bzw. 17-mal so lange wie die default Strategie, welche mit iterativ 1 arbeitet. Da die Transitionszyklenelimination als einzige die Genauigkeit der Erreichbarkeitsanalyse beeinflusst, ist die schnellste Strategie nicht unbedingt die Beste. Beispielsweise werden durch die präventive Strategie im Statechart die Zyklen eliminiert, welche ein vielfaches von 7 sind. Die iterativen Strategie eliminiert diese Zyklen nicht, da diese bei normaler

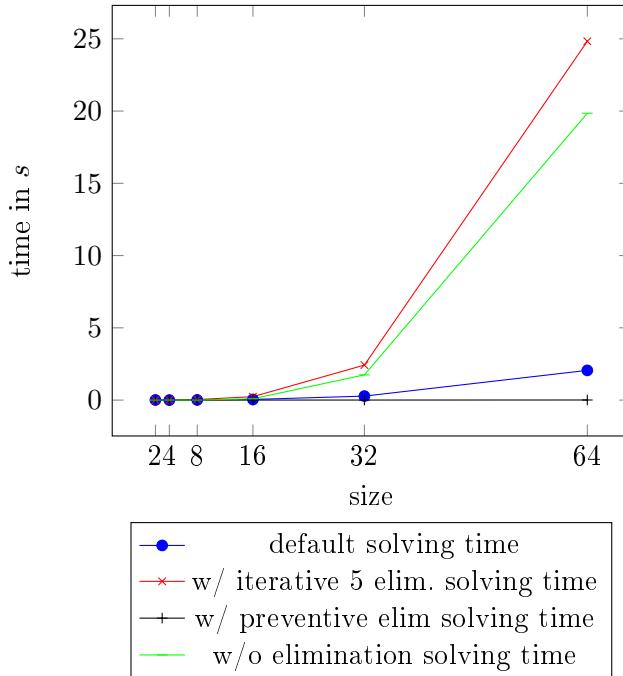


Abbildung 9.4: Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich der Transitionszyklenelimination (siehe Tabelle 9.2)

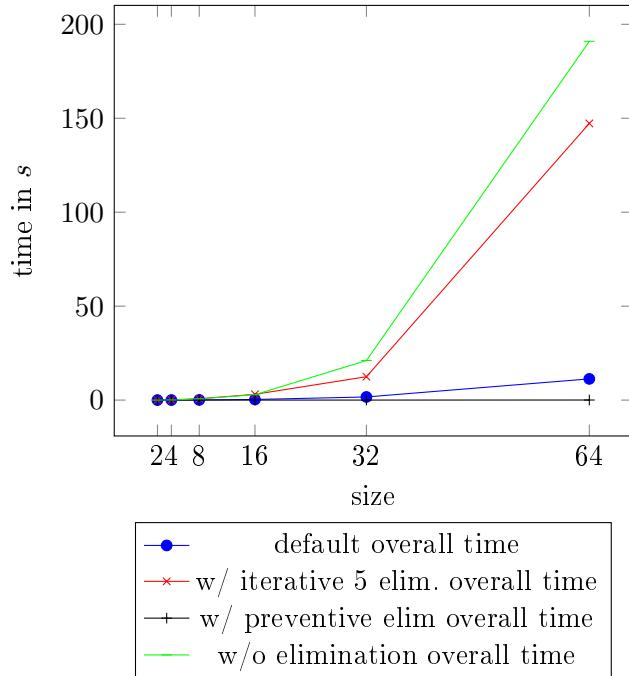


Abbildung 9.5: Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich der Transitionszyklenelimination (siehe Tabelle 9.2)

Ausführung keinen Zyklus verursachen würden. Hier ist also abzuschätzen, wie genau die Analyse sein muss und welche Zeiteinschränkungen bestehen. Es ist aber anzumerken, dass bei eliminierten Zyklen immer die gleichen Ergebnisse entstehen, egal ob iterativ 1 oder 5 gewählt wird. Der Vorteil der Genauigkeit ist also nur bei Zyklen vorhanden, die eine maximale Wiederholungsanzahl unter der gewählten Strategie besitzen. Demnach sollte eine hohe Wiederholungsanzahl nur eingestellt werden, wenn bereits Wissen über das zu analysierende Statechart bekannt ist, oder dieses klein ist. Bei unbekannten Statecharts ist durch den massiven Perfomanzgewinn wahrscheinlich die präventive Strategie am besten. Ohne Transitionszyklenelimination sollte nur analysiert werden, wenn sicher ist, dass keine unendlichen Transitionswiederholungen entstehen können. Andernfalls wird die Analyse nicht zwingend terminieren, oder bei eingestellter Zeitbeschränkung ohne Ergebnis abbrechen.

Abschließend wurde die präventive Strategie bis aufs Maximum getestet. Dieses liegt bei einer Größe von 82, also 6726 Knoten und 6807 Kanten, und einer Gesamt-Zeit von $\sim 0.18s$. Bei einer Größe von 83 ist die Ausführung bei der Berechnung der starken Zusammenhangskomponenten mit einem `StackOverflowError` abgebrochen. Um die Analyse größerer Statecharts zu unterstützen, müsste demnach die Berechnung verändert werden.

size	default strategy		default w/o independence opt.	
	solving time	overall time	solving time	overall time
2	0.0007	0.0055	0.0008	0.0054
4	0.0017	0.02	0.0031	0.0303
8	0.0112	0.0919	0.0214	0.1395
16	0.0451	0.332	0.1555	1.0215
32	0.2758	1.6631	1.9238	8.0436
64	2.056	11.2853	37.7648	58.6112

Tabelle 9.3: Vergleich der Performanz der Unabhängigkeitsoptimierung in Sekunden

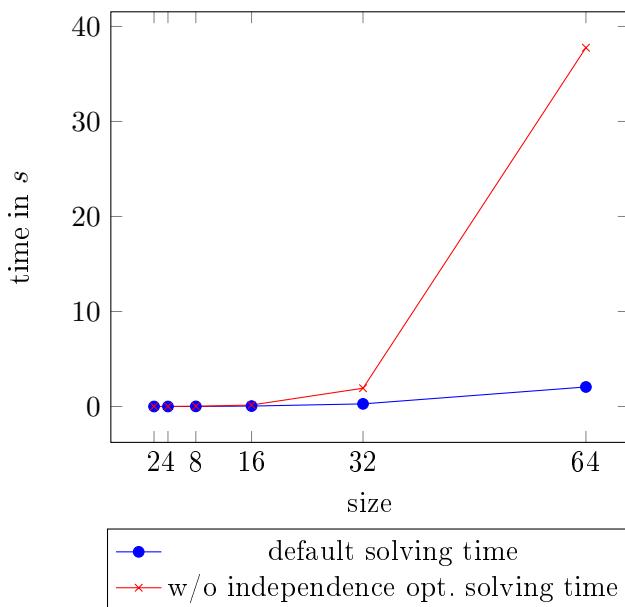


Abbildung 9.6: Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich der Unabhängigkeitsoptimierung (siehe Tabelle 9.3)

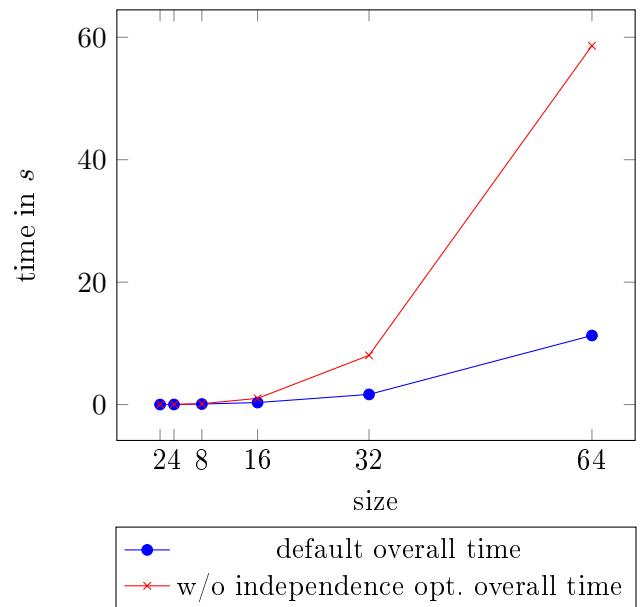


Abbildung 9.7: Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich der Unabhängigkeitsoptimierung (siehe Tabelle 9.3)

9.2.4 Evaluierung der Unabhängigkeitsoptimierung

In diesem Abschnitt wird die Unabhängigkeitsoptimierung im Vergleich zur default Strategie evaluiert. Dazu wird das Statechart ohne diese Optimierung analysiert, der Rest der Strategie ist mit der default Strategie identisch. Die Performance ist in Tabelle 9.3 und Abbildung 9.7 dargestellt.

Es ist gut zu erkennen, wie effektiv die Unabhängigkeitsoptimierung an diesem Statechart arbeitet. Bei einer kleineren Größe bis zu 8 sind die Zeiten noch recht identisch. Ab einer Größe von 16 wird die Performance viel schlechter, sodass bei einer Größe von 64 ca. die 5-fache Zeit der default Strategie benötigt wird. Insgesamt ist zudem erkennbar, dass der Anteil der Solving-Zeit deutlich höher ist. Bei einer Größe von 8 ist der Anteil $\sim 15\%$, bei einer Größe von 64 liegt der Anteil hingegen bei $\sim 58\%$.

Zu beachten ist aber, dass das Statechart optimal für die Unabhängigkeitsoptimierung ist, da in jedem Zyklus eine neue Variable genutzt wird. Insgesamt funktioniert die Optimierung am besten, wenn Variablen häufiger konkrete Werte ohne Abhängigkeit zu anderen Variablen zugewiesen werden. In diesem Fall kann die Optimierung die Pfadbedingung reduzieren und bringt einen enormen Performanz Vorteil.

Verbessern ließe sich die Unabhängigkeitsoptimierung mit einem Cache, der die Resultate von einzelnen Solving Aufrufen der Pfadbedingung speichert, so wie dies in EXE ([10]) der Fall ist.

9.2.5 Evaluierung des Kontext-Solving

size	default strategy		default w/o context-solving	
	solving time	overall time	solving time	overall time
2	0.0007	0.0055	0.0009	0.0072
4	0.0017	0.02	0.0021	0.0304
8	0.0112	0.0919	0.0109	0.1036
16	0.0451	0.332	0.0436	0.428
32	0.2758	1.6631	0.255	1.9378
64	2.056	11.2853	1.9555	11.1038

Tabelle 9.4: Vergleich der Performance des Kontext-Solving in Sekunden

In diesem Abschnitt wird das Kontext-Solving im Vergleich zur default Strategie evaluiert. Dazu wird das Statechart ohne diese Optimierung analysiert, der Rest der Strategie ist mit der default Strategie identisch. Die Performance ist in Tabelle 9.4 und Abbildung 9.9 dargestellt.

Bei diesem Vergleich sind fast keine Unterschiede zwischen den beiden Strategien erkennbar bzw. sind diese nicht aussagekräftig. Daran ist erkennbar, dass das Kontext-Solving

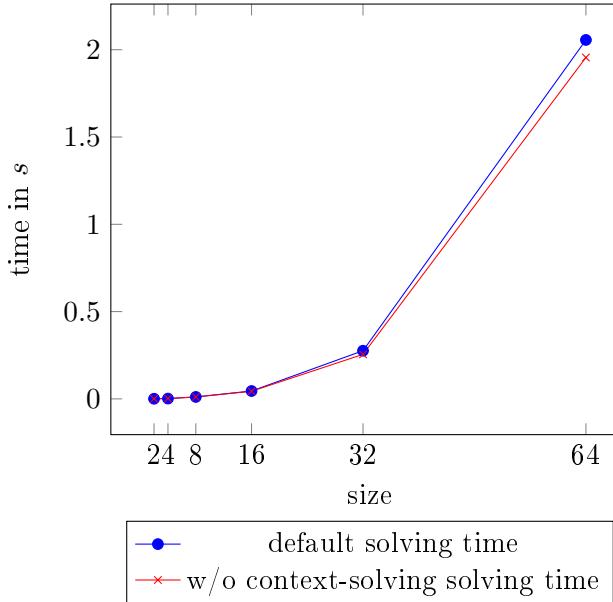


Abbildung 9.8: Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich des Kontext-Solving (siehe Tabelle 9.4)

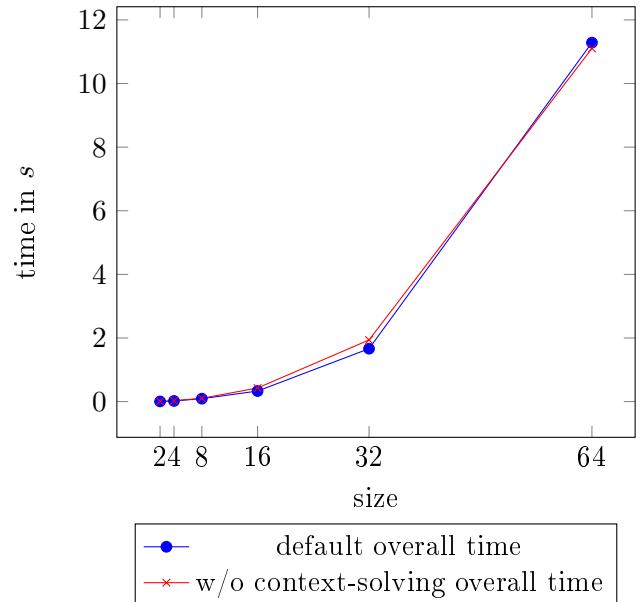


Abbildung 9.9: Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich des Kontext-Solving (siehe Tabelle 9.4)

in diesem Fall keinen Nutzen bringt. In anderen Fällen, bzw. verglichen mit anderen Strategien, bringt die Optimierung sehr wohl einen Performance Gewinn. Dies ist in Abschnitt 9.2.10 beschrieben.

9.2.6 Evaluierung der Binäre-Suche Ergebnisverbreitung

size	default strategy		default w/ binary search	
	solving time	overall time	solving time	overall time
2	0.0007	0.0055	0.0008	0.0042
4	0.0017	0.02	0.0007	0.018
8	0.0112	0.0919	0.0058	0.1084
16	0.0451	0.332	0.0261	0.3611
32	0.2758	1.6631	0.1723	1.8627
64	2.056	11.2853	1.5105	11.8874

Tabelle 9.5: Vergleich der Performance der binary-search Ergebnisverbreitung in Sekunden

In diesem Abschnitt wird die binäre-Suche Ergebnisverbreitung im Vergleich zur default Strategie evaluiert. Dazu wird das Statechart mit binärer-Suche analysiert, der Rest der Strategie ist mit der default Strategie identisch. Die Performance ist in Tabelle 9.5 und Abbildung 9.11 dargestellt.

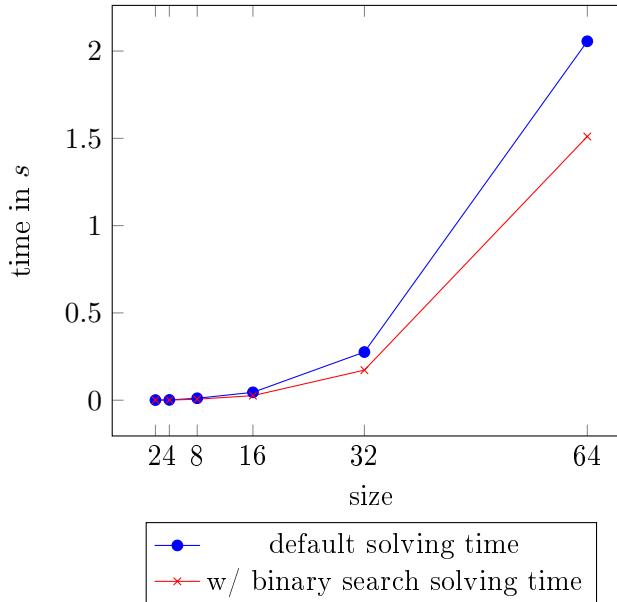


Abbildung 9.10: Graphische Visualisierung der Solving-Zeit des Performance-Vergleich der binary-search Ergebnisverbreitung (siehe Tabelle 9.5)

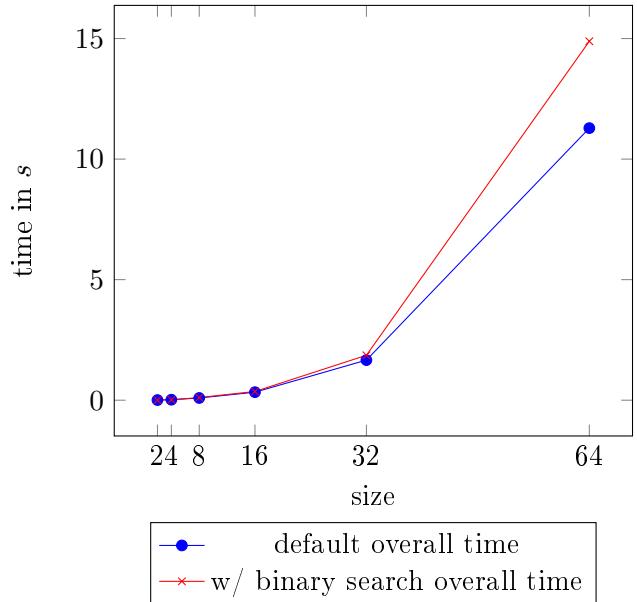


Abbildung 9.11: Graphische Visualisierung der Gesamt-Zeit des Performance-Vergleich der binary-search Ergebnisverbreitung (siehe Tabelle 9.5)

Erkennbar ist, dass die Zeit für die gesamte Analyse nahezu identisch ist. Im Vergleich dazu ist die Zeit gesunken, welche für das Solving gebraucht wird. Bei einer Größe von 8 wird nur knapp die Hälfte, bei einer Größe von 64 nur $\sim 73\%$ der Zeit benötigt. Genauso wie beim Kontext-Solving tritt aber im Vergleich mit anderen Strategien ein deutlicherer Performanz Gewinn von oldest-search gegenüber binary-search auf. Dies ist in 9.2.10 beschrieben.

9.2.7 Evaluierung der DFS Konstruktion

size	default strategy		default w/ dfs construction	
	solving time	overall time	solving time	overall time
2	0.0007	0.0055	0.0006	0.0044
4	0.0017	0.02	0.0018	0.0209
8	0.0112	0.0919	0.0109	0.0898
16	0.0451	0.332	0.0469	0.3328
32	0.2758	1.6631	0.2714	1.6418
64	2.056	11.2853	2.1395	11.7803

Tabelle 9.6: Vergleich der Performanz der DFS-Konstruktion in Sekunden

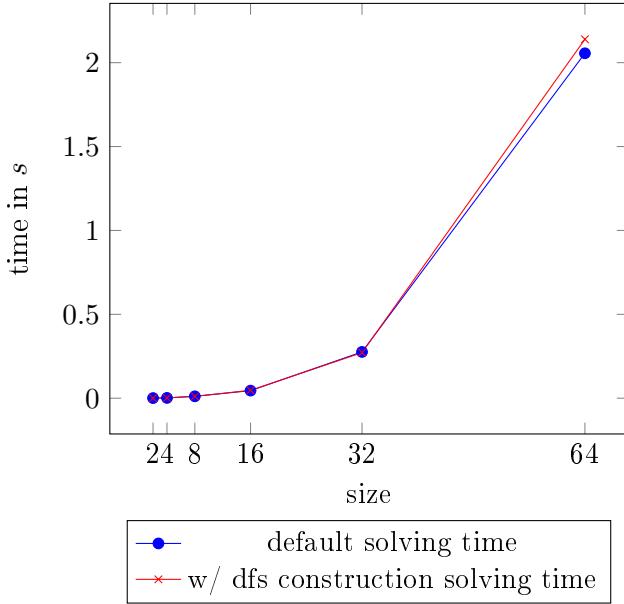


Abbildung 9.12: Graphische Visualisierung der Solving-Zeit des Performance-Vergleich der DFS-Konstruktion (siehe Tabelle 9.6)

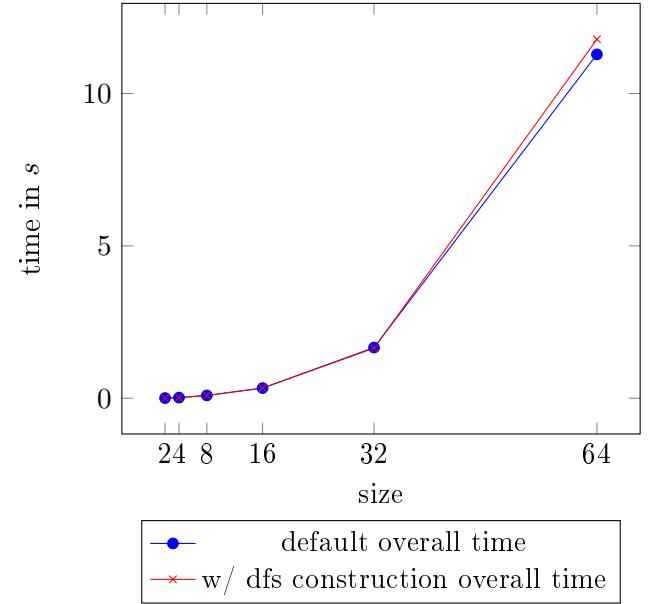


Abbildung 9.13: Graphische Visualisierung der Gesamt-Zeit des Performance-Vergleich der DFS-Konstruktion (siehe Tabelle 9.6)

In diesem Abschnitt wird die DFS Konstruktion im Vergleich zur default Strategie evaluiert. Dazu wird das Statechart mit der DFS Konstruktion analysiert, der Rest der Strategie ist mit der default Strategie identisch. Die Performance ist in Tabelle 9.6 und Abbildung 9.13 dargestellt. In diesem Fall ist an der Performance kein signifikanter Unterschied erkennbar. Hier würde sich anbieten die Performance durch weitere Optimierungen zu steigern, welche mit DFS zusammenarbeiten. Beispielsweise könnte dies mit nativer Z3 Erfüllbarkeitsbestimmung (6.6) mit verbesserter push und pop Benutzung erreicht werden. Weiterhin würde die Strategie gut mit einer, im Kontext der Erreichbarkeitsanalyse befindenden, Optimierung zusammenarbeiten. Diese würde schauen, ob vom aktiven Zustand noch ein bis dahin unerreichbarer Knoten erreichbar ist. Wenn dies nicht der Fall ist, braucht die Konstruktion bestimmte Zweige des SET für eine korrekte Erreichbarkeitsanalyse nicht zu konstruieren.

9.2.8 Evaluierung von jConstraints

In diesem Abschnitt wird die Erfüllbarkeitsbestimmung mit jConstraints im Vergleich zur default Strategie evaluiert. Dazu wird das Statechart mit jConstraints analysiert, der Rest der Strategie ist mit der default Strategie identisch. Die Performance ist in Tabelle 9.7 und Abbildung 9.15 dargestellt. Erkennbar ist, dass sich durch die Verwendung von nativem Z3 insgesamt eine leichte Performance Verbesserung ergibt. Dies deckt sich auch mit Tests an anderen Statecharts, bei denen die Ergebnisse eindeutiger waren. Die Bestimmung

size	default strategy		default w/ jConstraints-Solver	
	solving time	overall time	solving time	overall time
2	0.0007	0.0055	0.0029	0.0353
4	0.0017	0.02	0.0062	0.0608
8	0.0112	0.0919	0.022	0.148
16	0.0451	0.332	0.0606	0.4023
32	0.2758	1.6631	0.328	1.7996
64	2.056	11.2853	2.3749	11.9567

Tabelle 9.7: Vergleich der Performanz mit jConstraints in Sekunden

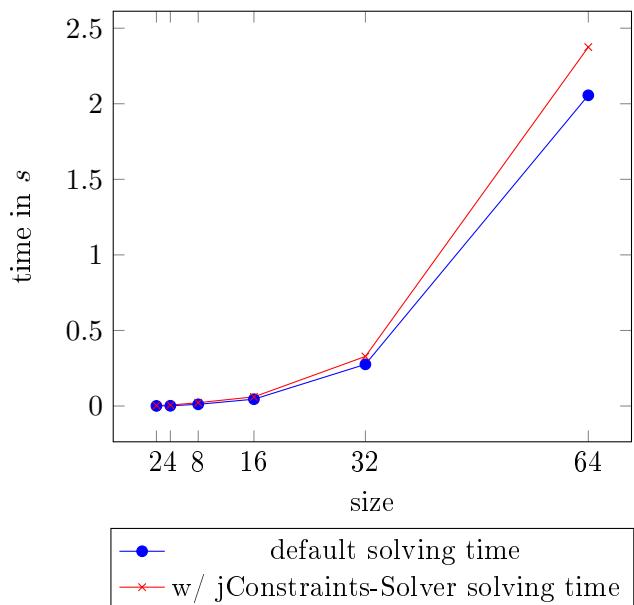


Abbildung 9.14: Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich mit jConstraints (siehe Tabelle 9.7)

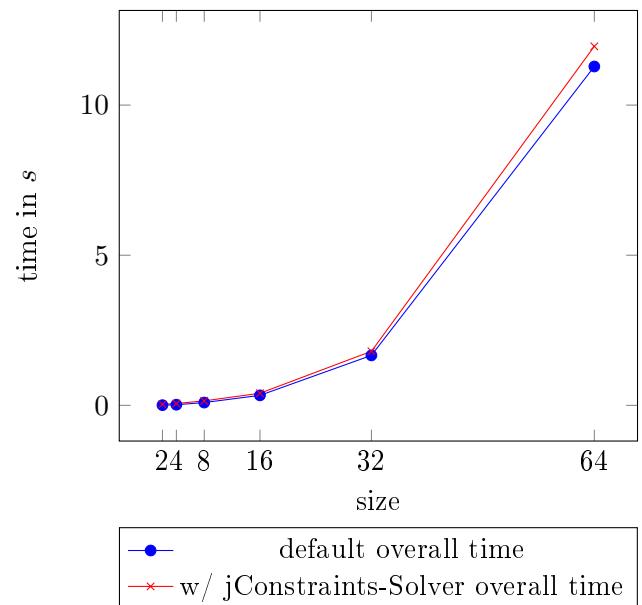


Abbildung 9.15: Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich mit jConstraints (siehe Tabelle 9.7)

mit jConstraints hat den Nachteil, dass die Konvertierung von SMT-LIB2 in jConstraints Objekte zu viel Overhead erzeugt und dadurch langsamer ist. Dieses wurde durch Test bestätigt. Eventuell würde hier eine direkte Konvertierung von Yakindu-Expressions in jConstraints-Objekte eine bessere Performance als natives Z3 liefern. Weiterhin könnte man auch konvertierte Objekte speichern und immer nur neu hinzugefügte Bedingungen der Pfadbedingung konvertieren, um Performance zu gewinnen. Insgesamt sind hier noch einige Verbesserungsmöglichkeiten offen.

9.2.9 Evaluierung der Kostenfunktion

size	default strategy		default w/ satCost 1	
	solving time	overall time	solving time	overall time
2	0.0007	0.0055	0.0016	0.0037
4	0.0017	0.02	0.005	0.0109
8	0.0112	0.0919	0.0248	0.0467
16	0.0451	0.332	0.1671	0.2739
32	0.2758	1.6631	1.6223	2.4341
64	2.056	11.2853	17.9563	25.908
size	default w/ satCost 10		default w/ satCost 50	
	solving time	overall time	solving time	overall time
2	0.005	0.0041	0.0008	0.0054
4	0.0037	0.013	0.0021	0.0235
8	0.0129	0.0489	0.0095	0.1956
16	0.0676	0.1928	0.0379	0.5216
32	0.4076	1.4816	0.2164	2.1827
64	4.0896	12.9983	0.5873	13.1669

Tabelle 9.8: Vergleich der Performanz der Kostenfunktion in Sekunden

In diesem Abschnitt wird die Kostenfunktion der Erfüllbarkeitsbestimmung evaluiert. Dazu wird das Statechart mit verschiedenen Erfüllbarkeitskosten analysiert, der Rest der Strategie ist mit der default Strategie identisch. Die Performance ist in Tabelle 9.8 und Abbildung 9.17 dargestellt. Zu erkennen ist, dass die Kostenfunktion bei einer Größe von 64 optimal bei ca. 30 funktioniert, wie bei der default Strategie eingestellt ist. Bei Kosten von 10 oder 50 ist die Performanz leicht schlechter. Bei Kosten von 1, also ausgeschalteter Kostenfunktion, ist die Performanz deutlich schlechter und braucht mehr als doppelt so lange. Bei kleineren Statecharts ist die Engine mit Kosten von 10 am schnellsten, was beispielsweise bei einer Größe von 16 zu sehen ist. Insgesamt ist gut erkennbar, dass die Solving-Zeit mit höheren Kosten einen immer kleiner werdenden Anteil erhält.

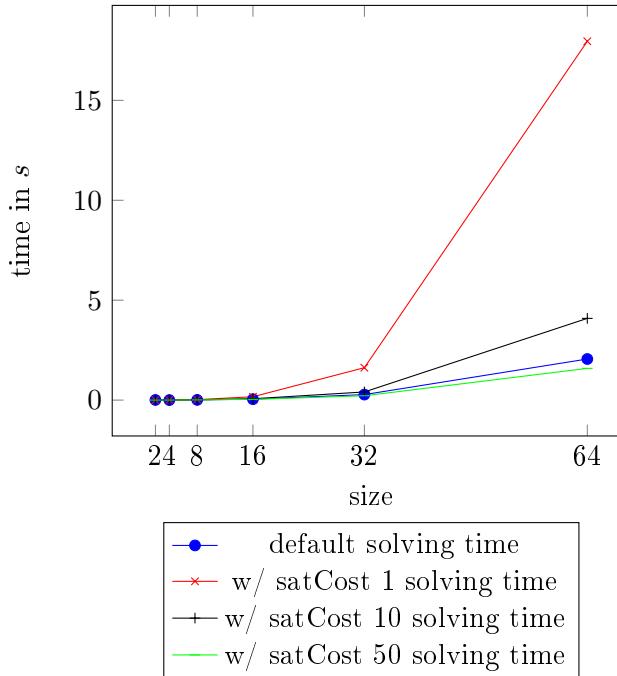


Abbildung 9.16: Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich der Kostenfunktion (siehe Tabelle 9.8)

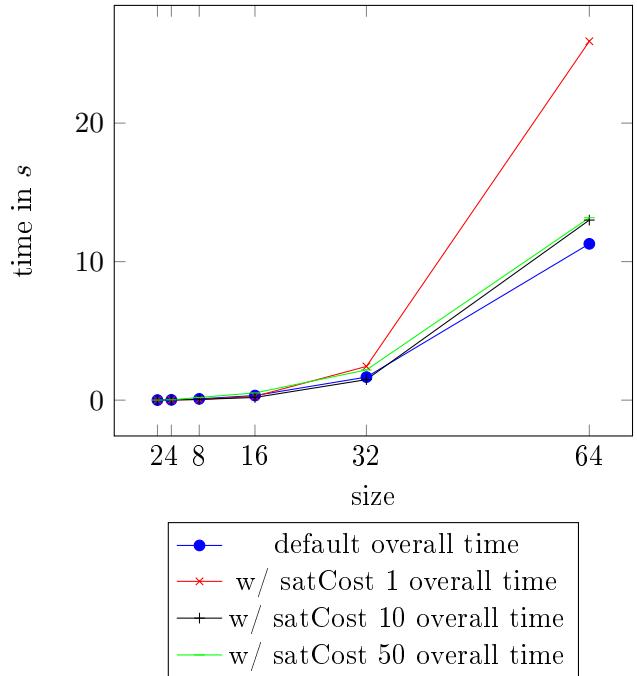


Abbildung 9.17: Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich der Kostenfunktion (siehe Tabelle 9.8)

9.2.10 Evaluierung weiterer Strategien

Weiterhin wurden folgende Strategien evaluiert:

1. default Strategie ohne Kontext-Solving und ohne Unabhängigkeitsoptimierung
2. default Strategie mit binary-search und ohne Kontext-Solving und Unabhängigkeitsoptimierung
3. default Strategie ohne Transitionszyklenelimination und ohne Kontext-Solving

Die Resultate sind in Tabelle 9.9 dargestellt. An der Strategie 1. ist gut erkennbar, dass die Kontext-Solving Optimierung ohne die Unabhängigkeitsoptimierung einen besseren Performanz-Gewinn erzielt. Bei einer Größe von 64 wurden $\sim 79.6s$ benötigt. Mit eingeschaltetem Kontext-Solving brauchte die Analyse hingegen nur $\sim 58.6s$ (siehe 9.3). Bei Strategie 2. ist erkennbar, dass binary-search ohne die Kontext-Solving und Unabhängigkeitsoptimierung langsamer als oldest-search ist. Bei einer Größe von 64 wurden $\sim 110.8s$ benötigt, während mit oldest-search $\sim 79.6s$ benötigt wurden. Hingegen ist bei dieser Strategie der Anteil der Solving-Zeit deutlich geringer.

Bei Strategie 3. ist auch der Nutzen von Kontext-Solving erkennbar. Mit Kontext-Solving und ohne Transitionszyklenelimination benötigt die Analyse $\sim 190.95s$ bei einer Größe von 64. Ohne Kontext-Solving und Transitionszyklenelimination hingegen $\sim 209.5s$.

size	default strategy		1	
	solving time	overall time	solving time	overall time
2	0.0007	0.0055	0.0011	0.0085)
4	0.0017	0.02	0.004	0.0601
8	0.0112	0.0919	0.0267	0.2545
16	0.0451	0.332	0.1847	2.5372
32	0.2758	1.6631	2.0898	18.4301
64	2.056	11.2853	38.7557	79.5583

size	2		3	
	solving time	overall time	solving time	overall time
2	0.001	0.0045	0.0014	0.012
4	0.0023	0.0265	0.0094	0.1181
8	0.0139	0.1929	0.0445	0.0352
16	0.0874	1.1257	0.303	5.3956
32	1.2436	9.7399	3.1189	18.1618
64	28.3944	110.8079	30.6385	209.5077

Tabelle 9.9: Vergleich der Performanz verschiedener Strategien in Sekunden

9.3 Verbesserungen zum ursprünglichen Algorithmus

Der entwickelte Algorithmus aus [11] wurde in dieser Arbeit deutlich verbessert. Es wurde die Priorisierung von Transitionen umgesetzt, lokale Reaktionen in Statecharts sind analysierbar und können durch eine Zyklenenelimination keine unendlichen Ausführungspfade erzeugen. Die Transitionszyklenenlimination wurde verbessert, sodass eine modifizierbare Anzahl an Transitionswiederholungen, bzw. eine präventive Strategie einstellbar sind. Weiterhin wird bei dieser Elimination nur das Wissen über Variablen vergessen, die im Zyklus veränderbar sind. Bei dem vorherigen Algorithmus wurde das Wissen über alle Variablen des Statecharts zurückgesetzt. Dies hat sich auch durch die Korrektheitstests bestätigt, bei denen der ursprüngliche Algorithmus insgesamt eine niedrige Genauigkeit aufwies. Weitere Verbesserungen betreffen die Optimierungen der Erfüllbarkeitsbestimmung, wie die Unabhängigkeitsoptimierung oder Kostenfunktion. Zudem wurde die Engine modular entwickelt, sodass weitere Optimierungen oder Veränderungen einfach funktionieren. Die größte Verbesserung bildet aber der konstruierte SET. Dieser ermöglicht zum einen eine bessere Darstellung aller Ausführungspfade und die Möglichkeit der Zurückverfolgung der Konstruktion. Zum anderen eignet dieser sich nicht nur für eine Erreichbarkeitsanalyse, sondern auch für eine Vielzahl weiterer Analysen.

In Tabelle 9.10 und Abbildung 9.18 ist der Performanz Vergleich zu erkennen, bei dem der ursprüngliche Algorithmus einmal gegenüber der default Strategie und der präventiven

Transitionszyklenelimination verglichen wird. Erkennbar ist, dass der ursprüngliche Algorithmus gerade bei kleineren Statechart einen Overhead besitzt und nicht schneller als $0.3s$ wird. Bei einer Größe von 16 ist der Algorithmus leicht schneller als die default Strategie, bei einer Größe von 64 gewinnt wieder die default Strategie. Im Gegensatz zur präventiven Elimination ist der Algorithmus aber deutlich langsamer. Es ist aber zu sagen, dass die gemessenen Zeiten nur bedingt aussagekräftig sind. Der ursprüngliche Algorithmus hat die meisten Zustände als *vielleicht* erreichbar markiert und konnte keine definitive Antwort geben. Zusätzlich ist dieser bei einer Größe von 64 bei der Ergebnisausgabe mit einem **HeapError** abgebrochen. Insgesamt hat die entwickelte Engine also sowohl in Korrektheit als auch in Performanz und Zuverlässigkeit besser abgeschnitten.

size	default strategy	original algorithm	default with preventive elimination
2	0.0055	0.031	0.008
4	0.02	0.0351	0.0013
8	0.0919	0.475	0.0022
16	0.332	0.588	0.0048
32	1.6631	1.35	0.0129
64	11.2853	15.3	0.0367

Tabelle 9.10: Vergleich der Gesamt-Zeit Performanz zum ursprünglichen Algorithmus in Sekunden

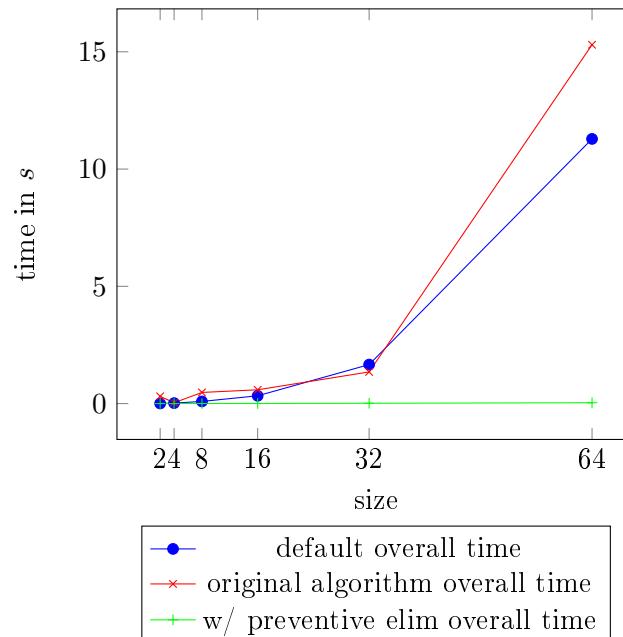


Abbildung 9.18: Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich zum ursprünglichen Algorithmus (siehe Tabelle 9.10)

9.4 Folgerungen

Die Engine funktioniert in allen Tests zuverlässig und korrekt und bildet eine gute Verbesserung zum ursprünglichen Algorithmus. Bei der Evaluation wurden alle Komponenten getestet und eine Code-Coverage von 85% erreicht. Da nur ein Bruchteil aller möglichen Strategien an einem synthetischen Statechart evaluiert wurde, bildet dieser Test kein vollständiges Bild, sondern gibt nur einen Überblick über die Performanz. Insgesamt wäre die Evaluierung vieler weiterer Strategien und Statecharts interessant, um die best mögliche Strategie zu finden und die Zusammenarbeit der einzelnen Optimierungen zu bewerten. Die mit Abstand schnellste Strategie dieser Evaluation ist die default Strategie mit präventiver Transitionszyklenelimination. Diese dauerte in allen Tests mit weniger als 6800 Knoten und Kanten unter 0.2s. Bei der default Strategie wurde sich gegen die präventive und für die konservative Transitionszyklenelimination mit einer Transitionswiederholung entschieden, da der Performanz Verlust für die Möglichkeit einer genaueren Analyse in Kauf genommen wird. Bei größeren Statecharts wird hingegen die präventive Elimination empfohlen. Die maximale Größe von zu analysierenden Statecharts war durch die Berechnung der starken Zusammenhangskomponenten gegeben. Bei den synthetischen Statecharts war dies bei jeweils ca. 6800 Knoten und Kanten der Fall.

Weiterhin wurden einige Verbesserungsmöglichkeiten genannt, mit denen die Optimierungen noch performanter werden können. Alles in allem arbeitet die Engine sehr schnell, zuverlässig und korrekt.

Kapitel 10

Zusammenfassung

In dieser Arbeit wird eine statische Analyse von Statecharts mittels symbolischer Ausführung beschrieben. Die Engine wurde modular entwickelt, und in Verbindung mit Yakindu Statechart Tools umgesetzt. In Kapitel 2 werden die Voraussetzungen für das Verständnis der Engine erläutert. Kapitel 3 beschreibt den allgemeinen Ablauf der Engine und gibt einen Überblick über die Vorbedingungen und Limitierungen der Engine.

Das Kernstück der Arbeit – der symbolische Ausführungsbaum – wird in Kapitel 4 vorgestellt. Dieser erlaubt eine Darstellung aller möglichen Ausführungspfade des zu analysierenden Statecharts. Dabei werden zum einen die Knoten des Baumes, und die dort verwalteten Informationen, erläutert. Weiterhin werden Regeln eingeführt, welche für die Generierung von Kindknoten angewandt werden und so die symbolische Ausführung vorantreiben.

Kapitel 5 beschreibt die Handhabung der Pfadexplosion durch Zyklen, eines der Hauptprobleme symbolischer Analyse. Hier wurden zwei Verfahren vorgestellt, welche für die Elimination unendlicher Ausführungspfade verantwortlich sind. Zum einen wird die Transitionszyklenelimination vorgestellt, welche unendliche Transitionswiederholungen mithilfe von starken Zusammenhangskomponenten verhindert. Zum anderen wird die Elimination unendlicher lokaler Reaktionen innerhalb von Zuständen vorgestellt.

Kapitel 6 erläutert den Ablauf der Erfüllbarkeitsbestimmung und der Optimierungen, die für diese umgesetzt wurden. Beispielsweise wird Kontext-Solving und die Unabhängigkeitsoptimierung, sowie die eigentliche Bestimmung mittels SMT-Solver, vorgestellt.

Nachdem zuvor alle Komponenten des symbolischen Ausführungsbaumes erläutert wurden, wird in Kapitel 7 der Ablauf seiner Konstruktion beschrieben. Zudem werden die verschiedenen Strategien erläutert, mit welchen die Engine diese Konstruktion ausführt.

In Kapitel 8 wird die Erreichbarkeitsanalyse und weitere Analysen, die auf dem Ausführungsbaum möglich sind, besprochen. Als zusätzliche Analyse wird beispielsweise die Testfallgenerierung erläutert.

Abschließend wird die Engine in Kapitel 9 evaluiert. Dazu wird als erstes die Korrektheit bewertet. Je nach gewählter Transitionszyklenelimination kann es vorkommen, dass Zustände und Transitionen fälschlicherweise als erreichbar markiert werden, obwohl diese es nicht sind. Andersrum tritt dies aber nicht auf. Danach wird die Performanz anhand verschiedener Strategien verglichen. Zusätzlich werden Verbesserungen aufgezählt, mit denen die einzelnen Komponenten und Optimierungen noch performanter arbeiten könnten.

Insgesamt wurde herausgefunden, dass die entwickelte Engine zuverlässig und korrekt arbeitet. Die benötigte Zeit der Analyse ist für einen Nutzer zudem sehr zufriedenstellend und hat mit der schnellsten Strategie und großen synthetisch konstruierten Statecharts unter 0.2s benötigt. Daher wurde eine Strategie empfohlen, welche etwas langsamer, dafür aber genauere Erreichbarkeitsergebnisse liefert. Die benötigte Zeit für diese Statecharts mit 1000 Knoten lag unter 2 Sekunden, ist also immer noch schnell.

10.1 Ausblick

Die entwickelte Engine besitzt viele verschiedene Ansatzpunkte, an denen diese verbessert werden kann. Nachfolgend werden einige dieser Bereiche kurz erläutert.

Die Fachliteratur liefert einige Ansätze, mit denen die Analyse performanter und genauer umgesetzt werden kann. Beispielsweise gibt es im Bereich der Pfadexplosion viele Ideen, wie Zyklen performant behandelbar sind, ohne die Genauigkeit einzubüßen. Der Bereich der Erfüllbarkeitsbestimmung könnte durch Ansätze, wie dem Bedingungscaching in EXE [10], weiter verbessert werden. Im Kontext der Erreichbarkeitsanalyse könnte eine Optimierung entworfen werden, durch die nur ein Bruchteil des symbolischen Ausführungsbaumes konstruiert werden muss. Diese würde nur Ausführungspfade konstruieren, durch die bisher unerreichbare Zustände als erreichbar markiert werden könnten.

Eine offensichtliche Verbesserung ist die Integration weiterer Elemente von Statecharts. Beispielsweise könnten weitere Variabtentypen, die Benutzung von Funktionen oder mehrfache Regionen unterstützt werden. Dies würde die Engine besonders im kommerziellen Bereich interessanter machen.

Aufschlussreich wäre auch die Performanz Evaluation weiterer Statecharts. In Verbindung mit maschinellem Lernen könnte man die Strategien herausfinden, welche für eine bestimmte Art von Statecharts geeignet sind. Diese könnten beispielsweise in der Größe, Komplexität der Zyklen und Expressions oder ähnlichem unterteilt werden.

Weiterhin sind weitere Analysen auf dem symbolischen Ausführungsbaum interessant. Besonders fällt hierbei die Testfallgenerierung auf. Mit einer gut umgesetzten Generierung könnte die Entwicklungszeit von Statecharts massiv reduziert werden, indem alle möglichen Ausführungspfade von diesen Tests abgedeckt werden. Zusätzlich erlaubt es bessere Tests, da erfahrungsgemäß Testfälle vergessen werden.

Anhang A

Weitere Informationen

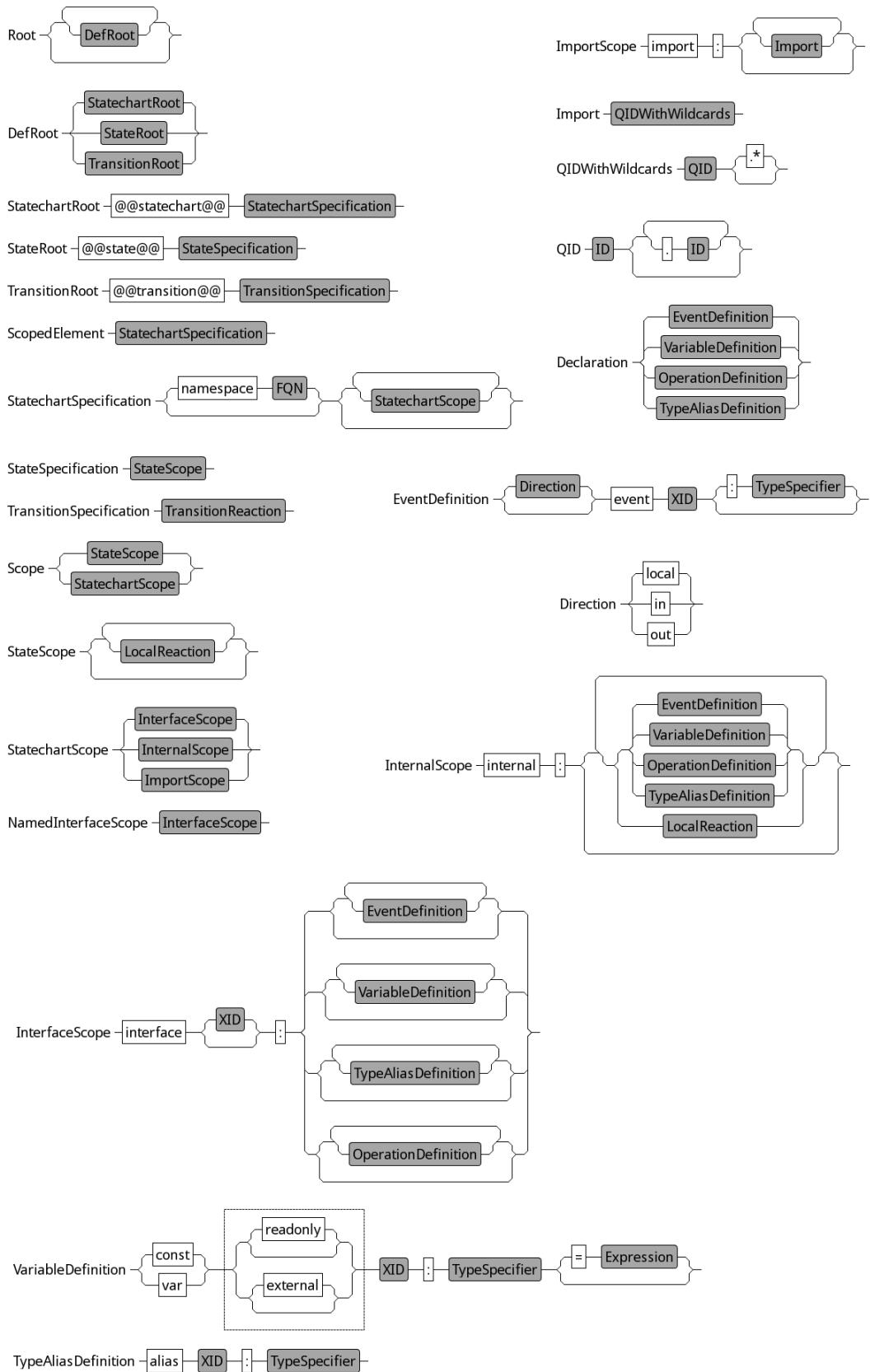


Abbildung A.1: Grammatik der Statechart Elemente in Yakindu ScT Teil 1

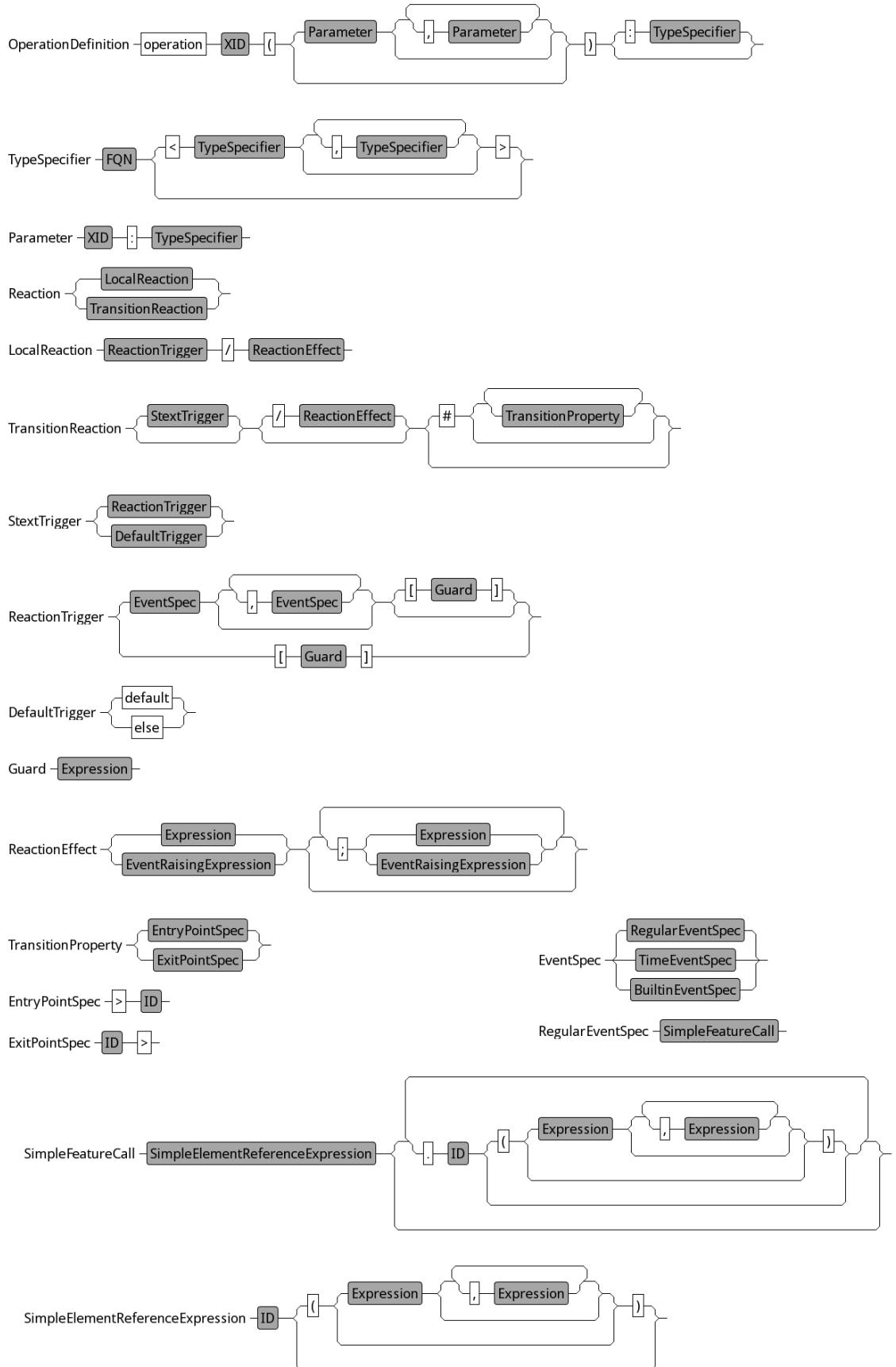


Abbildung A.2: Grammatik der Statechart Elemente in Yakindu ScT Teil 2

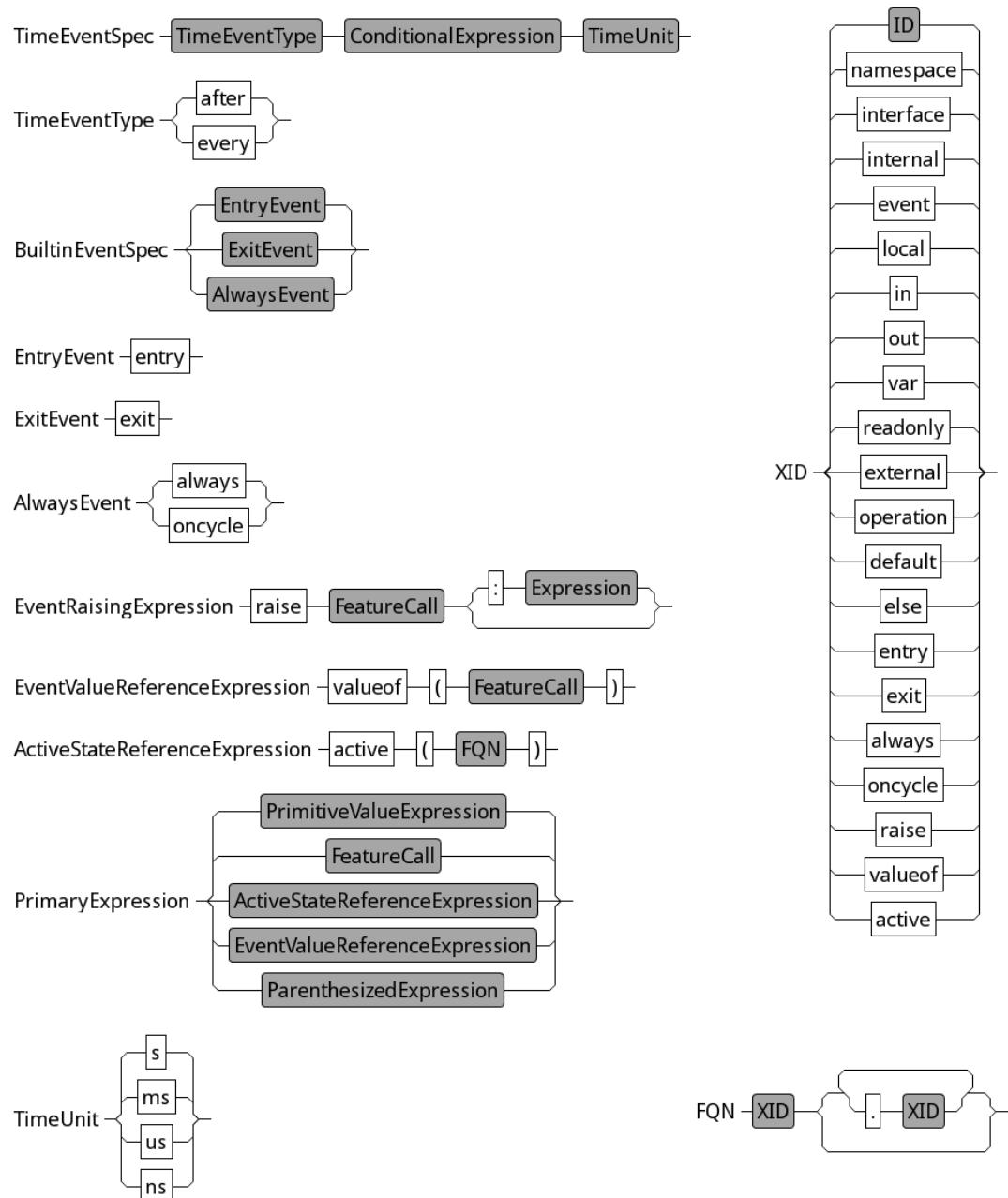


Abbildung A.3: Grammatik der Statechart Elemente in Yakindu ScT Teil 3

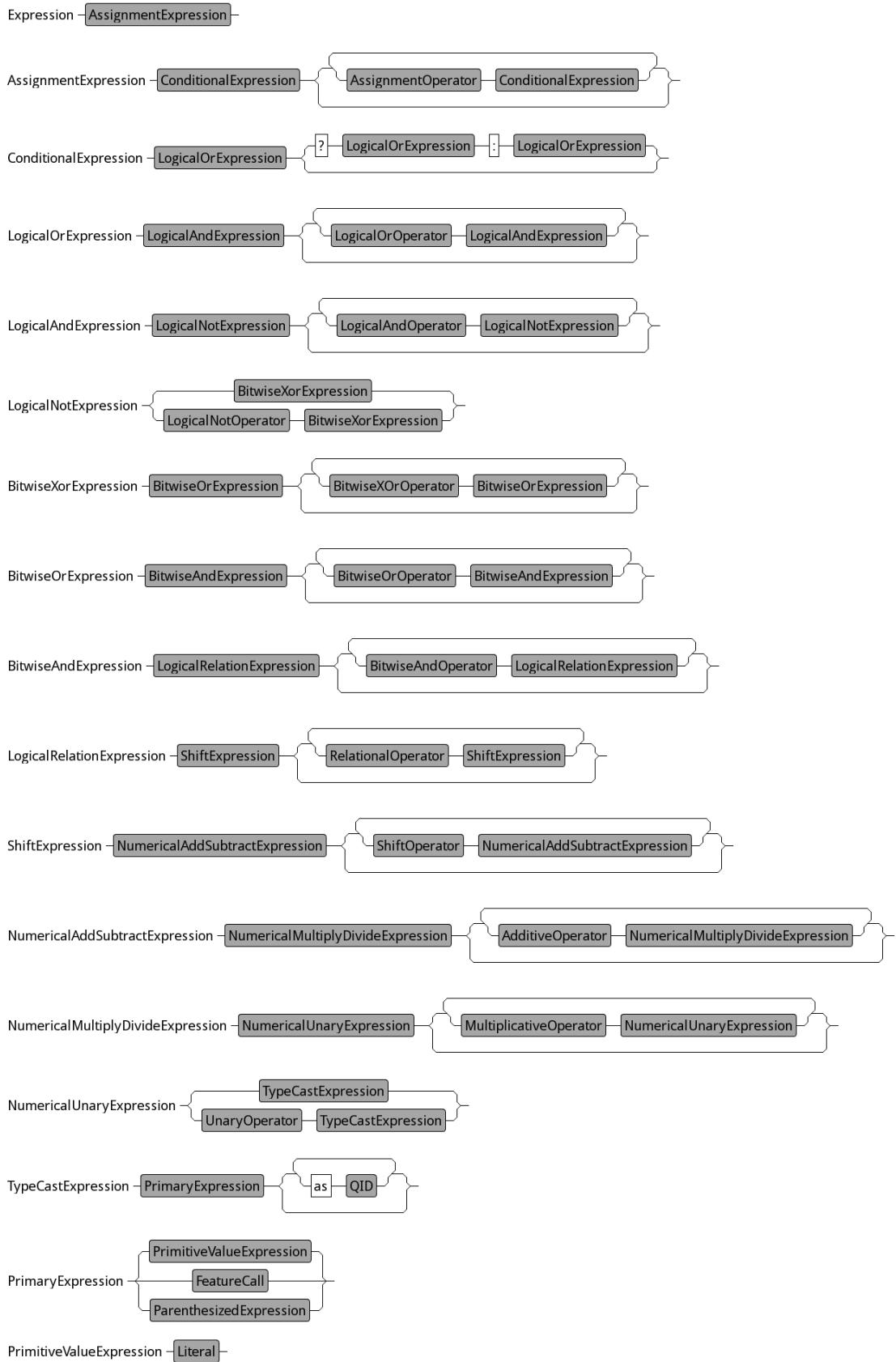


Abbildung A.4: Expression Grammatik in Yakindu ScT Teil 1

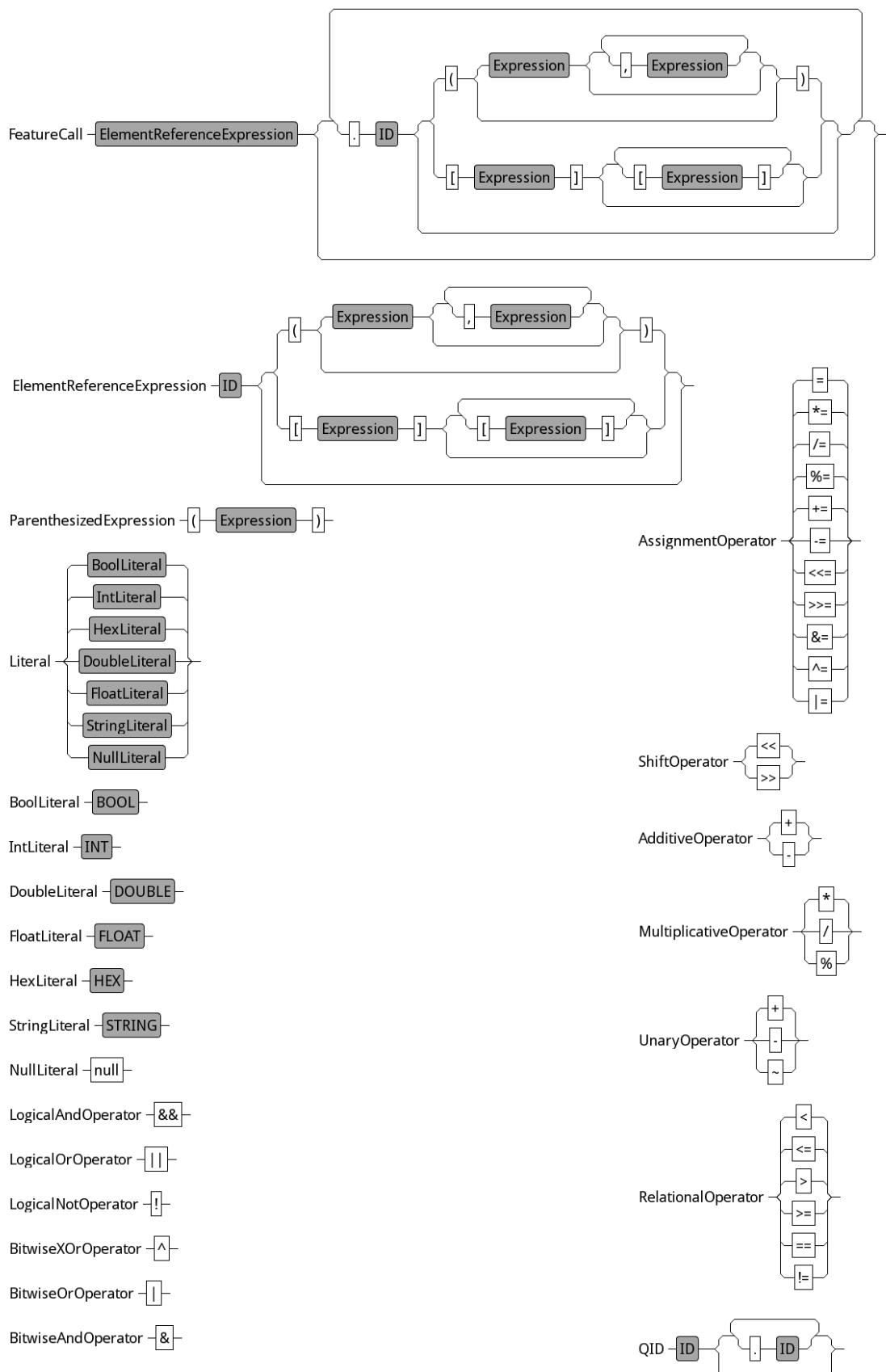


Abbildung A.5: Expression Grammatik in Yakindu ScT Teil 2

Abbildungsverzeichnis

2.1	Yakindu ScT Logo [17]	3
2.2	Beispielhaftes Statechart welches in Yakindu ScT modelliert ist ([17])	4
2.3	Grammatik der <code>SequenceBlockExpression</code>	7
2.4	Beispiel Statechart mit unerreichbaren Zuständen C und D	10
2.5	Beispiel starker Zusammenhangskomponente eines Graphen	13
3.1	Allgemeiner Ablauf der symbolischen Ausführung als Flowchart	16
3.2	Ablauf der Vereinfachung des Statecharts dargestellt als Flowchart.	16
3.3	Beispielhafte Vereinfachung von Transitionen eines Statecharts. Die Zahlen an den Transitionen beschreiben die Prioritäten. Der Guard der Transition $C \rightarrow D$ wird als Widerspruch erkannt und entfernt. Der Guard der Transition $C \rightarrow E$ wird als Tautologie erkannt und zu <code>true</code> vereinfacht. Die Transition $E \rightarrow G$ wird von der Transition $E \rightarrow F$ überschattet und entfernt.	17
3.4	Beispiele zur Transformation von Expressions. Links steht jeweils die ursprüngliche Expression, rechts die transformierte. 1 zeigt die Transformation einer Numerischen Expression, 2 die einer Booleschen. Beispiele 3 und 4 zeigen die Auflösung von Post-Fix Ausdrücken	18
4.1	Beispielhafter Knoten des SET mit allen verwalteten Informationen.	23
4.2	Ablauf der <code>TransitionRule</code> als Flowchart	25
4.3	Beispielhafte Anwendung der <code>TransitionRule</code> , welche die Durchführung von Transitionen zeigt. Knoten 2 enthält die exit-und entry-Reaktion als <code>ConditionalExpression</code> . Bei Knoten 3 wird der negierte Guard $\neg(a > 1)$ von Transition 1 ergänzt, da die Transitionen 2 und 1 den selben Trigger besitzen. Der Trigger von Transition 3 überlappt mit keinem anderen Trigger.	26
4.4	Ablauf der Durchführung lokaler Reaktionen bei der Anwendung der <code>TransitionRule</code> als Flowchart	27

4.5 Beispielhafte Anwendung der TransitionRule , welche die Durchführung lokaler Reaktionen zeigt. Für jede Teilmenge der Triggermenge $\{evt_1, evt_2\}$ entsteht ein Kindknoten, bei welchem die lokalen Reaktionen durchgeführt wurden. Beispielsweise ist in Knoten 4 die lokale Reaktion $evt_2[y]/a = c$ durch die Teilmenge $\{evt_2\}$ durchgeführt worden.	28
4.6 Anwendung der ConditionalRule mit ConditionalExpression $x?y : z$. Dem „true“-Kind 2 wurde die Bedingung x und TrueCaseExpression y den aktiven Anweisungen hinzugefügt. Dem „false“-Kind 3 die negierte Bedingung $\neg x$ und FalseCaseExpression z	29
4.7 Anwendung der AssignmentConditionalRule mit ConditionalExpression als Assignor. Dem „true“-Kind 2 wurde die Bedingung x und TrueCaseExpression als Zuweisung $a = y$ den aktiven Anweisungen hinzugefügt. Dem „false“-Kind 3 die negierte Bedingung $\neg x$ und FalseCaseExpression als Zuweisung $a = z$	30
4.8 Anwendungsbeispiel der SequenceBlockRule . Die SequenceBlockExpression $[x; y; z]$ wird entfernt und die darin enthaltenen Expressions x, y und z werden der Reihenfolge nach den aktiven Anweisungen hinzugefügt.	30
4.9 Anwendungsbeispiel der PrimitiveValueRule . Von Knoten 1 auf 2 wurde die Pfadbedingung nicht verändert. Bei Knoten 3 wird die PrimitiveValueExpression false erkannt und der Knoten wird als unerfüllbar gewertet.	31
4.10 Ablauf der SSA-Transformation als Flowchart	32
4.11 Beispielhafte Anwendung der AssignmentRule , bei der die Variable c uninitialisiert ist.	34
4.12 Anwendung der AssignmentDivisionRule . Knoten 2 zeigt den Fall, dass der Divisor ungleich 0 ist. In Knoten 3 wird der Divisor als 0 gewertet. . . .	34
4.13 Beispielhafte Anwendung der DefaultStatementRule , bei der eine LogicalRelationExpression verarbeitet wird.	35
 5.1 Statechart mit mehreren Ein- und Ausgängen des Zyklus $D \rightarrow E \rightarrow D$. . .	38
5.2 Beispielhafte Anwendung der TransitionCycleEliminationRule an einem Statechart. Knoten 10 ist nach zweimaliger Durchführung der Transition $A \rightarrow B$ entstanden. Nach Anwendung der Regel wird für die, durch die Transition veränderbare, Variable a eine neue symbolische Repräsentation erzeugt. Bei Anwendung der TransitionRule in Knoten 11 und 12 entsteht durch die blockierte Reaktion kein Kindknoten für den anderen Zustand des Zyklus. Knoten 15 und 16 sind durch die Anwendung der Regel beide plausibel und der SET ist nicht unendlich.	41

5.3 Beispielhafte Anwendung der LocalReactionCycleEliminationRule an einem Statechart. Knoten 9 ist aus der zweimaligen Durchführung der Reaktion r_1 entstanden. Nach der Anwendung der Regel wird für die in r_1 zugewiesene Variable a eine neue symbolische Repräsentation erstellt. Bei Anwendung der TransitionRule in Knoten 10 entsteht durch die blockierte Reaktion kein Kindknoten für r_1 . Knoten 13 und 14 sind durch die Anwendung der Regel beide plausibel und der SET ist nicht unendlich.	43
6.1 Ablauf der Erfüllbarkeitsbestimmung als Flowchart	46
6.2 Ablauf des Kontext-Solvings als Flowchart	48
6.3 Ablauf der Erfüllbarkeitsbestimmung mittels jConstraints als Flowchart . .	54
6.4 Ablauf der Verbreitung der Erfüllbarkeit als Flowchart	55
6.5 Beispielhafte Verbreitung eines Erfüllbarkeitsergebnisses im SET. Knoten 5 ist hierbei der Startknoten der Verbreitung. Zu beachten ist, dass die Erfüllbarkeit von Knoten 4 am Ende immer noch unbekannt ist.	56
7.1 Beispielhafte Wurzel des SET. Die Variablen a und x werden als aktive Anweisung hinzugefügt.	57
7.2 Ablauf der Konstruktion des SET als Flowchart	58
8.1 Ablauf der Erreichbarkeitsanalyse als Flowchart	64
9.1 Beispielhaftes Statechart an dem die Korrektheit evaluiert wurde.	68
9.2 Skalierbares Statechart in Abhängigkeit des Parameters n	69
9.3 Graphische Visualisierung der default Strategie in Sekunden (siehe Tabelle 9.1)	70
9.4 Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich der Transitionszyklenelimination (siehe Tabelle 9.2)	72
9.5 Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich der Transitionszyklenelimination (siehe Tabelle 9.2)	72
9.6 Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich der Unabhängigkeitsoptimierung (siehe Tabelle 9.3)	73
9.7 Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich der Unabhängigkeitsoptimierung (siehe Tabelle 9.3)	73
9.8 Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich des Kontext-Solving (siehe Tabelle 9.4)	75
9.9 Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich des Kontext-Solving (siehe Tabelle 9.4)	75
9.10 Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich der binary-search Ergebnisverbreitung (siehe Tabelle 9.5)	76

9.11	Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich der binary-search Ergebnisverbreitung (siehe Tabelle 9.5)	76
9.12	Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich der DFS-Konstruktion (siehe Tabelle 9.6)	77
9.13	Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich der DFS-Konstruktion (siehe Tabelle 9.6)	77
9.14	Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich mit jConstraints (siehe Tabelle 9.7)	78
9.15	Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich mit jConstraints (siehe Tabelle 9.7)	78
9.16	Graphische Visualisierung der Solving-Zeit des Performanz-Vergleich der Kostenfunktion (siehe Tabelle 9.8)	80
9.17	Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich der Kostenfunktion (siehe Tabelle 9.8)	80
9.18	Graphische Visualisierung der Gesamt-Zeit des Performanz-Vergleich zum ursprünglichen Algorithmus (siehe Tabelle 9.10)	82
A.1	Grammatik der Statechart Elemente in Yakindu ScT Teil 1	88
A.2	Grammatik der Statechart Elemente in Yakindu ScT Teil 2	89
A.3	Grammatik der Statechart Elemente in Yakindu ScT Teil 3	90
A.4	Expression Grammatik in Yakindu ScT Teil 1	91
A.5	Expression Grammatik in Yakindu ScT Teil 2	92

Literaturverzeichnis

- [1] *jConstraints-z3*. <https://github.com/tudo-aqua/jconstraints-z3>. visited on 2020-06-04.
- [2] ALBERT, ELVIRA, PURI ARENAS, MIGUEL GÓMEZ-ZAMALLOA und JOSÉ MIGUEL ROJAS: *Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency*, Seiten 263–309. 06 2014.
- [3] BARRETT, CLARK, PASCAL FONTAINE und CESARE TINELLI: *The SMT-LIB Standard: Version 2.6*. Technischer Bericht, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.
- [4] BARRETT, CLARK, DANIEL KROENING und THOMAS MELHAM: *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories*. Knowledge Transfer Report, Technical Report 3. London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering, Juni 2014.
- [5] BRAUN, MATTHIAS, SEBASTIAN BUCHWALD, SEBASTIAN HACK, ROLAND LEISSA, CHRISTOPH MALLON und ANDREAS ZWINKAU: *Simple and Efficient Construction of Static Single Assignment Form*. In: JHALA, RANJIT und KOEN DE BOSSCHERE (Herausgeber): *Compiler Construction*, Seiten 102–122, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [6] CHEN, TING, XIAO-SONG ZHANG, SHI-ZE GUO, HONG-YUAN LI und YUE WU: *State of the art: Dynamic symbolic execution for automated test generation*. Future Generation Computer Systems, 29:1758–1773, 09 2013.
- [7] CHRIST, JÜRGEN, JOCHEN HOENICKE und ALEXANDER NUTZ: *SMTInterpol: An Interpolating SMT Solver*. In: DONALDSON, ALASTAIR F. und DAVID PARKER (Herausgeber): *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, Band 7385 der Reihe *Lecture Notes in Computer Science*, Seiten 248–254. Springer, 2012.

- [8] COOK, STEVE, CONRAD BOCK, PETE RIVETT, TOM RUTT, ED SEIDEWITZ, BRAN SELIC und DOUG TOLBERT: *Unified Modeling Language (UML) Version 2.5.1*. Standard, Object Management Group (OMG), Dezember 2017.
- [9] CRISTIAN CADAR, DANIEL DUNBAR und DAWSON ENGLER: *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. Band 8, Seiten 209–224, 01 2008.
- [10] CRISTIAN CADAR, VIJAY GANESH, PETER M. PAWLOWSKI, DAVID L. DILL und DAWSON R. ENGLER: *EXE: Automatically Generating Inputs of Death*. In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, Seiten 322–335. ACM, 2006.
- [11] DOMINIC STARZINSKI: *Statische Analyse auf Statecharts in YAKINDU SCT*, 2019. Masterarbeit.
- [12] EBBINGHAUS, HEINZ-DIETER, JÖRG FLUM und WOLFGANG THOMAS: *Der Satz von Löwenheim und Skolem und der Endlichkeitssatz*, Seiten 91–103. Springer Berlin Heidelberg, Berlin, Heidelberg, 2018.
- [13] GABOW, HAROLD N.: *Path-based depth-first search for strong and biconnected components*. Information Processing Letters, 74(3):107 – 114, 2000.
- [14] GODEFROID, PATRICE, NILS KLARLUND und KOUSHIK SEN: *DART: Directed Automated Random Testing*. SIGPLAN Not., 40(6):213–223, Juni 2005.
- [15] HAREL, DAVID: *Statecharts: A Visual Formalism for Complex Systems*. Sci. Comput. Program., 8(3):231–274, Juni 1987.
- [16] HOWAR, FALK, FADI JABBOUR und MALTE MUES: *JConstraints: A Library for Working with Logic Expressions in Java*, Seiten 310–325. Springer International Publishing, Cham, 2019.
- [17] ITEMIS AG: *Yakindu SCT documentation*. <https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide>. visited on 2019-10-21.
- [18] KLAUS HAVELUND und THOMAS PRESSBURGER: *Model checking JAVA programs using JAVA PathFinder*. International Journal on Software Tools for Technology Transfer, 2(4):366–381, Mar 2000.
- [19] LEONARDO DE MOURA und NIKOLAJ BJØRNER: *Z3: An Efficient SMT Solver*. In: C.R. RAMAKRISHNAN und JAKOB REHOF (Herausgeber): *Tools and Algorithms for the Construction and Analysis of Systems*, Seiten 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

- [20] MICHAIL, DIMITRIOS, JORIS KINABLE, BARAK NAVEH und JOHN V SICHI: *JGraphT-A Java library for graph data structures and algorithms*. arXiv preprint arXiv:1904.08355, 2019.
- [21] ROBERTO BALDONI, EMILIO COPPA, DANIELE CONO D'ELIA, CAMIL DEMETRESCU und IRENE FINOCCHI: *A Survey of Symbolic Execution Techniques*. ACM Comput. Surv., 51(3), 2018.
- [22] TURAU, VOLKER und CHRISTOPH WEYER: *Algorithmische Graphentheorie*. De Gruyter, Berlin, Boston, 2015.
- [23] VENGADESWARAN, S. und K. GEETHA: *Symbolic execution — An efficient approach for test case generation*. In: *2013 International Conference on Recent Trends in Information Technology (ICRTIT)*, Seiten 575–581, 2013.
- [24] WOLFGANG AHRENDT, BERNHARD BECKERT, RICHARD BUBEL, REINER HÄHNLE, PETER H. SCHMITT und MATTIAS ULBRICH (Herausgeber): *Deductive Software Verification - The KeY Book - From Theory to Practice*, Band 10001 der Reihe *Lecture Notes in Computer Science*. Springer, 2016.
- [25] XIAOFEI XIE, BIHUAN CHEN, YANG LIU, WEI LE und XIAOHONG LI: *Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis*. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, Seiten 61–72. ACM, 2016.

**Eidesstattliche Versicherung
(Affidavit)**

Wielage, Jonas

Name, Vorname
(Last name, first name)

165754

Matrikelnr.
(Enrollment number)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem folgenden Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present Bachelor's/Master's* thesis with the following title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution.

Titel der Bachelor-/Masterarbeit*:
(Title of the Bachelor's/ Master's* thesis):

Statische Analyse mittels Symbolic Execution von Zustandsautomaten

*Nichtzutreffendes bitte streichen
(Please choose the appropriate)

Bochum, 12.07.2020

Ort, Datum
(Place, date)


Unterschrift
(Signature)

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegender Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:

Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to €50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, section 63, subsection 5 of the North Rhine-Westphalia Higher Education Act (*Hochschulgesetz*).

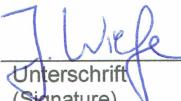
The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:**

Bochum, 12.07.2020

Ort, Datum
(Place, date)


Unterschrift
(Signature)

**Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.

