HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Komfortable, systematische Modellierung und Automatisierung von GUI-Tests

Diplomarbeit

zur Erlangung des akademischen Grades
Diplominformatiker (Dipl.-Inf.)

eingereicht von: Oliver Stadie
geboren am: 14.11.1984
geboren in: Schwedt/Oder
Gutachter/innen: Prof. Dr. Klaus Bothe
Dr. Joachim Wegener
Betreuer/innen: Peter M. Kruse

eingereicht am: ..... verteidigt am: .....

Zusammenfassung

Software als Black Box zu testen, kann viel Zeit kosten und anfällig für Fehler sein. Die *graphische Benutzeroberfläche* (engl. Graphical User Interface, kurz *GUI*) zu bedienen und zu beobachten, ist eine generische Methode, solche Systeme zu testen.

Diese Arbeit behandelt das komfortable und systematische Testen von GUI-Software-Systemen. Sie stellt einen neuen Ansatz zum modellbasierten GUI-Testen vor, indem sie die Stärken aus vier gut erforschten Gebieten kombiniert: (1) die systematische Klassifikationsbaum-Methode, (2) die intuitive Capture/Replay-Methode, (3) Zustandsautomaten und (4) Widget-Trees zur Modellierung der GUI. Der Ansatz wird als Plug-In für das Testwerkzeug TESTONA implementiert und auf einer realen GUI validiert.

Die vorgestellte Methode umfasst den kompletten Zyklus, vom Scannen der GUI über das modellbasierte Spezifizieren der Testfälle bis zur automatischen Ausführung der Tests.

Schlagworte

Automatisches GUI-Testen, Systematisches GUI-Testen, Modellbasiertes Testen, Klassifikationsbaum-Methode, TESTONA, Zustandsautomat, Capture/Replay

Abstract

Testing a software system as a black box can be a time consuming and error prone task. Operating and observing the systems *graphical user interface* (GUI) is a generic way to test these systems.

This work addresses the comfortable and systematical testing of GUI software systems. It proposes a new approach to model-based GUI testing, by combining the strengths of four well researched areas: (1) the systematical Classification Tree Method, (2) the intuitive capture/replay mechanics, (3) state machines and (4) widget trees to model the GUI. The approach shall be implemented as a plug-in for the testing tool TESTONA and validated on a real GUI.

The presented approach includes the whole test cycle, from scanning the GUI and model-based test specification to automatically executing the tests.

Keywords

Automated GUI testing, Systematic GUI testing, Model-based testing, Classification Tree Method, TESTONA, State Machine, Capture and Replay

Inhalt

1. Einleitung

Die folgenden Unterabschnitte bieten eine Einleitung in die Thematik dieser Diplomarbeit.

1.1 Kontext

In der Informatik, speziell in der Softwaretechnik, spielt das Testen von Software eine wichtige Rolle. Es gibt viele Herangehensweisen, wie man Software testen kann.

Heutzutage stellen viele Software-Systeme eine grafische Benutzeroberfläche (engl. Graphical User Interface, kurz GUI) bereit, um den Benutzern den Zugriff zu erleichtern. In aktuellen Systemen macht die GUI bis zu 60 Prozent des gesamten Quelltextes aus (vgl. [Memo02], [BrMe07]). Beim Testen dieser Software-Systeme kann die GUI benutzt werden, um die Software aus dem Blickwinkel des Benutzers zu testen (Black-Box-Testen) oder auch um die GUI selbst zu testen.

1.2 Motivation

Memon [Memo02] macht einerseits auf die Wichtigkeit und anderseits auf den Mangel von GUI-Testmethoden aufmerksam. Verwendete Methoden sind oft unsystematisch, *ad hoc* oder aufwändig. Memon sagt auch, dass die verbreitetste Methode die Benutzung von Capture/Replay-Werkzeugen (dt. Aufnahme und Wiederabspielen) ist. Für zukünftige Entwicklungen schlägt Memon vor, Werkzeugunterstützung, Abstraktion von technologieabhängigen Modellen und Systematisierung von GUI-Testmethoden für bessere Testfälle mit angemessenen Überdeckungskriterien zu entwickeln.

Laut Brooks und Memon [BrMe07] macht bei Software wahrscheinlich allein die GUI 45 bis 60 Prozent des gesamten Quelltextes aus. Testen verursacht im Allgemeinen 50 bis 60 Prozent der Gesamtkosten bei der Softwareentwicklung. Dabei ist das GUI-Testen besonders schwierig, da klassische Testmethoden nicht adäquat sind (vgl. [Memo01]). Durch

automatisierte GUI-Tests könnten bis zu 80 Prozent der Kosten gegenüber manuellen GUI-Tests gespart werden (vgl. [Turp08]).

Ein weiteres der Hauptproblem beim GUI-Testen ist, dass es eine nahezu unendlich große Menge möglicher Nutzereingaben gibt (vgl. [BrMe07], [RuPr07]).

Speziell bei Regressionstests ist ein weiteres Problem, dass Änderungen der GUI keine manuellen Schritte erfordern sollten, um die Tests anzupassen (vgl. [MeSo03]). Andererseits werden von den Testern 60 bis 80 Prozent der Fehler beim Spezifizieren der automatischen Tests gefunden, das heißt nur 20 bis 40 Prozent werden durch die Ausführung der fertigen Tests aufgedeckt (vgl. [Turp08]).

Diese Arbeit stellt – als Lösungsansatz für die genannten Probleme – eine neue modellbasierte GUI-Testmethode und eine prototypische Werkzeugunterstützung vor.

1.3 Stand der Technik/Wissenschaft

Seit der Veröffentlichung von Memon [Memo02] wurde viel im Bereich des GUI-Testens geforscht. Halb- und vollautomatische Methoden wurden veröffentlicht, um den GUI-Testprozess zu vereinfachen (vgl. [QuNa13, Turp08]). Trotzdem ist der am weitesten verbreitete Ansatz noch immer Capture/Replay (vgl. [ABPS12]). Diese Kluft zwischen erforschten und angewendeten Methoden könnte in einem Mangel an Intuitivität, Erlernbarkeit und mangelnder Werkzeugunterstützung begründet sein.[1]

Die verbreiteten Capture/Replay-Werkzeuge funktionieren folgendermaßen: Der Tester nimmt einmalig eine manuell ausgeführte Sequenz von Aktionen auf der GUI auf. Das ist der Capture-Anteil. Danach kann die aufgenommene Sequenz immer wieder automatisch auf der GUI ausgeführt werden. Das ist der Replay-Anteil. Ein Vorteil der Cap-

[1] Für GUI-Tests wurde keine Quelle gefunden welche diese Kluft untersucht. Jedoch beschreiben Causevic et al. [8] sieben Ursachen für eine solche Kluft zwischen erforschter und industriell angewandter Testgetriebener Entwicklung (kurz: TDD). Causevic et al. haben dabei eine Tendenz die „Schuld“ beim Anwender zu suchen anstatt bei den Methoden der TDD. Die identifizierten sieben Punkte sind: Erhöhte Entwicklungszeit, Unzureichende Erfahrung/Kenntnisse im Bereich TDD, Unzureichendes vorhergehendes (Software-) Design, Domänen- und Werkzeug-spezifische Ursachen, Unzureichende Fähigkeiten beim Schreiben von Testfällen Unzureichende Befolgung des TDD Protokolls, Code-Altlasten.

ture/Replay-Werkzeuge ist, dass sie leicht erlernt und benutzt werden können. Ein Nachteil ist, dass sie nicht inhärent systematisch sind und die Qualität der aufgenommenen Tests von den Fähigkeiten des Testers abhängt (vgl. [Memo02]).

Überschaut man die wissenschaftlichen Arbeiten, die sich mit dem modellbasieren GUI-Testen beschäftigen, gibt es zwei dominante Methoden um GUIs zu modellieren. Ein üblicher Ansatz zur Modellierung von GUIs in modellbasierten Testmethoden sind Zustandsautomaten (vgl. [QuNa13]). Ein zweiter Ansatz ist das Modell von Memon et al. [MeBN03a], bestehend aus *GUI forest*, *event-flow graphs* und einem *integration tree*.

Widget-Trees sind ein weiterer Ansatz zur Modellierung von GUI-Zuständen, welcher zum Beispiel von Bauersfeld [BaWW11] verwendet wird. Im Gegensatz zu Zustandsautomaten, fokussieren Widget-Trees sich auf die Modellierung aller Elemente in der Widget-Hierarchie, während Zustandsautomaten das Verhalten und mögliche Navigationspfade durch das System modellieren.

Eine systematische Methode zur Testspezifikation ist die Klassifikationsbaum-Methode [GrGr93]. TESTONA[2] ist ein Testwerkzeug, welches die Klassifikationsbaum-Methode implementiert (vgl. [KrLu10, LeWe00]), jedoch bisher keine spezielle Unterstützung für das Testen von GUIs bereitstellt.[3]

Da GUI-Tests komplex und anspruchsvoll sind, ist es immer noch üblich, gar keine Teststrategien zu erstellen und einfach solange zu testen, bis dafür „keine Zeit mehr ist“ (vgl. [Turp08]).

Nur wenige Methoden für systematisches Black-Box-Testfall-Design werden in der Industrie eingesetzt, obwohl in der Praxis Black-Box-Tests wichtiger sind als White-Box-Tests (vgl. [LeWe00]). Black-Box-Tests basieren auf der funktionalen Spezifikation oder anderen funktionalen Informationen des Testobjekts.

[2] Ehemals unter dem Namen *CTE XL* bekannt.

[3] Während der Entstehung dieser Arbeit ist eine weitere Arbeit von Nasarek [KrNF14, Nasa13] veröffentlicht worden, welche eine erste Unterstützung von Web-GUIs in das Werkzeug TESTONA integriert. Diese Unterstützung ist jedoch noch prototypisch und noch nicht ins finale Software-Produkt integriert.

1.4 Problemstellung

Diese Arbeit hat das Ziel, eine intuitive und leicht erlernbare Methode zur Unterstützung des GUI-Testprozesses zu entwickeln, indem es existierende Methoden miteinander kombiniert. Die entwickelte Methode soll in einem Werkzeug implementiert werden, welches für jede Art von GUI benutzt werden kann, unabhängig von der zugrundeliegenden Technologie.

Ein weiteres Ziel der Arbeit ist es herauszufinden, wie anwendbar die Klassifikationsbaum-Methode für allgemeine GUI-Tests ist. Das Werkzeug TESTONA soll prototypisch für allgemeine GUI-Tests erweitert werden.

Turpin [Turp08] sagt, dass eine Kombination von Testmethoden Fehler besser aufdecken, als jede der kombinierten Methoden einzeln es je könnte. Ähnlich sagt es auch Memon [Memo01]:

> *[…] it is necessary to combine several techniques to test a software, so that weaknesses of one technique do not have too much impact on the overall testing results. Rather, the combined strengths of several testing techniques will result in better testing of the software.*[4]

Zu diesem Zweck sollen verschiedene Methoden und Modelle aus dem Bereich des GUI-Testens zu einer einzigen Methode kombiniert werden. Speziell die folgenden Methoden und Modelle sollen benutzt werden:

- Die Klassifikationsbaum-Methode (engl. Classification Tree Method, kurz CTM),
- der Zustandsautomat (auch endlicher Automat, Zustandsmaschine) mit einigen Erweiterungen und Darstellungsformen des UML Zustandsdiagramms,
- der Widget-Tree und
- die Capture/Replay-Methode.

[4] Eigene Übersetzung: *[…] es ist notwendig verschiedene Techniken zum Softwaretesten zu kombinieren, damit Schwächen einer Technik nicht zu große Auswirkungen auf das Gesamtergebnis haben. Stattdessen resultieren die kombinierten Stärken verschiedener Test-Techniken in besserem Softwaretesten.*

Das zu erarbeitende Werkzeug soll den Benutzer zum komfortablen und systematischen Entwurf von Testszenarien für GUI-Systeme befähigen und es ihm ermöglichen, die entwickelten Testszenarien automatisch auszuführen. Das Ziel der so entwickelten Szenarien könnte es sein, die GUI selbst zu testen oder auch die GUI für einen Black-Box-Test des darunterliegenden Systems zu benutzen.

Die Arbeit soll folgende Fragen beantworten:

- Kann die Klassifikationsbaum-Methode auf GUI-Tests angewendet werden? Welchen Beitrag leistet die Klassifikationsbaum-Methode beim Testentwurfsprozess?
- Können die Klassifikationsbaum-Methode, das Widget-Tree-Modell, der Zustandsautomat und die Capture/Replay-Methode zu einer systematischen und komfortablen Methode für einen allgemeinen GUI-Testprozess kombiniert werden?
  - Welche Vor- und Nachteile trägt jedes Teilstück zum kombinierten Verfahren bei?
  - Wie können die Nachteile gehandhabt werden?
  - Welche Synergieeffekte treten durch die Kombination der Methoden auf? Welche Vorteile der Einzelmethoden werden durch die Kombination mit anderen Methoden unterdrückt?
- Welche Informationen einer konkreten GUI-technologie gehen verloren, wenn zu einem technologieunabhängigen GUI-Modell abstrahiert wird und wie wirkt sich dieser Informationsverlust auf den Testprozess aus?
- Wie wird der GUI-Testprozess durch die entwickelte Methode eingeschränkt?
- Wie kann die erarbeitete Methode als Werkzeug implementiert werden? Welche existierenden Werkzeuge, Frameworks und Bibliotheken werden verwendet und wie werden diese angepasst?

1.5 Abgrenzung

Die folgenden Themen werden nicht im Rahmen dieser Arbeit behandelt.

1.5.1 Testen vor der Implementierung

In einigen Vorgehensmodellen zur Softwareentwicklung werden Tests vor der Implementierung der Software spezifiziert (zum Beispiel in der Testgetriebenen Entwicklung [Beck03]). Die vorgestellte Methode wird nur Modelle aus vorhandenen GUI-Implementationen erzeugen. Das heißt, Testsequenzen können erst erzeugt werden, nachdem das Testobjekt implementiert wurde. Um Testfälle vor der Implementierung definieren zu können, wäre eine Technik nötig, um die verschiedenen Modelle manuell zu erzeugen. Denkbar wäre zum Beispiel eine Spezifikation aus Mock-up. Dieses Thema wird bewusst nicht in dieser Arbeit behandelt.

1.5.2 Regressionstests

Da die definierten Testsequenzen im Rahmen der Arbeit automatisch ausgeführt werden können, ließen sie sich potentiell auch für Regressionstests nutzen. Memon [Memo02] weist auf die Probleme bei Regressionstests von GUIs hin. Bereits kleine Änderungen der GUI können zu großen Änderungen im Widget-Tree und daher – übertragen auf unsere Arbeit – auch im Zustandsautomaten und im Klassifikationsbaum führen. Daher wird ein Mechanismus benötigt, um die Modelle gegenüber Änderungen am GUI zu stabilisieren. Bauersfeld [BaVo12a] schlägt eine Methode vor, um Widget-Trees zu stabilisieren. Die Stabilisations-Methode von Bauersfeld wird auch in dieser Arbeit verwendet. Weitere Probleme, die Regressionstests betreffen, werden im Rahmen dieser Arbeit nicht behandelt. Grechanik et al. [GrXF09] entwickelten einen Ansatz und ein Werkzeug speziell für Regressionstests von GUIs, welche nicht Teil dieser Arbeit sein werden. Auch Memon [MeSo03] behandelt dieses Thema.

1.5.3 Testorakel

Um festzustellen, ob die Ausführung der Testsequenzen erfolgreich war oder nicht, muss ein Testorakel definiert werden, das heißt eine Definition erwarteter Ergebnisse (vgl. [SRWL14]). Dieses Thema wird gar nicht im Rahmen dieser Arbeit behandelt und offen gelassen für weitere Untersuchungen.

Memon et al. [MeBN03a] erklärt richtig, dass auch automatisch generierte Modelle manuell auf ihre Richtigkeit überprüft werden müssen. Immerhin handelt es sich ein Modell, welches aus einem möglicherweise fehlerhaften SUT generiert wurde, jedoch dazu dienen soll die Fehler aufzudecken.

1.5.4 Controllability Problem

Ein Problem beim Generieren von Testfällen ist das *controllability problem,*[5] das heißt die GUI in einen Zustand zu versetzen, in dem die Testfälle ausgeführt werden können (vgl. [Memo01]). Zur Lösung dieses Problem müssen viele technische Details betrachtet werden, wie zum Beispiel Betriebssystem oder die Anbindung des SUTs an das Internet oder sonstige externe Einflussquellen. Das *controllability problem* wird im Rahmen dieser Arbeit nicht weiter behandelt.

1.6 Begriffe und Definitionen

Im Folgenden werden einige Begriffe und Abkürzungen definiert, die in dieser Arbeit verwendet werden.

1.6.1 GUI

Die Abkürzung *GUI* steht für „Graphical User Interface“, also grafische Benutzerschnittstelle oder grafische Benutzeroberfläche. Damit wird genau jener Teil einer Software bezeichnet, welcher auf dem Bildschirm für den Benutzer sichtbar ist. Mit *GUI-System* wird in dieser Arbeit die komplette Software bezeichnet, welche unter anderem eine GUI implementiert.

Turpin [Turp08] definiert GUI folgendermaßen:

> *The GUI is a hierarchical, graphical front-end to a software that receives input from the user by way of keyboards and mice or from system generated input. This input*

[5] auf Deutsch etwa „Beherrschbarkeitsproblem“

is then passed to the program which processes the information; the results are then displayed as graphical output on a GUI.[6]

Memon [MeBN03b] definiert ein Beispiel, wie man ein GUI mathematisch modellieren kann:

We model a GUI as a set of widgets $W = \{w_1, w_2, \ldots, w_n\}$ *(e.g., buttons, panels, text fields) that constitute the GUI, a set of properties* $P = \{p_1, p_2, .., p_m\}$ *(e.g., background color, size, font) of these widgets, and a set of values* $V = \{v_1, v_2 \ldots, v_n\}$ *(e.g., red, bold, 16pt) associated with the properties. Each GUI will contain certain types of widgets with associated properties. At any point during its execution, the GUI can be described in terms of the specific widgets.*[7]

Turpin [Turp08] und Memon et al. [MePS00] definieren weiter:

The actions $A = \{a_1, a_2, \ldots, a_n\}$ *associated with a GUI are functions that modify the GUI from one state to another state. These actions are triggered by events.*[8]

In einer GUI löst der Benutzer Ereignisse – wie Mausbewegungen, Objektmanipulation und Öffnen und Schließen von Fenstern – aus. Die GUI interagiert wiederrum mittels Nachrichten und Methoden-Aufrufen mit dem darunterliegenden Quelltext (vgl. [Memo01]).

[6] Eigene Übersetzung: Die GUI ist ein hierarchisches, grafisches Front-End einer Software, welches Tastatur- oder Mauseingaben des Nutzers oder vom System erzeugte Eingaben empfängt. Diese Eingaben werden dann an das Programm weitergegeben, welches diese Informationen verarbeitet. Die Resultate werden dann als grafische Ausgaben auf der GUI dargestellt.

[7] Eigene Übersetzung: Wir modellieren eine GUI als eine Menge von Widgets $W = \{w_1, w_2, \ldots, w_n\}$ (z.B. Buttons, Panels, Textfelder) welche die GUI ausmachen, einer Menge von Eigenschaften $P = \{p_1, p_2, .., p_m\}$ (z.B. Hintergrundfarbe, Größe, Schriftart) dieser Widgets, und einer Menge von Werten $V = \{v_1, v_2 \ldots, v_n\}$ (z.B. rot, fett, 16pt) zu diesen Eigenschaften. Jede GUI wird bestimmte Arten von Widgets mit ihren assoziierten Eigenschaften enthalten. Zu jedem Ausführungszeitpunkt kann die GUI mittels ihrer spezifischen Widgets beschrieben werden.

[8] Eigene Übersetzung: Die Aktionen $A = \{a_1, a_2, \ldots, a_n\}$ welche mit der GUI assoziiert sind, sind Funktionen welche die GUI von einem Zustand in einen anderen überführen. Diese Aktionen werden durch Ereignisse ausgelöst.

Will man bei GUI-Tests eine hohe Abdeckung erreichen, so zieht man laut Turpin [Turp08] andere Abdeckungskriterien heran als bei Nicht-GUI-Tests. Bei GUI-Tests misst man die Abdeckung an den getesteten Ereignissen, anstatt an den getesteten Quelltextzeilen. Um dies zu untermauern zeigt Memon [Memo01] am Beispiel von WordPad, dass eine Testsuite mit allen Testsequenzen, welche genau ein Ereignis auslösen, bereits 92 Prozent aller Code-Zeilen abdecken. Alle Sequenzen der Länge „zwei“ decken sogar 97 Prozent aller Code-Zeilen ab.

1.6.2 Widget

Ein *Widget* ist ein grafisches Element innerhalb einer GUI. Dabei kann es sich um bedienbare Elemente – wie Buttons oder Menüeinträge – handeln, aber auch um nichtbedienbare Elemente – wie Label oder Tooltips. Widgets haben verschiedene Eigenschaften – wie Farbe, Größe und Schriftart (vgl. Abschnitt 1.6.1). Die Widgets können – je nach Modell[9] – weitere Widgets enthalten.

Analog zur Definition von Memon et al. [MeBN03b] würde ein Widget w durch die Menge der Trippel (w, p_j, v_k) mit $w \in W$, $p_j \in P$ und $v_k \in V$ charakterisiert werden (vgl. Abschnitt 1.6.1). Da die Widgets in dieser Arbeit weitere Widgets enthalten können, werden Widgets w_i stattdessen hierarchisch definiert:

$$w_i := (W_i, K_i)$$

Wobei $W_i \subset W$ die Menge der in w_i enthaltenen Widgets und $K_i \subseteq P \times V$ die Menge der für w_i relevanten Eigenschaft-Wert-Paare ist. Widgets dürfen sich dabei nicht zirkulär enthalten, d.h. ein Widget darf kein Widget enthalten, in dem es (indirekt) selbst wieder enthalten ist.

1.6.3 Modale und nichtmodale Dialoge

Fenster- und Dialog-Widgets können entweder *modal* oder *nichtmodal* sein (vgl. [MeBN03a]). Wird über einem *Haupt-Fenster* ein *modales Fenster* (meist *modaler Dialog*

[9] Das in dieser Arbeit benutzte Modell sind sog. Widget-Trees. Diese erlauben, dass Widgets weitere Widgets enthalten. Widget-Trees werden später im Abschnitt 2.4 genauer definiert.

genannt) angezeigt, so bleibt die Bedienung des Haupt-Fensters bis zum Schließen des modalen Fensters blockiert. Nichtmodale Fenster erlauben wiederum die Bedienung beider Fenster gleichzeitig.[10]

1.6.4 SUT

Die Abkürzung *SUT* steht für *system under test*, also für das zu testende System (auch Prüfling genannt). Da es in dieser Arbeit ausschließlich um GUI-Systeme geht, seien die Begriffe SUT und GUI-System im Kontext dieser Arbeit gleichbedeutend.

1.6.5 UML-Diagramme

Zur Visualisierung einiger Aspekte der Arbeit werden UML[11] Diagramme verwendet. Die Syntax und Semantik der verschiedenen verwendeten Diagrammtypen wird in dieser Arbeit nicht erläutert. Jedoch sind die Diagramme zusammen mit den textuellen Erklärungen weitestgehend auch ohne Vorkenntnisse in UML verständlich. Falls dennoch Bedarf besteht die Details einer Notation zu verstehen, sei auf Rumbaugh et al. [RuJB04] verwiesen.

Eine Ausnahme bilden hier die Zustandsdiagramme der UML. Diese werden später in Abschnitt 2.2.3 genauer erläutert, da sie nicht nur zur Erklärung der entwickelten Methode dienen, sondern auch ein von der Methode erzeugtes Artefakt sind.

Die verwendeten Diagrammtypen sind: Klassendiagramme zur Beschreibung des entwickelten Meta-Modells, Objektdiagramme zur Darstellung von Beispielen (vor allem Widget-Trees), Aktivitätsdiagramme zur Beschreibung des entwickelten Algorithmus, Komponenten-Diagramme zur Beschreibung der Implementationsarchitektur und Zustandsdiagramme zur Darstellung von Beispielen und entstandenen Artefakten.

---

[10] Man kann bei modalen Fenstern weiter unterscheiden zwischen anwendungsmodalen und systemmodalen Fenstern (vgl. http://en.wikibooks.org/wiki/Windows_Programming/Dialog_Boxes). Anwendungsmodale Fenster blockieren den Zugriff auf den Rest der Anwendung, während systemmodale Fenster den Zugriff auf alle anderen Anwendungen blockieren. Im Rahmen dieser Arbeit spielt diese Unterscheidung keine Rolle, da nur eine einzige Anwendung getestet wird.

[11] Unified Modeling Language (dt. Vereinheitlichte Modellierungssprache)

Zum besseren Verständnis werden einige Elemente der Diagramme zusätzlich eingefärbt. Die Bedeutung der Farben wird jeweils bei der ersten Verwendung beschrieben und dann im Rest der Arbeit konsistent weiterverwendet.

1.7 Verwandte Arbeiten

In [MeBN03a] stellen Memon et al. eine Methode vor, um eine GUI während der Ausführung mittels Reverse Engineering auszulesen und ein abstraktes Modell daraus zu erstellen. Das Auslesen funktioniert, indem sich das System automatisch mittels Tiefensuche durch die GUI „hindurchklickt“. Das erzeugte Modell setzt sich zusammen aus einem, in der gleichen Arbeit vorgestellten, *GUI forest*, aus *event-flow graphs* und einem *integration tree*. Ein *GUI forest* ist dabei eine Menge von Bäumen, die einen Baum je möglichem Fenster beim Start des SUTs enthält. Diese Bäume haben das erste Fenster als Wurzelknoten und eine Hierarchie daraus erreichbarer Fenster als Kindknoten. Die vorgestellte Methode ist weitgehend automatisiert, weswegen die Modelle gut für Regressionstests verwendet werden können. Memon et al. merken auch an, dass für andere Arbeiten zum modellbasierten GUI-Testen das Modell oft aufwändig manuell erzeugt werden muss, oder gar nicht erklärt wird, wie man das Modell erhält. Das führt laut Memon et al. dazu, dass viele Anwender eingeschüchtert sind und die erforschten Methoden nicht anwenden. Es deckt sich auch mit unserer Erfahrung beim Lesen der Literatur, dass die Erzeugung der Modelle meistens zu umständlich ist, oder nicht ausreichend beschrieben.

Givens et al. [GCSY13] lesen eine GUI während der Ausführung – mittels Verfahren der Computer Vision – aus und erzeugen einen Zustandsautomaten der internen Zustände des SUTs. Sie nutzen dann *grammatical inference* – eine Methode aus dem Bereich *machine learning* und *natural language processing* – um den Zustandsautomaten zu minimieren. Ziel ist dabei den „internen“ Zustand eines SUT durch Interaktion von „außen“, also über die GUI, zu erkennen. Sie benutzen dabei verschiedene Explorationsstrategien, um den Zustandsraum zu erkunden: zufällige Exploration, die nach einer festen Anzahl von Schritten abbricht und systematische Tiefensuche, welche nur bei deterministischen Anwendungen funktioniert. Aus den, bei der Erkundung durchlaufenen, Pfaden wird ein Baum aufgebaut. Gleichartige Knoten des Baumes werden dann zusammengefasst, um

so den Zustandsautomaten zu konstruieren. Unserer Meinung nach werden die in [19] erzeugten Zustandsautomaten bereits für einfache Beispiele recht groß und unübersichtlich.[12]

Nguyen et al. [NgMT12] kombinieren Zustandsautomaten mit der Klassifikationsbaum-Methode. Sie wählen Pfade auf dem Zustandsautomaten, welche abstrakte Testsequenzen repräsentieren. Zu jedem dieser Pfade konstruieren sie einen Klassifikationsbaum. Mit diesen Bäumen werden für jeden Pfad mehrere konkrete Testsequenzen spezifiziert. Nguyen et al. sehen die Stärke der Zustandsautomaten in der Beschreibung und Auswahl von Sequenzen (aufeinanderfolgenden Ereignissen) und die Stärke der Klassifikationsbaum-Methode in der Auswahl konkreter Eingabeparameter und einer sinnvollen Reduktion der Eingabeparameter. Diese Methode ist vielversprechend, beschreibt jedoch nicht, wie der Zustandsautomat im Detail zu konstruieren ist und gibt nur eine erste Idee wie die Methode auf GUI-Systeme anzuwenden ist. Darauf wird jedoch in Marchetto und Tonella [MaTo10] eingegangen. Sie generieren Tests für AJAX Anwendungen. Sie erzeugen einen Endlichen Zustandsautomaten (engl. *finite-state machine*, FSM), indem sie (vorher definierte) Testsequenzen auf zu testenden Anwendungen ausführen und zusätzlich den Quelltext analysieren. Dabei repräsentiert jeder Zustand des Automaten einen DOM-Baum (Document Object Model) der Anwendung und jede Transition ein Ereignis (Benutzereingaben oder Nachrichten vom Server). Aus dieser FSM berechnen sie eine Menge von *semantisch interagierenden Ereignissen.*[13] Daraus sollen nun mittels metaheuristischer Algorithmen (*hill climbing* und *simulated annealing*) möglichst verschiedene Testsequenzen generiert werden, also Sequenzen, bei denen je zwei aufeinander folgende Ereignisse semantisch miteinander interagieren. Die automatische Ausführung der Testfälle wird nicht behandelt. Die Integration dieser Ansätze in unsere Methode wird in dieser Arbeit nicht weiter verfolgt, ist jedoch eine denkbare Erweiterung für zukünftige Arbeiten.

[12] Für den Windows-Taschenrechner, auf dem nur Arithmetische Operatoren und die Ziffer „5“ gedrückt wurde, umfasst der erzeugte Zustandsautomat bereits 49 Zustände. Die Kantenzahl ist in [GCSY13] nicht angeben und aufgrund der verworrenen Anordnung nur schwer auszuzählen.

[13] Informell interagieren Ereignisse semantisch miteinander, wenn sie voneinander abhängen, das heißt ihre Reihenfolge von Bedeutung ist. Für eine formelle Definition siehe [MaTo10].

Huang et al. [HuCM10] erzeugen automatisch ein Modell des SUTs (in Form eines *event-flow graphs*), indem sie systematisch Widgets im SUT anklicken. Aus diesen leiten sie Testfälle mittels verschiedener Überdeckungskriterien ab. Die verwendeten SUTs sind sehr klein und teils synthetisch. Dadurch ist die automatische Erzeugung eines Modells einfacher als bei großen realen Anwendungen.

Vieira et al. [VLHS06] transformieren eine GUI und deren Spezifikation manuell in ein UML Modell (Anwendungsfall-, Klassen- und Aktivitätsdiagramm) und generieren daraus Testfälle. Sie nutzen die *Category-Partition method* – welche, wie auch die Klassifikationsbaum-Methode, den Eingabedatenraum in Äquivalenzklassen zerlegt – um systematisch ihre Benutzereingaben auszuwählen. Die gewählten Benutzereingaben bestimmen dann den Pfad, das heißt die Testsequenz, in einem Aktivitätsdiagramm. Ähnlich wird in dieser Arbeit die Klassifikationsbaum-Methode genutzt, um Eingabewerte festzulegen und ein Zustandsautomat, um die daraus resultierenden Testsequenzen abzuleiten.

Obwohl andere Modelle zur automatischen Testgenerierung für GUIs denkbar sind, liegt bei verwandten Arbeiten der Fokus auf den zwei konkurrierenden Modellen *event-flow graphs* und *Zustandsautomaten*. Dabei haben wir den Eindruck,[14] dass Zustandsautomaten einige Vorteile gegenüber den *event-flow graphs* haben: (1) sie sind kompakter und damit übersichtlicher, (2) sie sind intuitiv leichter zu verstehen, (3) sie gehören zum Basiswissen für Informatiker und (4) es existiert bereits ein automatisiertes Verfahren für *event-flow graphs*, jedoch noch keines für Zustandsautomaten.[15] Daher fiel die Wahl in der Problemstellung auf Zustandsautomaten anstatt auf *event-flow graphs*.

Man kann Testfälle auf der Grundlage der SUT-Struktur generieren oder auf der Grundlage der üblicherweise benutzten SUT-Aufgaben. Beide Herangehensweisen sind fundamental verschieden und decken unterschiedliche Arten von Fehlern auf (vgl. [Memo01]).

[14] Dies sind keine fundierten Fakten, sondern ein intuitiver Gesamteindruck des Autors. Man findet sicher für jedes Argument verschiedene Gegenbeispiele.

[15] Zumindest konnte im Rahmen dieser Arbeit keine Literatur mit einem solchen Verfahren gefunden werden. Es existieren bereits viele „Einzelteile“, um zum Beispiel GUI-System automatisch als Zustandsautomaten zu modellieren oder um aus einem Zustandsautomaten Testsequenzen zu generieren. Jedoch existiert noch kein Gesamtkonzept mit Zustandsautomaten, dass den Tester über den gesamten Testprozess – vom Capture bis zur automatischen Testausführung – begleitet.

Da in dieser Arbeit eine Methode entwickelt wird, die auf verschiedensten GUIs arbeiten soll, von denen die konkreten Aufgaben nicht definiert sind, werden unsere Testfälle auf der SUT-Struktur basieren. Jedoch können die zugrundeliegenden Modelle auf die konkreten SUT-Aufgaben beschränkt werden.

1.8 Überblick

Dieser Abschnitt gibt einen kurzen Überblick über den Aufbau der Arbeit.

Abschnitt 0 gab eine kurze Einführung in das Gebiet des modellbasierten GUI-Testens und über die Ziele dieser Arbeit.

In Abschnitt 2 werden die vier vorhandenen Konzepte, welche in dieser Arbeit zusammengeführt werden, detailliert erläutert, in Bezug zu GUI-Tests gesetzt und deren Vor- und Nachteile aufgezeigt.

Abschnitt 3 definiert dann ein Zielmodell, welches durch die folgende Lösung anzustreben ist und in der späteren Evaluierung als Soll- und Vergleichswert dient.

Abschnitt 4 ist dann der Kern der Arbeit. Hier wird zuerst definiert, wie das entwickelte Testverfahren aussieht und wie der Tester und verschiedene Modelle darin involviert sind. Danach wird werden die verschiedenen Modelle beschrieben, wie sie vom Verfahren erzeugt werden. Dann wird das Verhalten zur Automatisierung dieses Verfahrens definiert, welches die zuvor beschrieben Modelle erzeugt und zur automatischen Testausführung verwendet.

Abschnitt 0 gibt einen groben Überblick über die Implementation des Prototyps. Hier wird beschrieben, was die Software leistet, welche Technologien verwendet werden, wie die zuvor beschriebenen Modelle und das Verhalten auf verschiedene Module verteilt sind und wie das Ganze ins Testwerkzeug TESTONA integriert wurde.

Abschnitt 6 zeigt die Ergebnisse dieser Arbeit und bewertet diese. Dazu wird sowohl das Konzept (Abschnitt 4) gegen das Zielmodell (Abschnitt 3), als auch der Prototyp (Abschnitt 5) gegen das Konzept (Abschnitt 4) verglichen.

Abschnitt 7 fasst die Arbeit noch einmal abrundend zusammen und bietet Ideen und Ansätze für zukünftige Arbeiten.

2. Theoretische Grundlagen

Memon et al. [MeBN03a, Memo01] sagen: *„GUIs, by their very nature, are hierarchical.“*[16] Diesen Grundsatz versuchen wir zu berücksichtigen und Modelle und Methoden zu finden, welche diese Hierarchie repräsentieren. Hierarchien haben auch den Vorteil, dass sie eine leicht verständliche Methode zur Strukturierung darstellen, also komfortabel und strukturiert sind.

2.1 Klassifikationsbaum-Methode

Ein Weg, Black-Box-Testfall-Design zu systematisieren, ist die *Klassifikationsbaum-Methode (engl. classification tree method, Abk. CTM)*. Sie definiert eine klare Vorgehensweise zum Entwerfen von Testfällen und eine kompakte und klare Repräsentation über die Gesamtheit der entworfenen Tests (vgl. [LeWe00]).

2.1.1 Definition

Die Idee der Klassifikationsbaum-Methode ist es, den Eingabedatenraum (im Fall von GUIs also die Benutzereingaben) – und gegebenenfalls auch den Ausgabedatenraum (im Fall von GUIs also die Bildschirmausgaben) – in bestimmte Gesichtspunkte zu unterteilen und diese systematisch und vollständig in eine endliche Anzahl disjunkter Äquivalenzklassen zu zerlegen (vgl. [LeWe00]). Diese Gesichtspunkte und Äquivalenzklassen notiert man in einem *Klassifikationsbaum*. Die Gesichtspunkte werden *Klassifikationen* genannt. Die Äquivalenzklassen werden *Klassen*, oder bei unklarem Kontext im Folgenden *CTM-Klassen*, genannt.

Aus diesem Klassifikationsbaum kann man Testfälle ableiten, indem man eine bestimmte Kombination von Klassen wählt und einen Repräsentanten aus jeder Klasse als Eingabewert für den Testfall nutzt. Um Testfälle systematisch auszuwählen, kann man Überdeckungskriterien heranziehen. So ist zum Beispiel das *minimality criterion* die Anzahl der

[16] Eigene Übersetzung: GUIs sind ihrer Natur nach hierarchisch.

nötigen Testfälle, um jede Klasse in mindestens einem Testfall zu verwenden. Das *maximality criterion* ist erfüllt, wenn jede mögliche Kombination von Klassen in einem Testfall verwendet wird (vgl. [LeWe00]). Diese Testfälle werden unter dem Klassifikationsbaum in einer Kombinationstabelle, auch *Testmatrix* genannt, notiert.

Als *Klassifikationsbaum-Methode* bezeichnet man den gesamten Vorgang, vom Zerlegen des Eingabedatenraumes, über das Notieren des Klassifikationsbaums bis zum Ableiten von Testfällen.

2.1.2 Beispiel

Die Klassifikationsbaum-Methode soll am folgenden Beispiel von Grochtmann [Groc94] verdeutlicht werden. Das zu testende System ist ein Computer Vision System, welches automatisch verschiedenartige Körper mittels einer Kamera erkennen soll (Abbildung 1).

Dazu definiert man zuerst die zu testenden Klassifikationen des Eingabedatenraums – im Beispiel die Gesichtspunkte der zu erkennenden Körper. Die Körper werden nach den Gesichtspunkten Größe, Farbe und Form untersucht (Abbildung 2).

Diese Klassifikationen zerlegt man dann jeweils in disjunkte Äquivalenzklassen. So kann zum Beispiel der Gesichtspunkt „Form“ in die Klassen „Kreis“, „Dreieck“ und „Viereck“ zerlegt werden. Diese Klassen können rekursiv nach weiteren Gesichtspunkten untersucht werden. So wird im Beispiel die Klasse „Dreieck“ nochmals nach dem Untergesichtspunkt „Form des Dreiecks“ untersucht, in die Klassen „gleichseitig“, „gleichschenklig“ und „ungleichseitig“ zerlegt wird. Die Klassifikationen und Klassen notiert man, wie oben in Abbildung 3 (oben), in Form eines Klassifikationsbaums.

In einer Testmatrix – deren Spalten die Klassen des Klassifikationsbaums repräsentieren – können nun zeilenweise Testfälle definiert werden, indem pro Testfall eine Klasse jeder Klassifikation markiert wird. Im Beispiel (Abbildung 3, unten) wurden drei Testfälle definiert, von denen der dritte Testfall spezifiziert, dass das System mit einem kleinen, blauen, gleichschenkligen Dreieck getestet werden soll.

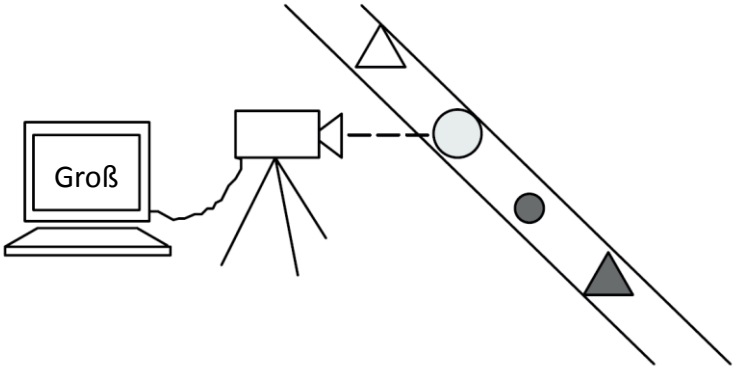


Abbildung 1: Ein Computer Vision System soll automatisch verschiedene Körper erkennen. Nach Grochtmann [Groc94].

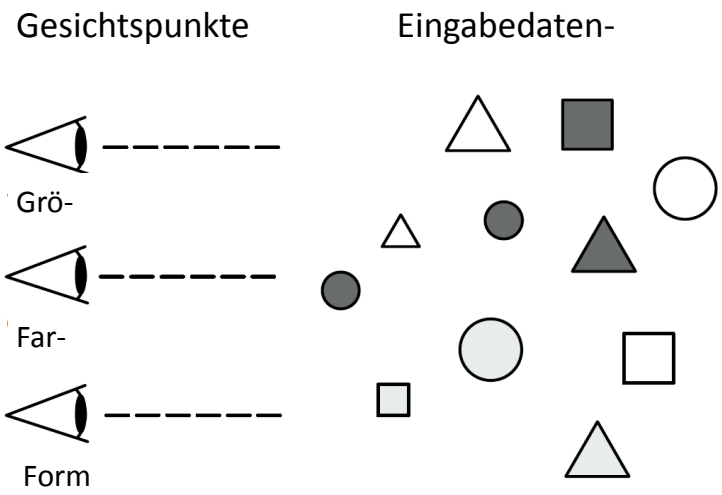


Abbildung 2: Der Eingabedatenraum eines zu testenden Systems wird bei der Klassifikationsbaum-Methode nach verschiedenen Gesichtspunkten, auch Klassifikationen genannt, untersucht. Die Körper im Beispiel werden nach den Gesichtspunkten Größe, Farbe und Form untersucht. Nach Grochtmann [Groc94].

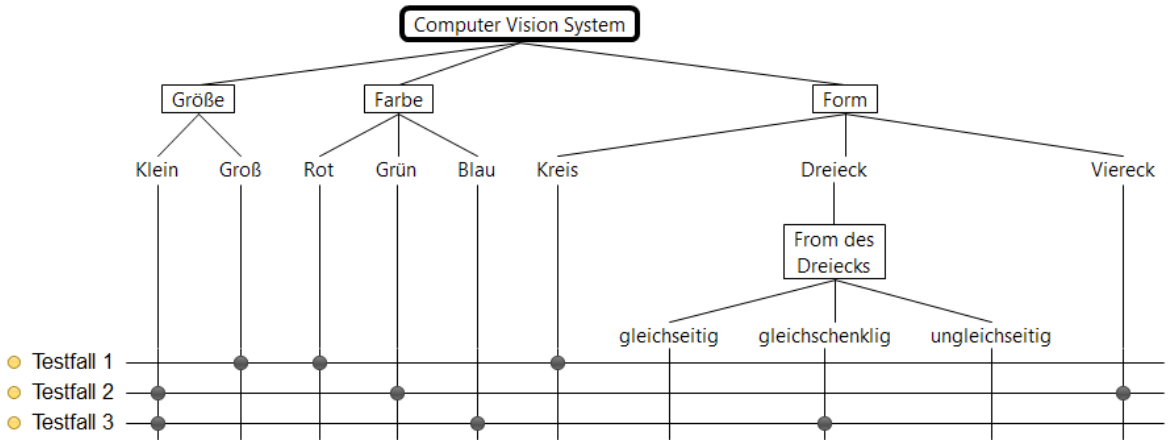


Abbildung 3: Der Eingabedatenraum des Beispielsystems wurde in verschiedene Gesichtspunkte und Klassen zerlegt. Diese werden in einem Klassifikationsbaum dargestellt. Testfälle werden als zeilenweise Markierungen einer Kombinationstabelle, auch Testmatrix genannt, welche mit dem Baum assoziiert ist, spezifiziert. Nach Grochtmann [Groc94].

2.1.3 Erweiterung

Lehmann und Wegener [LeWe00] beschreiben, wie die Klassifikationsbaum-Methode für die Praxis erweitert und als ein Werkzeug implementiert wurde. Zu diesen Erweiterungen gehören:

- Logische Abhängigkeiten zwischen Klassen: Diese machen es möglich, inkonsistente Kombinationen von Klassen aus den möglichen Testfällen auszuschließen. Die Alternative zur Vermeidung dieser Kombinationen wäre es den Baum umzustrukturieren. Dabei würde der Baum jedoch größer und damit unübersichtlicher und schlechter handhabbar werden.
- Automatische Testfallgenerierung: Testfälle werden automatisch, basierend auf dem Klassifikationsbaum und Überdeckungskriterien generiert.
- Wichtigkeit von Klassen und Klassifikationen für den Test: Für einzelne Klassen und Klassifikationen ist definierbar, dass sie wichtiger oder unwichtiger für den Test sind. Dadurch wird beeinfluss wie Testfälle generiert werden.
- Angepasste Überdeckungskriterien: Die Überdeckungskriterien *minimality criterion* und *maximality criterion* wurden angepasst, um die logischen Abhängigkeiten und Wichtigkeit von Klassen und Klassifikationen zu berücksichtigen.

Kruse und Wegener ([Krus11, KrWe12]) beschreiben, wie man zu einem Klassifikationsbaum Testsequenzen, also Folgen von Testschritten, anstatt einfacher Testfälle generieren kann. Hier wird zusätzlich zum Klassifikationsbaum ein zugehöriger Zustandsautomat beschrieben. Dieser definiert, welche Transitionen zwischen den Klassen möglich sind. Eine so generierte Testsequenz für das vorherige Beispiel (Abschnitt 2.1.2) ist in Abbildung 4 dargestellt. Das Zusammenspiel zwischen Zustandsautomat und Klassifikationsbaum und die Generierung der Testsequenzen wird im Verlauf der Arbeit noch detaillierter erläutert.

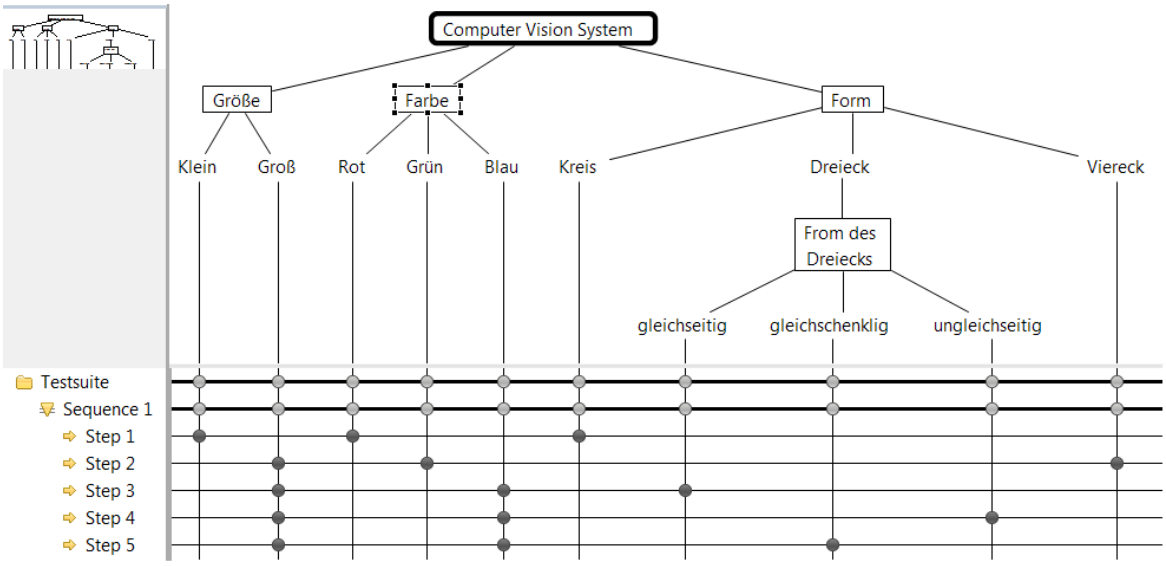


Abbildung 4: Beispiel für eine Testsequenz die zu einem Klassifikationsbaum generiert wurde.

2.1.4 Nachteile

Bei komplexen Systemen kann der Klassifikationsbaum schnell sehr groß werden und damit seinen Vorteil einer guten Visualisierung verlieren. Daher sollte man beim Testfallentwurf *top-down* arbeiten und die Testfälle in mehrere kleinere Klassifikationsbäume aufspalten. (vgl. [BaMc08]).

Das Problem kann auch durch Verbergen von Informationen angegangen werden. Die aktuelle Version des Klassifikationsbaum-Editors TESTONA bietet bereits zwei Mechanismen zum Verbergen von Informationen: Zuklappen von Knoten und Varianten-Management (vgl. [Bern14]).

2.2 Zustandsautomat

Jacob [Jaco83] und Parnas [Parn69] empfehlen Zustandsautomaten für die Beschreibung von GUIs. Zustandsautomaten sollen in den folgenden Unterabsätzen kurz definiert und erläutert werden.

2.2.1 Definition

Ein *Zustandsautomat* (auch *endlicher Automat* oder *Zustandsmaschine*) ist ein häufig verwendetes Modell zur Repräsentation von Verhalten. Der Zustandsautomat wird hier nur kurz definiert.

Eine Art Zustandsautomaten darzustellen ist der Moore-Automat (vgl. [CDDD03]). Dieser kann als ein 7-Tupel $(Q, \Sigma, \Delta, \delta, \mu, q_0, F)$ beschrieben werden, wobei gilt:

- Q ist eine endliche, nicht leere Menge von Zuständen, auch Zustandsmenge genannt.
- Σ ist das Eingabealphabet, in unserem Fall die endliche Menge möglicher Nutzereingaben.
- Δ ist das Ausgabealphabet, in unserem Fall die endliche Menge möglicher Bildschirmausgaben.
- $\delta: Q \times \Sigma \rightarrow Q$ ist die partielle Zustandsübergangsfunktion.
- $\mu: Q \rightarrow \Delta$ ist die Ausgabefunktion.
- $q_0 \in Q$ ist der Anfangszustand.
- $F \subseteq Q$ ist die (möglicherweise leere) Menge von Endzuständen, in unserem Fall die Zustände in denen das SUT beendet wurde.

Ein Zustandsautomat kann auch in Form einer Zustandsübergangstabelle oder in Form eines *Zustandsübergangsdiagramms* dargestellt werden. Im Zustandsübergangsdiagramm werden dabei die Zustände aus Q als Kreise, Zustandsübergangsfunktion δ als Pfeile zwischen diesen Kreisen dargestellt. Die Pfeile werden mit den Elementen des Eingabealphabets Σ beschriftet und die Ausgabefunktion μ wird notiert, indem die Ausgabe (auch Aktion genannt) an der Stelle notiert wird, an der sie Auftritt. Endzustände werden

mit zwei konzentrischen – anstatt einfachen – Kreisen dargestellt. Ein Beispiel eines solchen Diagramms ist in Abbildung 5 zu sehen.

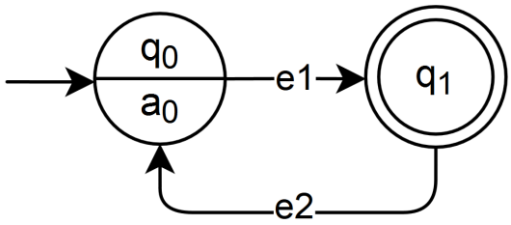


Abbildung 5: Beispiel eines Zustandsübergangsdiagramms, mit $\boldsymbol{q_0}, \boldsymbol{q_1} \in \mathbf{Q}$; $\boldsymbol{e_1}, \boldsymbol{e_2} \in \boldsymbol{\Sigma}$; $\boldsymbol{a_0} \in \boldsymbol{\Delta}$ und $\boldsymbol{q_1} \in \boldsymbol{F}$.

Die Elemente des Funktionsgraphen der Zustandsübergangsfunktion werden nachfolgend *Transitionen* oder *Zustandsübergänge* genannt. Sie überführen den Automaten aus einem *Ausgangszustand* – nicht zu verwechseln mit dem Anfangszustand – in einen *Folgezustand*.

2.2.2 Nachteile und Ansätze

Harris [Harr03] sagt, dass die Komplexität, resultierend aus der Größe des Zustandsraumes, das wichtigste Problem an Zustandsautomaten ist. Er beschreibt eine exponentielle Zunahme der Zustände im Zustandsautomaten für größere Software-Systeme. Dieses Problem bringt zwei Nachteile mit sich. Zum einen kann der Zustandsautomat zu groß werden, um von einem Menschen verwaltet zu werden. Zum anderen kann der Zustandsautomat zu groß und komplex werden, um ihn zu speichern oder Berechnungen in angemessener Zeit auf ihm durchzuführen.

Der Prozess der Testplanung sollte vor dem Entwurf einzelner Testsequenzen stattfinden und das Testobjekt in kleinere zu testende Einheiten einteilen. Diese kleineren Einheiten resultieren in entsprechend kleineren Zustandsautomaten. Diese Einteilung während der Testplanung adressiert sowohl das erste Problem (Speicherbarkeit und Berechenbarkeit), als auch das zweite Problem (Benutzbarkeit durch Menschen) der Zustandsautomaten.

Auch Martin und McClure [MaMc85] sagen, dass klassische Zustandsdiagramme ungeeignet für komplexe Systeme sind, da sie unter anderem schwer zu lesen, nicht benutzerfreundlich und nicht schrittweise verfeinert werden können.

Bergmann und Horowitz [BeHo99] projizieren einen Zustandsautomaten auf eine kleinere Menge interessanter Zustände und erhalten dadurch eine reduzierten Zustandsautomaten. Dabei wird jeder Zustand durch seine Variablenbelegung identifiziert. Durch Vernachlässigen uninteressanter Variablen wird die Menge projizierter Zustände kleiner. Ein vergleichbarer Ansatz wird in dieser Arbeit verfolgt, denn zum Identifizieren eines Zustands werden auch wir instabile Eigenschaften, wie die Bildschirmkoordinaten der Widgets, ignorieren.

Harel und die UML greifen die genannten Nachteile auf und erweitern die Zustandsautomaten, um die Nachteile zu kompensieren. Einige dieser Erweiterungen werden im folgenden Abschnitt 2.2.3 vorgestellt.

2.2.3 Statecharts und UML Zustandsdiagramme

Harel [Hare87] sagt, dass Zustände und Ereignisse *a priori* eine recht natürliche Herangehensweise sind, um das dynamische Verhalten komplexer Systeme zu beschreiben. Dennoch sind, laut Harel, klassische Zustandsautomaten ungeeignet, um komplexe Systeme zu beschreiben, da die Anzahl der Zustände schnell auf ein unüberschaubares Maß wächst.

Harel [Hare87] sagt, dass die Kombination unabhängiger Zustände zu einem größeren Zustand der Hauptgrund für die exponentielle Zunahme von Zuständen ist. Wählt man zum Beispiel einen beliebigen *Basiszustand* x aus einer Menge X mit M Zuständen und kombiniert diesen mit einem beliebigen Basiszustand y aus einer Menge Y mit N Zuständen zu einem *Produktzustand* (x, y) so erhält man bereits $M \cdot N$ solcher Produktzustände, anstatt der $N + M$ Basiszustände.

Harel [Hare87] stellt zur besseren Anwendbarkeit von Zustandsübergangsdiagrammen eine erweiterte Form namens *Statecharts* vor. Dabei handelt es sich um Zustandsübergangsdiagramme, welche um folgende drei Kernaspekte erweitert wurden:

- Hierarchische Zustände: Mehrere Zustände können einem *Überzustand* zugeordnet werden (Bottom-Up) – wodurch eine Abstraktion stattfindet – oder ein Zustand kann in mehrere *Unterzustände* zergliedert werden (Top-down).
- Orthogonale[17] Zustände: Ein Zustand S kann aus mehreren orthogonalen, also nebenläufigen Unterzuständen bestehen. So kann das o.g. Problem der $M \cdot N$ Produktzustände vermieden werden, indem die $M + N$ Basiszustände identifiziert werden.
- Kommunikation: Das Verlassen oder Betreten von Zuständen kann *Aktionen* auslösen. Diese werden wiederrum als *Ereignisse* empfangen und können weitere Zustandswechsel auslösen oder Variablen ändern, welche vor Zustandswechseln geprüft werden. Außerdem stellen die so definierten Aktionen und Ereignisse eine „Schnittstelle zur Außenwelt“ dar. So könnten bestimmte Aktionen als Signale an Hardwarekomponenten definiert werden und bestimmte Benutzereingaben definierte Ereignisse auslösen.

Neben diesen Kernaspekten stellt Harel noch andere Erweiterungen vor, welche im Rahmen dieser Arbeit aber keine Rolle spielen.

Die meisten Erweiterungen aus Harels Statecharts wurden inzwischen in normierter Form ins *Zustandsdiagramm* der UML [RuJB04] aufgenommen und mit genauerer Semantik versehen. Jedoch wurde speziell der Broadcast-Mechanismus – also das globale Senden und Empfangen von Events – von Harel so nicht in die UML übernommen, da die Semantik mehrdeutig ist. Stattdessen müssen Events in UML an eindeutige Empfänger gesendet werden und es muss definiert werden, wann und in welcher Reihenfolge Events verarbeitet werden [Lano09].

Harel [Hare87] sagt, dass Statecharts in der Praxis stets schnell erlernt wurden und hohe Akzeptanz fanden, sogar unter Menschen aus anderen Fachbereichen.

[17] Orthogonal heißt senkrecht. Im Rahmen dieser Arbeit ist es – im Sinne von Harel [Hare87] – als „unabhängig“ oder „nebenläufig“ zu verstehen.

Wir verwenden in dieser Arbeit die durch UML normierte und erweiterte Version der Statecharts, also *UML Zustandsdiagramme*. Die von uns verwendete Syntax und Semantik der *UML Zustandsdiagramme* wird in den folgenden Unterabschnitten erklärt.

2.2.3.1 Notation klassischer Eigenschaften

Bereits die unveränderten Moore-Automaten werden in der UML-Notation etwas anders dargestellt, als in Abschnitt 2.2.1 beschrieben. Abbildung 6 erklärt die UML-konforme Darstellung der grundlegenden Elemente des Zustandsautomaten.

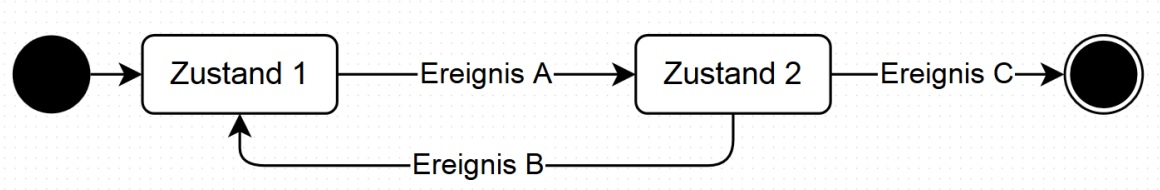


Abbildung 6: Beispiel eines UML Zustandsdiagramms, ohne Harels Erweiterungen. Ein Startzustand ist als schwarzer Kreis (links) dargestellt, ein Endzustand als konzentrischer Kreis (rechts), sonstige Zustände als Rechtecke mit abgerundeten Ecken und Transitionen weiterhin als Pfeile. Die Transitionen sind mit der Eingabe beschriftet, welche den Zustandsübergang auslöst (in UML *Ereignis* genannt). Bei Start- und Endzustand handelt es sich um sogenannte Pseudo-Zustände, in welchen sich der Automat nicht befinden kann. Der Folgezustand des Startzustands (im Bild ***Zustand 1***) ist der Anfangszustand $\boldsymbol{q_0}$ im Sinne unserer Definition aus Abschnitt 2.2.1. Die Vorgängerzustände eines UML Endzustands (im Bild ***Zustand 2***) sind die Endzustände $\boldsymbol{F}$ im Sinne unserer Definition aus Abschnitt 2.2.1, welch jedoch zusätzlich ein beendendes Ereignis erfordern können (im Bild ***Ereignis C***). Die Darstellung der meisten UML-Elemente ist optional, so werden wir zum Beispiel die Ausgabefunktion $\boldsymbol{\mu}$ in dieser Arbeit nicht extra notieren, da ein Zustand stets genau eine Ausgabe repräsentieren wird.

2.2.3.2 Hierarchische Zustände

Zustände können in der UML hierarchisch strukturiert werden. Das heißt, ein Zustand kann Unterzustände enthalten oder einen Überzustand haben.

Abbildung 7 zeigt ein Beispiel, wie ein solcher Zustandsauttomat mit Hierarchien in UML dargestellt wird. Die Semantik ist hierbei die eines XOR-Operators (exklusives ODER). Befindet sich der dargestellte Zustandsautomat beispielsweise in *Zustand A*, so bedeutet dies, dass er sich in *A1 XOR A2* befindet, also *entweder in Zustand A1 oder in Zustand A2*.

Umgekehrt befindet sich der Zustandsautomat immer auch in einem Überzustand, wenn er sich in einem der Unterzustände befindet. Befindet der Automat sich beispielsweise in *Zustand B2*, dann befindet er sich damit auch in *Zustand B*. [18]

Verlässt eine Transition einen Überzustand (im Beispiel *Ereignis 1*), so ist dies gleichbedeutend mit einer Transition, die jeden Unterzustand verlässt und im gleichen Folgezustand endet (Beispielsweise eine Transition die *A1* verlässt und eine Transition die *A2* verlässt).

Endet eine Transition in einem Überzustand, so wird als Unterzustand der Anfangszustand des Überzustands betreten, im Beispiel *Zustand B1*.

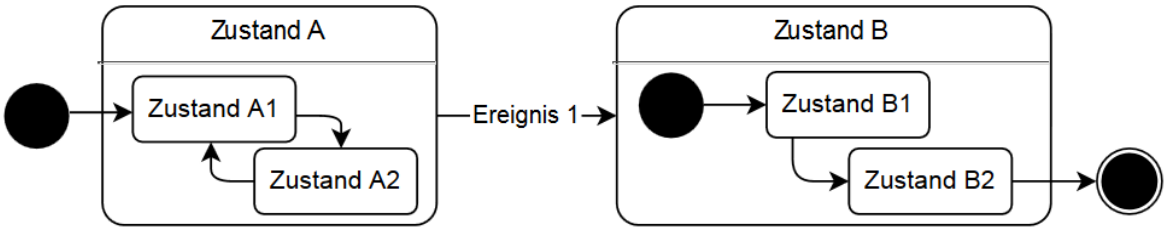


Abbildung 7: Zustände können in der UML hierarchisch strukturiert werden, das heißt Über- und Unterzustände haben. Stellt man die Unterzustände eines Überzustands dar, so wird das abgerundete Rechteck des Überzustands in zwei Abschnitte unterteilt. Oben steht der Name des Überzustands, unten werden die Unterzustände dargestellt.

2.2.3.3 Orthogonale Bereiche

Man kann einen Zustand in *orthogonale Bereiche* (auch *Regionen* genannt) untergliedern. Ein Beispiel dafür ist in Abbildung 8 dargestellt. Die Semantik der orthogonalen Unterteilung ist die des UND-Operators. Befindet man sich beispielsweise in *Zustand A*, so befindet man sich in *Bereich M* und in *Bereich N*, welche selbst wieder Unterzustände mittels XOR zusammenfassen. Oder etwas formeller formuliert:

$$Zustand\ A \Leftrightarrow Bereich\ M\ UND\ Bereich\ M\ \Leftrightarrow\ (M1\ XOR\ M2)\ UND\ (N1\ XOR\ N1)$$

[18] Für mehr als zwei Unterzustände funktioniert der n-äre XOR-Operator nur dann wie beschrieben, wenn er genau dann wahr ist, wenn *genau einer* der Parameter wahr ist. Würde man stattdessen die – auf dem Assoziativ-Gesetz basierende – XOR-Definition nutzen, wäre nur eine *ungerade Anzahl* wahrer Parameter nötig. Ein Überzustand A könnte sich also beispielsweise gleichzeitig in drei Unterzuständen A1, A2 und A3 befinden, was nicht in Harels Sinn ist. Im folgenden wird XOR im Sinne der ersten Definition verwendet.

Zusätzlich bietet die UML die Möglichkeit diese Bereiche miteinander kommunizieren zu lassen. Dazu können Transitionen zusätzlich zu den Ereignissen auch mit Aktionen beschriftet werden. Diese Aktionen werden beim Durchlaufen der Transition ausgelöst und können dann, als Ereignisse, weitere Transitionen auslösen. Tritt zum Beispiel im Zustand $(M1, N1)$ $Ereignis\ E$ ein, so wird beim Zustandsübergang die $Aktion\ A$ ausgelöst. $Aktion\ A$ fungiert dann wiederum als Ereignis, welches eine zweite Transition auslöst, so dass wir letzlich in Zustand $(M2, N2)$ enden. Die Reihenfolge der Zwischenschritte, ob es beispielsweise einen Zwischenzustand $(M2, N1)$ oder $(M1, N2)$ oder keinen Zwischenzustand gibt, ist in der UML nicht genau definiert, spielt im Rahmen dieser Arbeit aber auch keine Rolle.

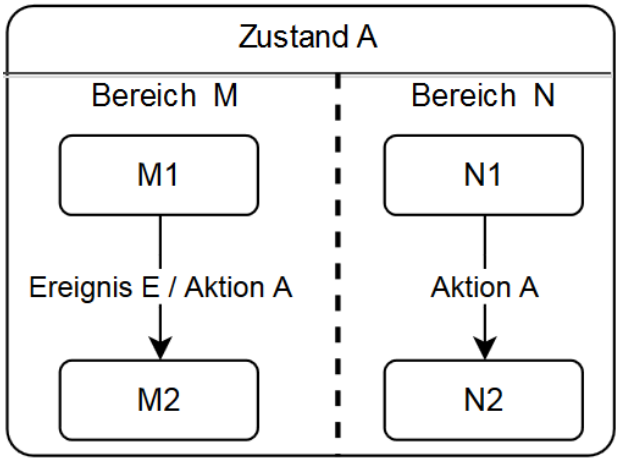


Abbildung 8: Zustände können in orthogonale Bereiche aufgespaltet werden. Diese werden durch gestrichelte Linien voneinander abgegrenzt und können mit einem Namen versehen werden. Transitionen können beim Zustandsübergang Aktionen auslösen („Aktion A“ links), welche von anderen als Ereignis empfangen werden können („Aktion A“ rechts). So ausgelöste Aktionen werden hinter einem Schrägstrich nach dem Ereignis notiert.

2.2.4 GUI-Testen mittels Zustandsautomaten

Zustandsautomaten modellieren in der bisherigen Form nicht zwangsweise GUI-Systeme, sondern beliebige zustandsbasierte Systeme. In der Definition der Moore-Automaten (Abschnitt 2.2.1) wurden bereits einige Zusammenhänge zwischen Zustandsautomaten und GUI-Systemen gegeben. Dieser Abschnitt erklärt genauer, welche Verbindung es zwischen Zustandsautomaten und GUI-Systemen gibt.

Memon et al. [MeBN03a] definieren Zustände auf einer GUI folgendermaßen, basierend auf der Definition für GUIs aus Abschnitt 1.6.1 (Seite 15): Sei $W = \{w_1, w_2, \dots, w_l\}$ die Menge der Widgets, $P = \{p_1, p_2, \dots, p_m\}$ die Menge der Eigenschaften und $V = \{v_1, v_2, \dots, v_n\}$ die Menge der Werte. Dann ist der Zustand eines Fensters zu einer bestimmten Zeit die Menge der Trippel $\{(w_i, p_j, v_k)\}$.

Ein Beispiel wie so ein Zustand aussehen könnte ist in Abbildung 9 zu sehen.

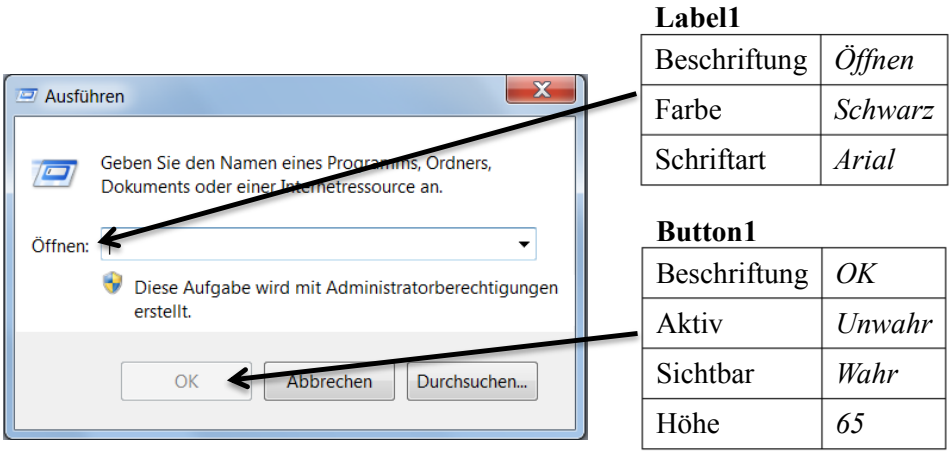


Abbildung 9: Ein Zustand nach Memon et al. [MeBN03a] für die beispielhaft dargestellte GUI ist: *Zustand = {(Label1, Beschriftung, „Öffnen:“), (Label1, Farbe, Schwarz), (Label1, Schriftart, Arial), …, (Button1, Beschriftung, „OK“), (Button1, Aktiv, Unwahr), (Button1, Sichtbar, Wahr), (Button1, Höhe, 65), …}.*

Während Memons Zustände alle Widgets und Eigenschaften als gleichwertig betrachten, erweitern wir dieses Modell noch um eine hierarchische Komponente. So können unsere Widgets weitere Kind-Widgets enthalten und Widget-Eigenschaften sind Elemente der Widgets, anstatt Elemente des Zustands (vgl. Abschnitt 1.6.2). Details dieser Hierarchie sind in der (späteren) Definition von Widget-Trees (Abschnitt 2.4) erläutert. Anders als bei Memon werden unsere Zustände daher als die Menge der Widgets auf oberster Hierarchieebene definiert. Beispielweise wäre der Zustand aus Abbildung 9 definiert als $S = \{Ausführen Fenster\}$.

Durch die Spezialisierung auf GUIs ist zur Definition aus Abschnitt 2.2.1 ist zu ergänzen, dass es in dieser Arbeit für jede unterschiedliche sichtbare Bildschirmausgabe genau einen Zustand geben wird. Unsere Ausgabefunktion $\mu: Q \to \Delta$ ist also bijektiv.

2.3 Capture/Replay

Capture/Replay-Werkzeuge sind Testwerkzeuge mit folgender Idee: Man führt manuell einen Test durch und das Werkzeug nimmt die durchgeführte Sequenz auf. Die so aufgenommenen Sequenzen von Benutzereingaben können später dann später automatisch wieder abgespielt werden. Die existierenden Werkzeuge variieren dabei in der Art, wie Nutzereingaben gespeichert werden – einige speichern die Pixelpositionen jedes Mausklicks, andere Speichern die angeklickten Widgets. Beide Varianten sind nicht besonders stabil gegenüber Änderungen des SUTs (vgl. [Mari97]).

Die Nutzung von Capture/Replay-Werkzeugen ist kontrovers, denn das Aufnehmen und Pflegen verursacht viel Arbeit für die Tester. Speziell bei sich häufig ändernden SUTs entstehen hohe Kosten für die Testpflege mit Capture/Replay-Werkzeugen (vgl. [BaVo12b],[Memo07]).

Janetschek und Tegos [JaTe07] stellen beispielsweise ein Capture/Replay-Werkzeug für Java-Programme vor. Der Nutzer kann Eingabesequenzen auf einer SWT-Benutzeroberfläche aufnehmen. Diese Testsequenzen werden als folgen von Interkationen mit Widgets gespeichert und sind danach automatisch ausführbar.

Memon et al. [MeBN03a] nutzen ein Capture-ähnliches Verfahren, damit der Tester komfortabel ein automatisch generiertes Modell manuell ergänzen kann. So muss der Tester unzugängliche Pfade (zum Beispiel wenn eine Passworteingabe nötig ist) manuell ergänzen. Auch in dieser Arbeit wird der Capture-Mechanismus genutzt, um ein Modell des SUTs zu erzeugen, anstatt direkt Testsequenzen aufzunehmen.

2.4 Widget-Trees

In Abschnitt 1.6.1 wurde bereits eine Definition von Memon [MeBN03b] vorgestellt, welche eine GUI als eine Menge von Widgets modelliert, und diesen Widgets wiederrum Mengen von Eigenschaften und Werten zuordnet. Diese Definition kann man zusätzlich noch um das Element der Hierarchie erweitern, indem man Widgets selbst wieder als

Mengen von Widgets ansieht. Um einen Zustand der gesamten GUI hierarchisch zu modellieren, benutzt Bauersfeld [Baue11] eine Struktur, die er *widget tree* nennt. In dieser Arbeit werden wir eine solche Struktur *Widget-Tree* nennen. Ein *Widget-Tree* ist ein Baum, in dem jeder Knoten einem Widget der GUI und all seinen Eigenschaften entspricht, ähnlich wie der DOM-Baum eines HTML-Dokuments. Abbildung 10 zeigt ein Beispiel für einen solchen *Widget-Tree*.

Die Modellierung komplexer Systeme mittels Hierarchien hat, wie auch schon bei den Zustandsautomaten und CTM, den Vorteil, dass sie komfortabler zu verwalten und zu überschauen ist als eine flache Hierarchie.

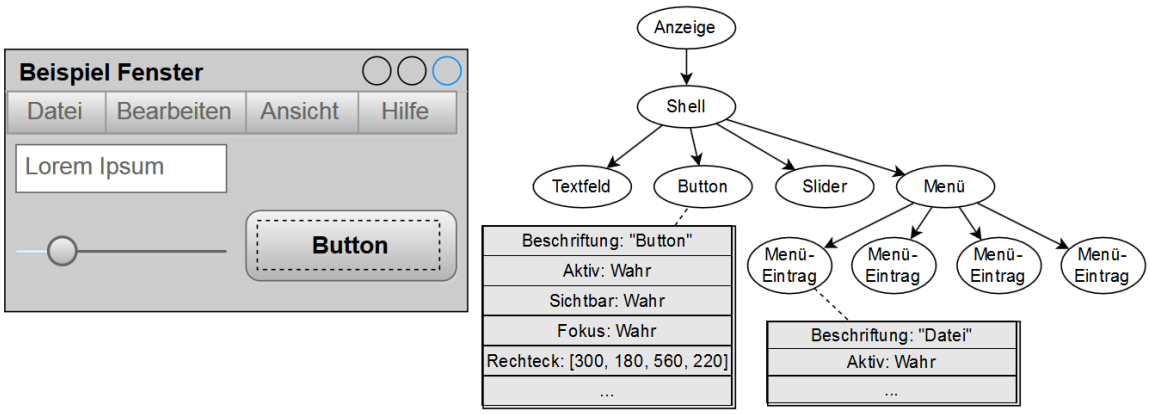


Abbildung 10: Eine einfache GUI-Anwendung und der ***Widget-Tree***, der die Anwendung im gezeigten Zustand modelliert, nach Bauersfeld [Baue11].

Anders als bei Bauersfeld werden Eigenschaften und Werte in dieser Arbeit nicht als Eigenschaft der Knoten modelliert, sondern als eigene Kindknoten. Das heißt, in unseren Widget-Trees existiert nicht nur ein Typ von Knoten, sondern verschiedene Typen: *Widget-Knoten*, *Eigenschafts-Knoten* und *Werte-Knoten*. Im Verlauf der Arbeit wird noch ein zusätzlicher Typ eingeführt werden. Abbildung 11 zeigt das in dieser Arbeit verwendete Widget-Tree-Modell. Abbildung 12 vergleicht beispielhaft einen Widget-Tree nach unserem Modell und dem ursprünglichen von Bauersfeld.

Wie Bauersfeld die Typen von Widgets – ob es sich zum Beispiel um einen Button, ein Label oder ein Fenster handelt – modelliert, ist nicht genau beschrieben. In unserer Arbeit seien sie Unterklassen der Widget-Klasse (vgl. Abbildung 11).

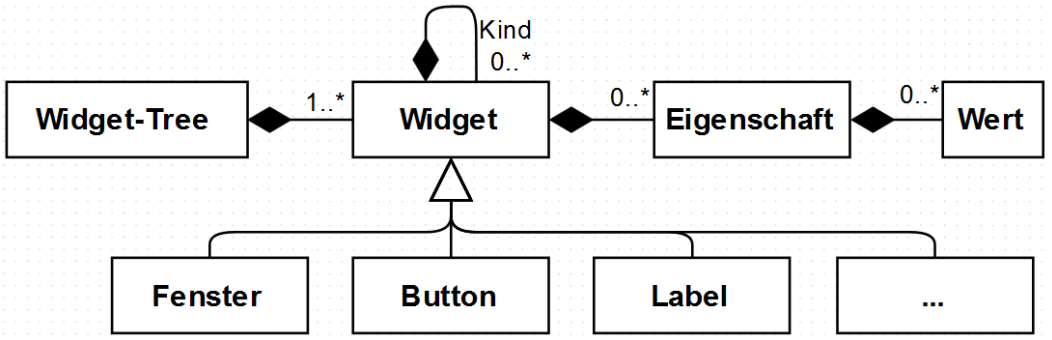


Abbildung 11: Modell der Widget-Trees, die in dieser Arbeit verwendet werden.

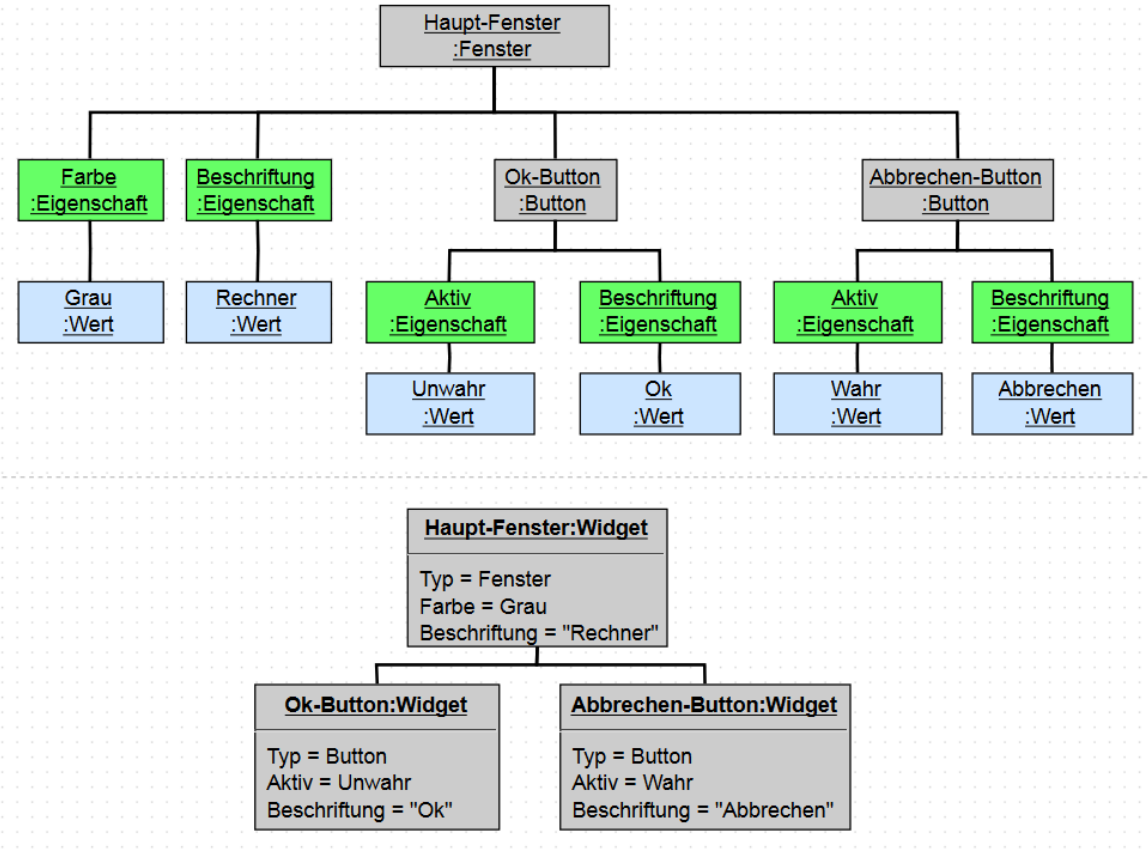


Abbildung 12: Vergleich zwischen unserem veränderten Widget-Tree (oben) und den original Widget-Trees von Bauersfeld (unten). Das Modell von Bauersfeld ist zwar kompakter in der Darstellung, dafür ermöglich unser Modell eine einheitliche Behandlung von Widgets (grau), Eigenschaften (grün) und Werten (blau).

2.4.1 Stabile und Instabile Widget-Trees

Bauersfeld und Vos [BaVo12a] beschreiben auch, dass es einige Eigenschaften gibt, die stabiler sind als andere. So ist beispielsweise die pixelgenaue Bildschirmposition eine instabile Eigenschaft. Sie kann sich bereits durch Kleinigkeiten, wie eine andere Bildschirmauflösung oder das Verschieben von Fenstern ändern. Andererseits ist der Widget-

Typ eine stabile Eigenschaft. Einen *vollständigen Widget-Tree* bezeichnen wir daher auch als *instabilen Widget-Tree*. Um daraus einen *stabilen Widget-Tree* zu konstruieren, reduzieren wir – analog zu Bauersfeld und Vos – den Widget-Tree um alle seine instabilen Eigenschaften. Ein Beispiel ist in Abbildung 13 zu sehen. Im Rahmen dieser Arbeit definieren wir lediglich den Widget-Typ, den Widget-Name und die Beschriftung als stabile Eigenschaften.[19] Auch andere Einteilungen scheinen sinnvoll, werden aber im Rahmen dieser Arbeit nicht weiter betrachtet.

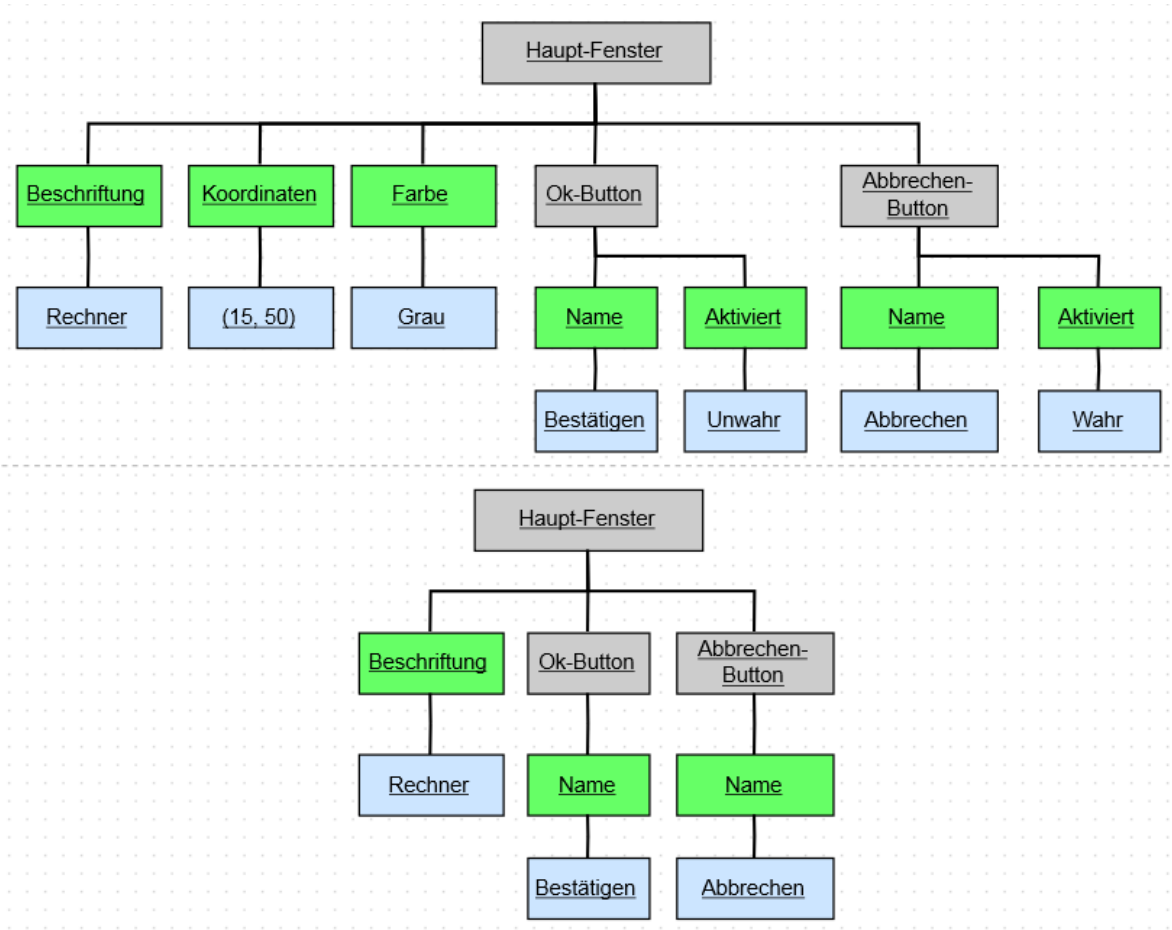


Abbildung 13: Instabile Widget-Trees (oben) sind Widget-Trees mit instabilen Eigenschaften. Reduziert man sie um alle instabilen Eigenschaften (behält also nur die stabilen), erhält man einen stabilen Widget-Tree (unten).

[19] Der Widget-Typ ist in dieser Arbeit – anders als bei Bauersfeld – nicht als Eigenschaft modelliert, sondern als Unterklasse von Widget. Dennoch ist es denkbar, dass er in anderen Modellen als instabil eingestuft werden könnte.

2.5 Überblick der Vor- und Nachteile

In den vorherigen Abschnitten wurden die vier zu kombinierenden Methoden vorgestellt. Jede dieser Methoden hat ihre eigenen Vor- und Nachteile fürs GUI-Testen. Tabelle 1 gibt einen Überblick über die Vor- und Nachteile der Einzelmethoden. Diese Methoden sollen in den folgenden Abschnitten kombiniert werden, mit dem Ziel möglichst viele der Vorteile zu erhalten und möglichst viele der Nachteile zu eliminieren.

Methode	Vorteile / Stäken	Nachteile / Schwächen
Zustandsautomaten	• Modellierung und Auswahl von Systemverhalten (Sequenzen) • Relativ stabil gegenüber Änderungen der GUI • Leicht zu erlernen	• Schlecht manuell handhabbar. Verbesserungen zur Handhabbarkeit vorhanden. • Schwer (voll-) automatisch konstruierbar.
Klassifikationsbaum-Methode	• Systematische Herleitung von Testfällen. • Klassifikation des Eingabedatenraumes / Reduktion der nötigen Testfälle. • Stabil gegenüber Änderungen der GUI • Etabliert in der Praxis • Geeignet für funktionales Black-Box-Testen	• Können für komplexe Systeme zu groß werden. Aufspalten in mehrere Bäume umgeht das Problem.
Capture/Replay	• Intuitive Bedienbarkeit • Weite Verbreitung • Leichte und schnelle Spezifikation einzelner Sequenzen • Leichte Ergänzung eines Modells	• Hoher Wartungsaufwand • Geringe Stabilität gegenüber Änderungen
Widget-Trees	• Detaillierte Modellierung von GUI-Zuständen • Komfortable Verwaltung und Überblick durch Hierarchien	• Stabilität gegenüber Änderungen unsicher

Tabelle 1: Vor- und Nachteile der einzelnen Methoden

3. Zielmodell

Bevor die konkrete Methode zum modellbasierten GUI-Testen entwickelt wird, soll zuerst bestimmt werden, wie ein Modell der GUI idealerweise auszusehen hat. Aussehen und Ermittlung des *Zielmodells* – im Folgenden auch *Idealmodell* oder *Idealwerte* genannt – werden in diesem Abschnitt beschrieben.

3.1 Verwendetes GUI-System

Um ein *Zielmodell* zu ermitteln, wurde der vorinstallierte Taschenrechner von Windows 7 verwendet (vgl. Abbildung 14). Dabei handelt es sich um ein verhältnismäßig einfaches GUI-System, dessen Kernfunktionalität auch ohne genaue Spezifikation nachvollziehbar sein sollte. Der Taschenrechner eignete sich durch seine Einfachheit und bereits vorhandenes Verständnis der Funktionalität gut dazu, die grundlegenden Anforderungen an das GUI-Modell zu ermitteln. Aufgrund seiner Bekanntheit wird der Taschenrechner im Folgenden auch als Fallbeispiel zur Veranschaulichung verwendet.

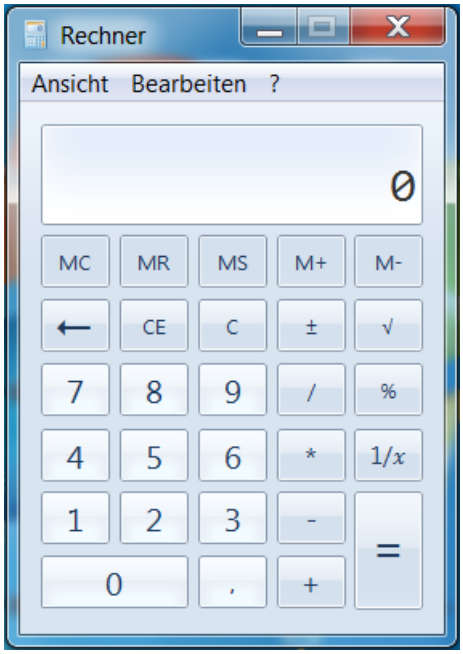


Abbildung 14: Screenshot des Taschenrechners von Windows 7.

3.2 Vorgehen zum Ermitteln des Zielmodells

Um eine Auswahl zu der zu untersuchenden Funktionalität zu definieren, wurde ein Tutorial des Windows-Taschenrechners [Gont13] gewählt. Das Tutorial gibt zum Teil präzise Folgen von zu tätigenden Eingaben und erwarteten Ergebnissen vor, wie in Abbildung 15 zu sehen. An anderen Stellen ist es nur textuell beschrieben und ist mehrdeutig oder lässt Raum für Interpretationen. Für diese Arbeit wurde an diesen Stellen eine feste Folge von Eingaben gewählt und beibehalten, durch welche die beschriebene Funktionalität repräsentativ getestet wird.

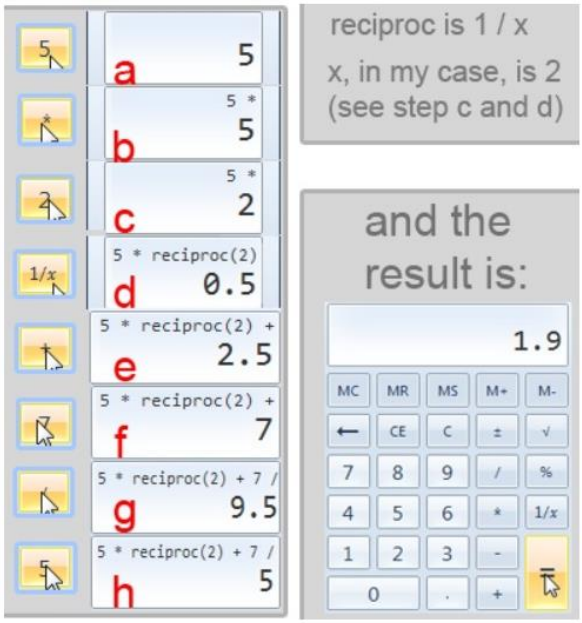


Abbildung 15: Das Tutorial zum Windowstaschenrechner macht teils präzise Vorgaben über auszuführende Eingaben und zu erwartende (Zwischen-)Ergebnisse. Aus Gontarius Tutorial [Gont13].

Es wurde manuell ein Modell entworfen, auf welchem die Schritte und Zustände des Tutorials vollständig abgebildet und durchlaufen werden können. Das Modell wurde frei von den Restriktionen technischer Möglichkeiten entworfen, um ein anzustrebendes Ergebnis vorzugeben. Zusätzlich dient es später dazu, die Qualität des realen Ergebnisses messbar zu machen.

Das so entwickelte *Idealmodell* besteht aus einem orthogonalen, hierarchischen Zustandsautomaten, einem Klassifikationsbaum und Testsequenzen, welche in einer Testmatrix spezifiziert sind. Der Zusammenhang zwischen Zustandsautomat und Klassifikati-

onsbaum ist dabei an Kruse und Wegener [Krus11, KrWe12] angelehnt. Dieser Zusammenhang wird im Verlauf der Arbeit ausführlicher beschrieben.

Der Zustandsautomat des Idealmodells ist dabei in fünf Regionen unterteilt (vgl. Abbildung 16):

- Eine Hauptregion *Main*, in welcher die Kernfunktionalität des Taschenrechners modelliert wird.
- Eine Region *Arbeitsblätter-Ansicht*, in welcher verschiedene Berechnungshilfen – zum Beispiel für Währungen oder Physikalische Einheiten – eingeblendet werden können.
- Die Region *Verlauf-Modus* erlaubt das aktivieren und deaktivieren des Verlaufs.
- Die Region *Ansicht-Menü* spiegelt die Optionen des *Ansicht*-Menüs wieder und
- die Region *Arbeitsblätter-Menü* die zusätzlichen Optionen des Untermenüs *Arbeitsblätter*.

Der Klassifikationsbaum des Idealmodells wird analog in fünf Klassifikationen zergliedert (vgl. Abbildung 17).

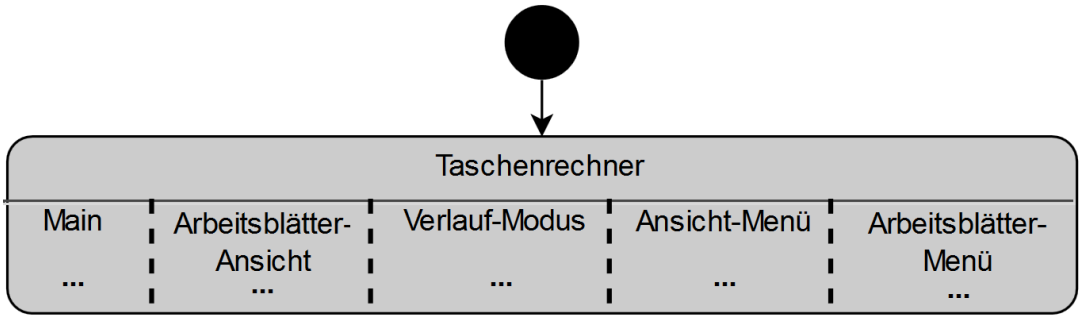


Abbildung 16: Das Idealmodell ist in fünf Regionen zergliedert. Zustände der obersten Hierarchieebene werden grau dargestellt.

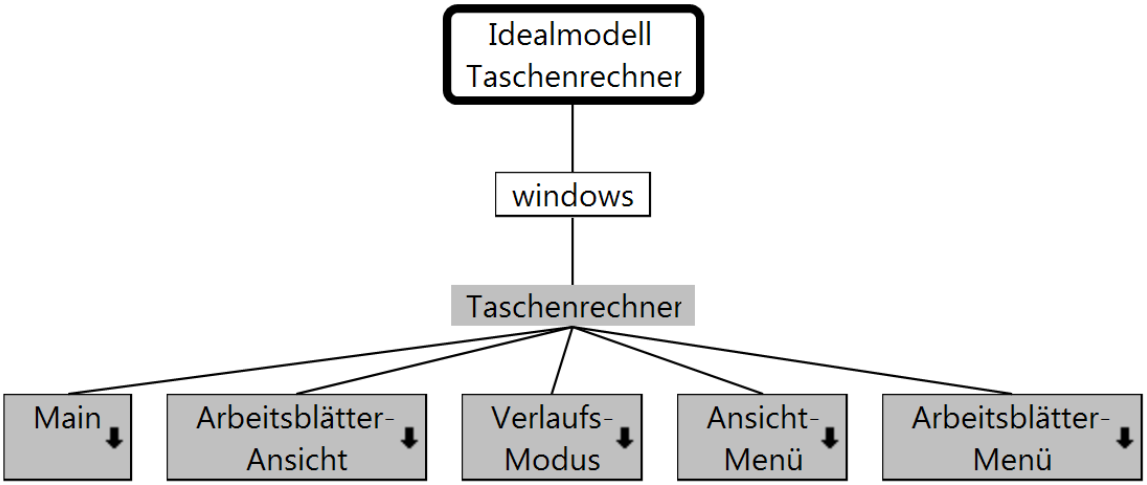


Abbildung 17: Analog zu den Regionen des Zustandsautomaten ist der Klassifikationsbaum des Idealmodells in fünf Klassifikationen zergliedert. Zusätzlich existiert eine Klassifikation ***windows*** (engl. für Fenster), in welcher eine Klasse je modalem Fenster existiert – für das Taschenrechner Tutorial nur ein einziges Fenster. Die Farben im Klassifikationsbaum sind ebenfalls analog zum zu den Zustandsdiagrammen.

Die Hauptregion *Main* wird auf oberster Ebene mit einem Zustand pro Taschenrechner-Modus modelliert und die Ereignisse und Aktionen der Region *Ansicht-Menü* erlauben den Wechsel zwischen diesen Zuständen (vgl. Abbildung 18). Im Klassifikationsbaum des Idealmodells werden analog zu den Unterzuständen Klassen modelliert (vgl. Abbildung 19).

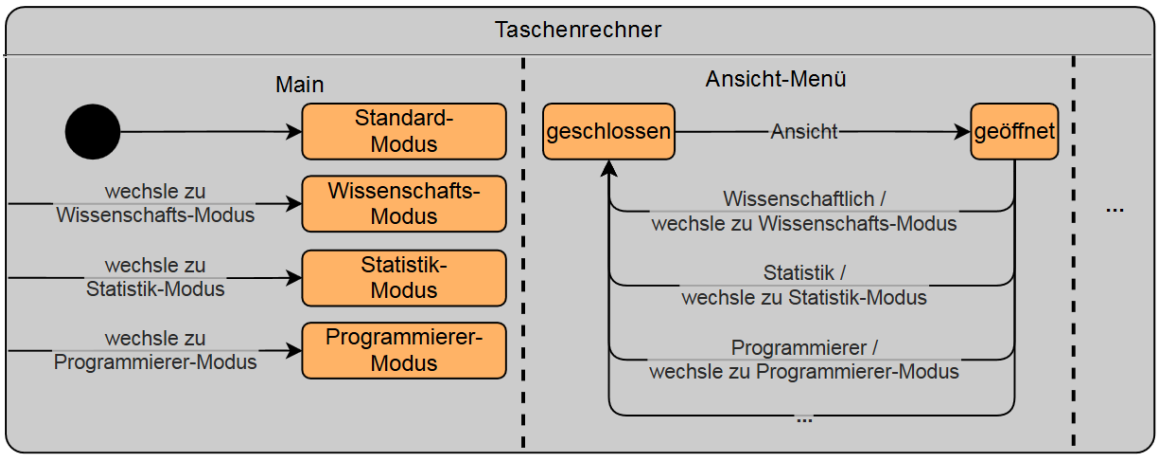


Abbildung 18: Die verschiedenen Taschenrechner-Modi werden in separaten Zuständen modelliert. Die Wechsel zwischen ihnen werden in einer separaten orthogonalen Region ausgelöst. Ein Wechsel zurück zum Standard-Modus wird nicht modelliert, da dieser im Tutorial nicht stattfindet. Zustände der zweiten Hierarchieebene werden orange dargestellt.

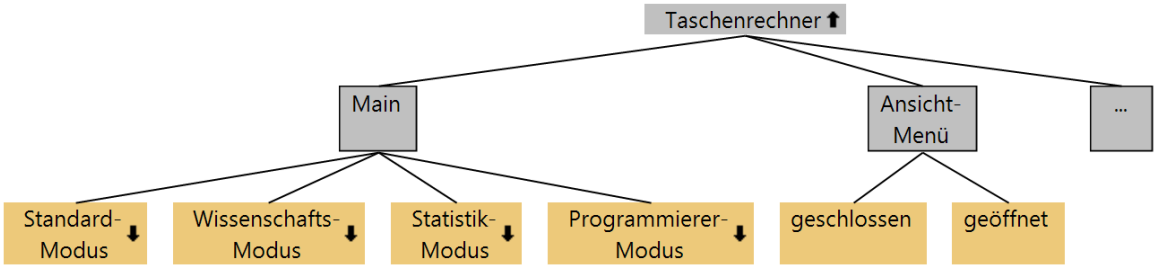


Abbildung 19: Analog zu den Unterklassen des Zustandsautomaten werden Klassifikationsbaums des Idealmodells Klassen modelliert.

Die Kernfunktionalität wird dann innerhalb der Modus-Zustände modelliert (vgl. Abbildung 20). Die Testsequenzen werden, wie im Tutorial beschrieben, nach der Klassifikationsbaum-Methode modelliert (vgl. Abbildung 21).

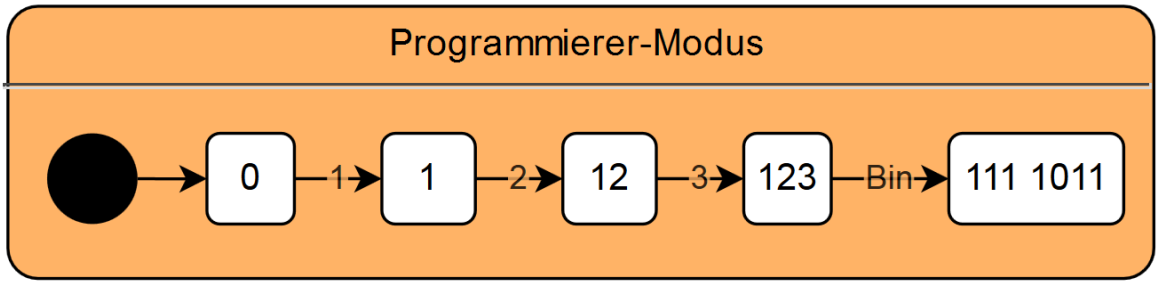


Abbildung 20: Kernfunktionalität wird innerhalb der Modus-Zustände modelliert. Zustände der dritten Hierarchieebene werden weiß dargestellt.

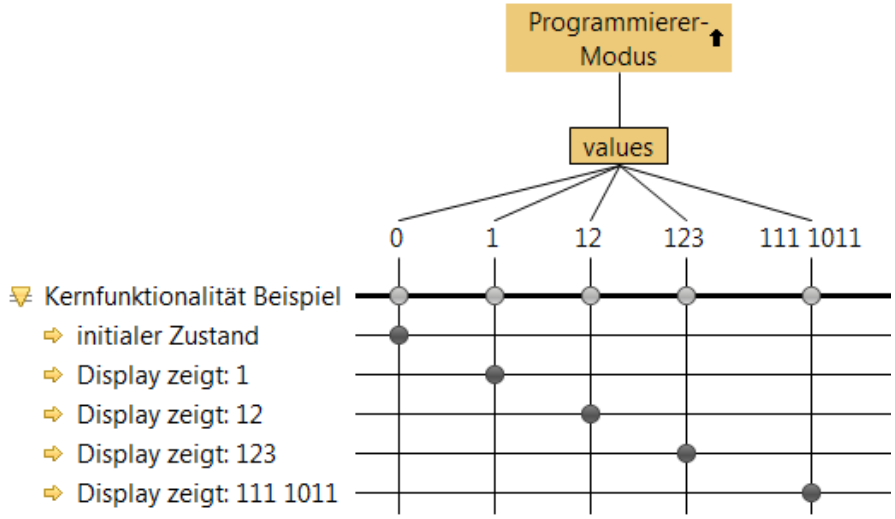


Abbildung 21: Testsequenzen werden nach der Klassifikationsbaum-Methode modelliert.

Das vollständige Idealmodell ist in Anhang A zu sehen.

Um den Tester bei der Ermittlung eines vergleichbaren Modells zu unterstützten, wurde eine Methode entwickelt, die ein solches Modell halbautomatisch ermittelt und die modellierten Testsequenzen automatisch ausführen kann. Die entwickelte Methode und die automatisch generierten Modelle werden im nächsten Abschnitt erläutert.

4. Kombination der Grundlagen zu einem Gesamtkonzept

Der Hauptbeitrag dieser Arbeit ist die Art, wie verschiedene existierende Methoden, Modelle und Werkzeuge zu einer neuen Methode und einem Werkzeug kombiniert werden. Dieser Abschnitt erläutert den entwickelten Lösungsansatz. Wie ein Modell ermittelt wird, welches das Zielmodell möglichst gut[20] annähert, wie daraus Testsequenzen generiert werden und wie diese Testsequenzen automatisch auf dem SUT ausgeführt werden.

Dabei wird top-down vorgegangen, das heißt zuerst wird ein Überblick verschafft, dessen Details erst danach erklärt werden. Es werden sowohl die Datenstrukturen der Modelle erläutert, als auch das Verhalten, um diese Modelle zu ermitteln.

4.1 Überblick: kompletter Testzyklus

Das entwickelte Verfahren unterschützt den Tester beim modellbasierten Testen einer GUI. Das Verfahren ist ein Zyklus, welcher beim Analysieren der GUI startet, daraus GUI-Modelle erstellt, aus diesen Testsequenzen herleitet und diese letztlich wieder auf der GUI ausführt. Ein Überblick über das Verfahren und seine Interaktionen mit der Umwelt und dem Tester ist in Abbildung 22 zu sehen.

[20] Eine weitere Annäherung wäre immer noch möglich. Hier wurde abgewogen zwischen der Komplexität des entwickelten Konzeptes – speziell einer Behandlung seltener Sonderfälle – und dem Mehrwert den die erhöhte Komplexität bringt.

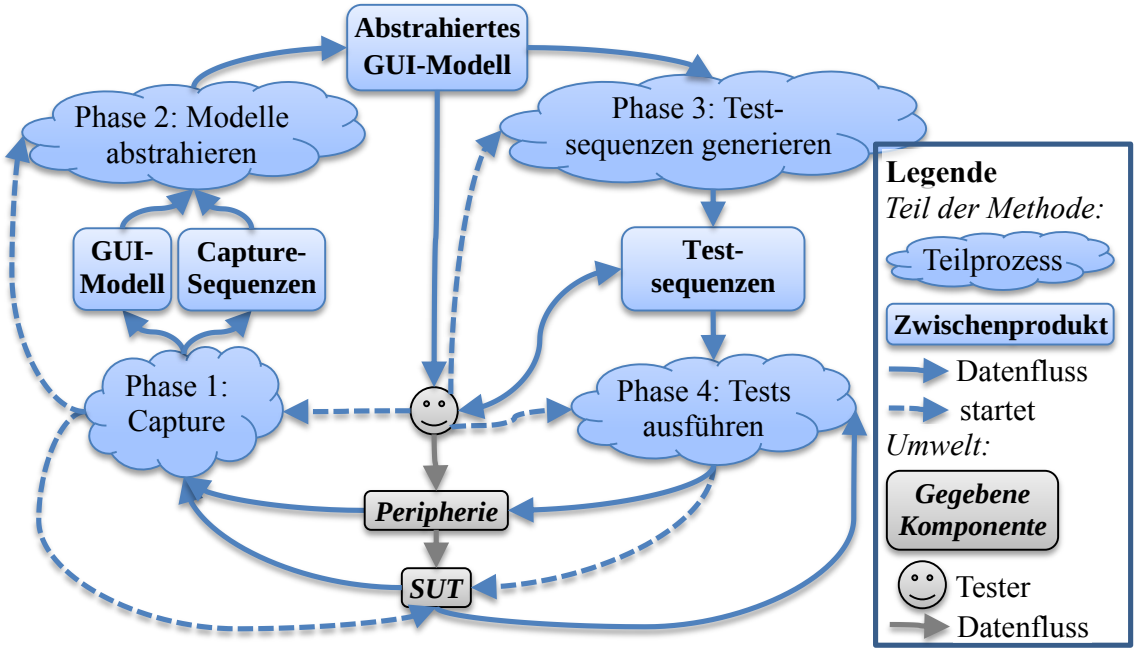


Abbildung 22: Überblick des entwickelten Verfahrens und des Zusammenspiels mit dem Tester und der Umwelt.

Zuerst initiiert der Tester den Capture-Teilprozess des Verfahrens. Dadurch wird das SUT gestartet. Der Tester führt, mittels Peripherie (Maus und Tastatur), aufzunehmende Sequenzen auf dem SUT aus. Die Eingaben in die Peripherie und die Ausgaben des SUTs werden beobachtet und daraus werden Modelle von GUI und Capture-Sequenzen erstellt und aktualisiert. Nachdem der Tester eine Sequenz beendet hat, kann er den Capture-Teilprozess beliebig oft neu starten, um weitere Sequenzen aufzunehmen. Immer nachdem der Capture-Prozess die Modelle von GUI oder Capture-Sequenzen verändert, startet der Prozess auch gleich den Teilprozess „Modelle abstrahieren“.

Der Teilprozess „Modelle abstrahieren“ nutzt die Modelle von GUI und Capture-Sequenzen und wendet einige Heuristiken an, um daraus ein abstrahiertes GUI-Modell zu erzeugen. Dieses abstrahierte GUI-Modell wird dem Tester präsentiert.

Ist der Tester zufrieden mit dem abstrahierten Modell, dann hört er auf weitere Capture-Sequenzen aufzunehmen. Der Tester initiiert dann den Teilprozess „Testsequenzen generieren“. Der Teilprozess generiert abstrakte Testsequenzen.

Zuletzt startet der Tester den Teilprozess „Tests durchführen“. Dieser Teilprozess konkretisiert die Testsequenzen und führt sie auf dem SUT aus. Dazu startet er für jede Testse-

quenz das SUT und manipuliert es über automatisch erzeugte, simulierte Peripherie-Eingaben.

Nach dem ersten Testzyklus kann der Tester die Teilprozesse erneut in beliebiger Reihenfolge anstoßen, um beispielsweise die Testsequenzen nochmals auszuführen oder die Modelle zu aktualisieren.

4.2 Erzeugte Modelle

Im Laufe des Verfahrens werden verschiedene Modelle erzeugt. Diese dienen als Schnittstellen zwischen der Methode und dem Tester oder als Schnittstelle zwischen den einzelnen Teilprozessen der Methode. Diese Modelle werden in diesem Abschnitt beschrieben.

4.2.1 GUI-Modell

Das GUI-Modell stellt eine Schnittstelle zwischen den Teilprozessen Phase 1 „Capture“ und Phase 2 „Modelle abstrahieren“ dar. Ein Überblick des GUI-Modells ist in Abbildung 23 zu sehen.

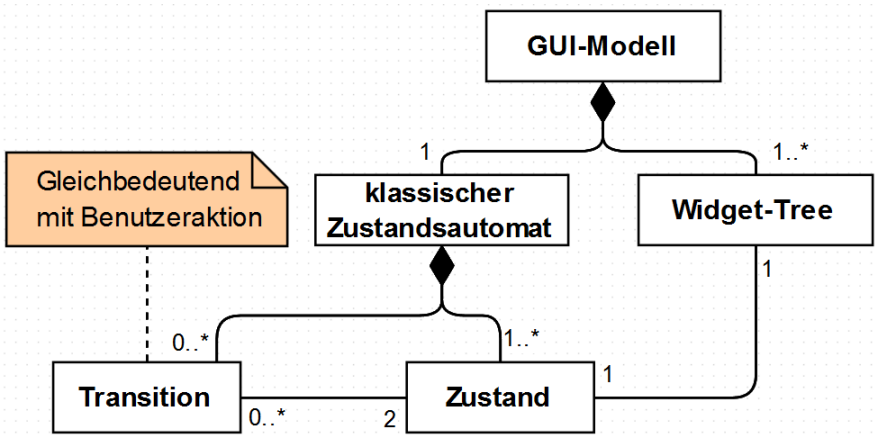


Abbildung 23: Aufbau des GUI-Modells: Ein GUI-Modell besteht aus einem klassischen Zustandsautomaten und einer Menge von Widget-Trees. Jedem Widget-Tree wird genau ein Zustand im Automaten zugeordnet.

Das GUI-Modell soll die sichtbare GUI[21] möglichst genau modellieren. Das GUI-Modell muss dem Tester nicht präsentiert werden, sondern dient als Grundlage für die spätere Abstraktion. Daher sollten im GUI-Modell noch möglichst viele Informationen über die GUI vorhanden sein, das heißt es soll einen möglichst geringen Abstraktionsgrad haben.

Das GUI-Modell setzt sich aus einer Menge stabiler Widget-Trees (vgl. Abschnitt 2.4) und einem klassischen Zustandsautomaten (vgl. Abschnitt 2.2.1) zusammen. Jedem Zustand ist bijektiv eindeutig einer der Widget-Trees zuzuordnen. Die Transitionen referenzieren ihre Ausgangs- und Folgezustände und die Zustände kennen ihre ein- und ausgehenden Transitionen.

Im GUI-Modell wird keine Historie beachtet, sondern lediglich das modelliert, was zu jeweils einem Zeitpunkt sichtbar ist. Das heißt in einem realen Systemzustand S könnte eine Aktion a zu verschiedenen Folgezuständen führen, je nachdem, was zuvor passiert ist. Genau dieses „was zuvor passiert ist“ wird im GUI-Modell nicht modelliert. Die Zustandsautomaten sind daher nichtdeterministisch.

Die Transitionen des Zustandsautomaten sind atomare Aktionen (zum Beispiel Mausklick oder Texteingabe), die von einem sichtbaren Zustand zum nächsten führen.

Obwohl der Abstraktionsgrad des GUI-Modells gering gehalten werden sollte, werden dennoch einige Abstraktionen der GUI vorgenommen. Folgende Punkte führen zur erhöhten Abstraktion:

- Das Konzept des GUI-Modells soll für alle GUIs funktionieren. Daher werden technologiespezifische Eigenheiten von GUIs nicht beachtet.[22]
- Die GUIs werden durch Zustandsautomaten und Widget-Trees modelliert. Jedes Modell bringt Abstraktion mit sich, so auch die hier verwendeten.

[21] „Sichtbare GUI“ bezeichnet die GUI Zustände, die beim Aufnehmen tatsächlich auf dem Bildschirm sichtbar waren. Bei der Aufnahme nicht passierte Zustände und Zwischenzuständen welche nur im Speicher existieren, jedoch nicht auf dem Bildschirm sichtbar sind, werden hier nicht modelliert.

[22] In den späteren Experimenten gab es keine solche technologiespezifische Eigenheit, die auf diese Art tatsächlich weggelassen werden musste.

- Es werden stabile Widget-Trees genutzt. Diese verwerfen bereits Informationen des vollständigen Widget-Trees. Andernfalls bestände der Zustandsautomat nur aus unabhängigen Zustandsketten für die verschiedenen Capture-Sequenzen. Gleichwertige Zustände ständen nicht in Beziehung zueinander.
- Benutzereingaben wurden bereits zu atomaren Aktionen zusammengefasst. Andernfalls würde jede noch so geringe Eingabe (zum Beispiel eine Mausbewegung) zu einer neuen Transition führen.

4.2.1.1 Beispiel

Ein Ausschnitt aus dem GUI-Modell des Windows-Taschenrechners könnte beispielsweise aussehen wie in Abbildung 24.

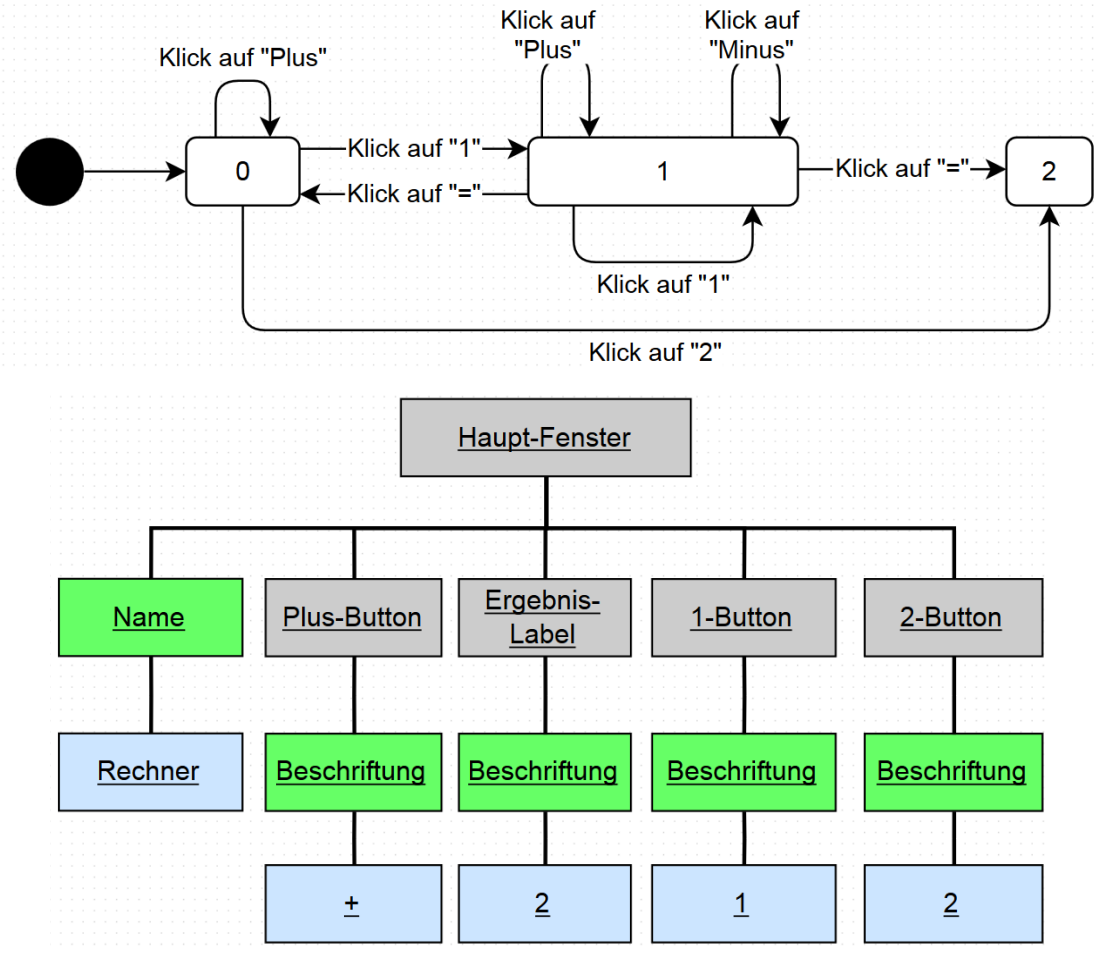


Abbildung 24: Beispiel-Ausschnitt eines GUI-Modells des Windows-Taschenrechners. Oben: Der nichtdeterministische klassische Zustandsautomat. Unten: Es Ausschnitt des Widget-Trees welcher durch Zustand „2“ repräsentiert wird.

4.2.2 Capture-Sequenzen

Die Capture-Sequenzen (Abbildung 25) stellen die zweite Schnittstelle zwischen den Teilprozessen Phase 1 „Capture“ und Phase 2 „Modelle abstrahieren“ dar. Sie modellieren die während des Capture-Prozesses vom Tester ausgeführten Sequenzen.

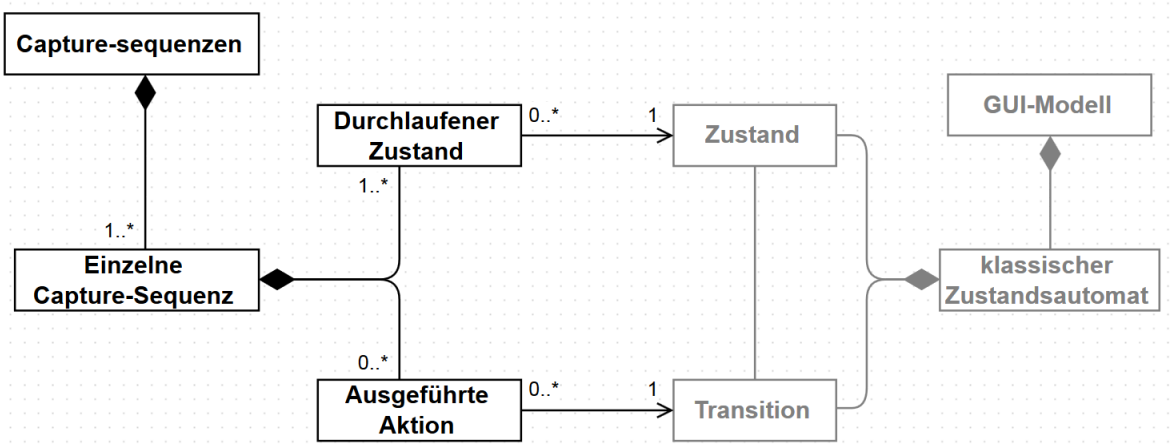


Abbildung 25: Das Modell der Capture-Sequenzen ist eine nichtleere Menge von einzelnen Capture-Sequenzen, welche wiederum als durchlaufene Zustände und ausgeführte Aktionen modelliert werden (schwarz). Capture-Sequenzen beziehen sich auf Elemente des GUI-Modells (grau).

Jede Capture-Sequenz ist eine Kette von durchlaufenen Zuständen und Transitionen. Die Zustände und Transitionen sind bereits im GUI-Modell modelliert (vgl. Abschnitt 4.2.1). Die Sequenzen referenzieren diese Zustände und Transitionen nur.

Im Vergleich zum GUI-Modell kann eine Capture-Sequenz einen Zustand oder eine Transition mehrmals referenzieren, einmal für jeden Durchlauf des Zustands oder der Transition. Dadurch modellieren die Capture-Sequenzen die Historie, die dem GUI-Modell fehlt. Andererseits ist jede Capture-Sequenz, im Vergleich zum GUI-Modell, linear (das heißt keine Verzweigung) und es besteht keine Verbindung zwischen den verschiedenen Sequenzen.

4.2.2.1 Beispiel

Ein Beispiel wie Capture-Sequenzen aussehen können, ist in Abbildung 26 zu sehen. Diese Sequenzen beziehen sich auf das Beispiel des GUI-Modells aus Abschnitt 4.2.1.1.

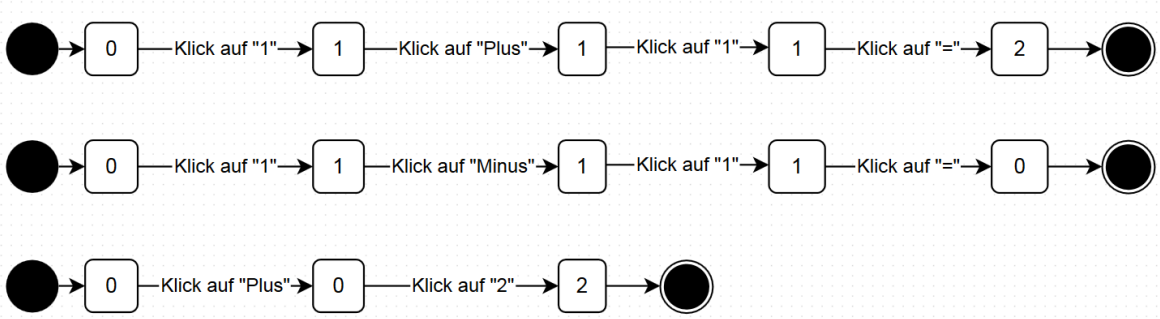


Abbildung 26: Beispiel für Capture-Sequenzen des Windows-Taschenrechners. Zu sehen sind drei verschiedene Sequenzen, welche sich auf die Zustände und Transitionen des GUI-Modells beziehen.

4.2.3 Abstrahiertes GUI-Modell

Der Teilprozess Phase 2 „Modelle abstrahieren“ erzeugt aus einem GUI-Modell ein *abstrahiertes GUI-Modell*. Dieses Modell repräsentiert die Annäherung an das angestrebte Zielmodell, ist jedoch im Vergleich dazu automatisch ermittelbar.

Das abstrahierte GUI-Modell wird dem Tester präsentiert und vom Teilprozess Phase 3 „Testsequenzen generieren“ genutzt, um daraus Testsequenzen zu generieren. Ein Überblick über das abstrahierte GUI-Modell ist in Abbildung 27 zu sehen. Die einzelnen Bestandteile werden in den folgenden Unterabschnitten genauer erläutert, Abbildung 27 soll bei diesen Erklärungen als Kontext zum Einordnen dienen.

Das abstrahierte GUI-Modell verbindet drei Teilmodelle: einen Klassifikationsbaum, einen orthogonalen, hierarchischen Zustandsautomaten und eine Menge orthogonaler Widget-Trees (diese werden nachfolgend in Unterabschnitt 4.2.3.2 definiert). Diese Modelle haben jeweils Bestandteile unterschiedlicher Granularität.

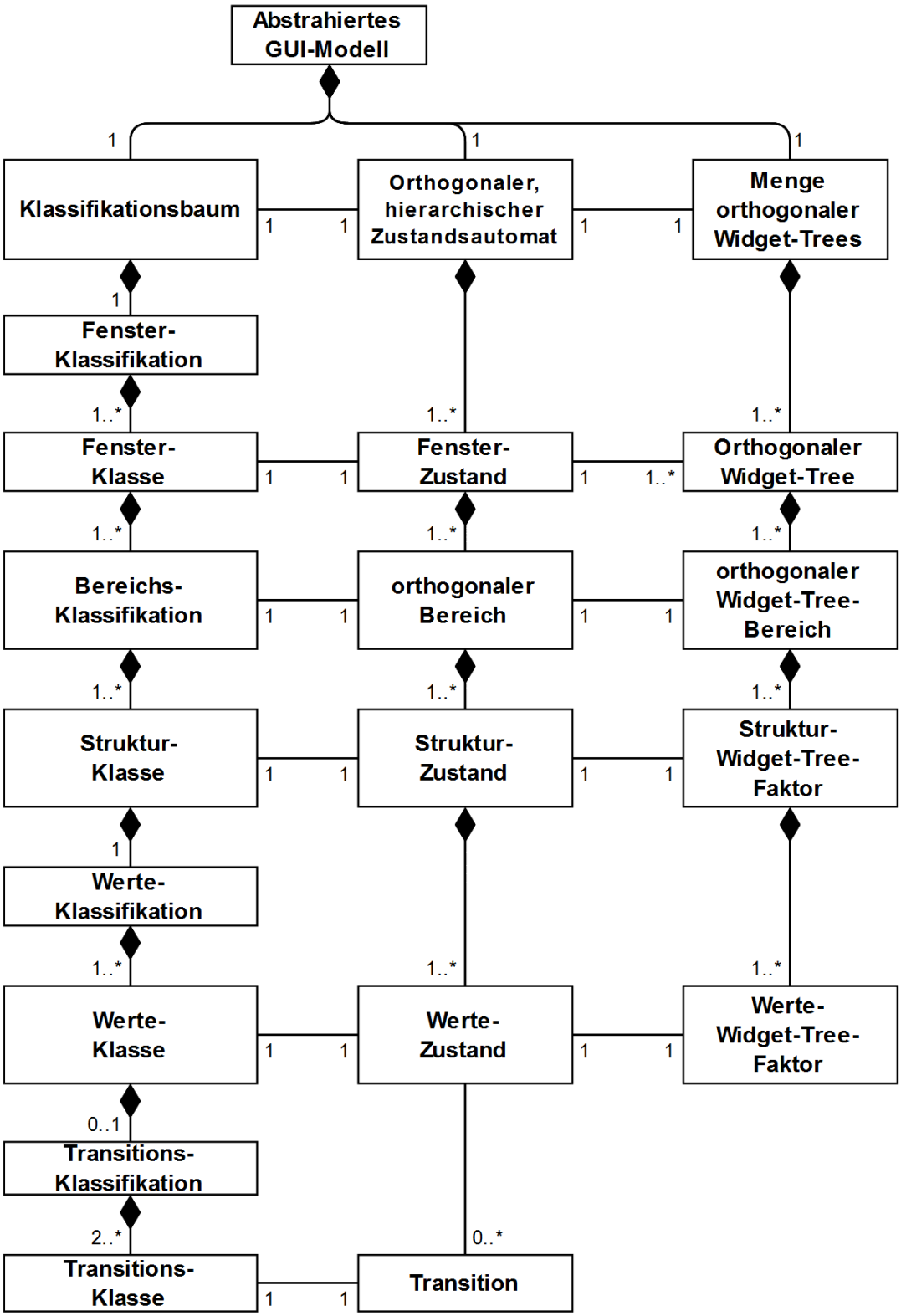


Abbildung 27: Aufbau des abstrahierten GUI-Modells. Das GUI-Modell besteht aus drei Teilmodellen. Aus dem Klassifikationsbaum (links), einem orthogonalen, hierarchischen Zustandsautomaten (mittig) und einer Menge von orthogonalen Widget-Trees (rechts). Die Modellbestandteile modellieren das SUT in verschiedenen Granularitäten, von grob (weiter oben) zu fein (weiter unten).

4.2.3.1 Struktur- und Werte-Widget-Trees

Neben der Unterscheidung zwischen stabilen und instabilen Widget-Trees (vgl. Abschnitt 2.4) unterscheiden wir in dieser Arbeit zusätzlich zwischen *Werte-Widget-Trees* und *Struktur-Widget-Trees* (vgl. Abbildung 28). Werte-Widget-Trees sind dabei Widget-Trees, wie sie bereits in Abschnitt 2.4 beschrieben wurden. Struktur-Widget-Trees verhalten sich genauso, bis auf den Unterschied, dass sie keine Werte-Knoten enthalten. Ein Modell für Werte- und Struktur-Widget-Trees ist in Abbildung 29 zu sehen.

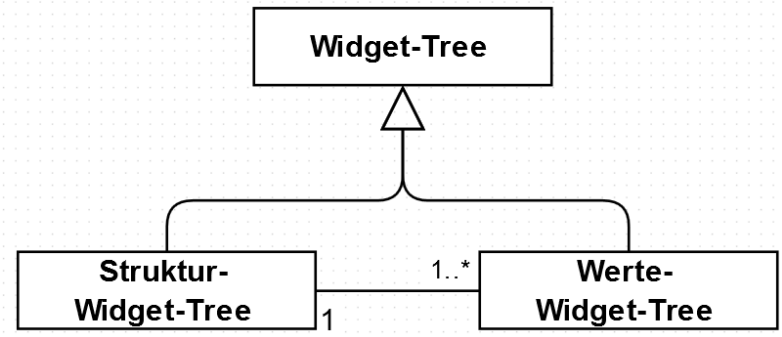


Abbildung 28: Wir unterscheiden bei Widget-Trees zwischen Werte- und Struktur-Widget-Trees. Jedem Werte-Widget-Tree ist eindeutig ein Struktur-Widget-Tree zuzuordnen, jedem Struktur-Widget-Tree mehrere Werte-Widget-Trees.

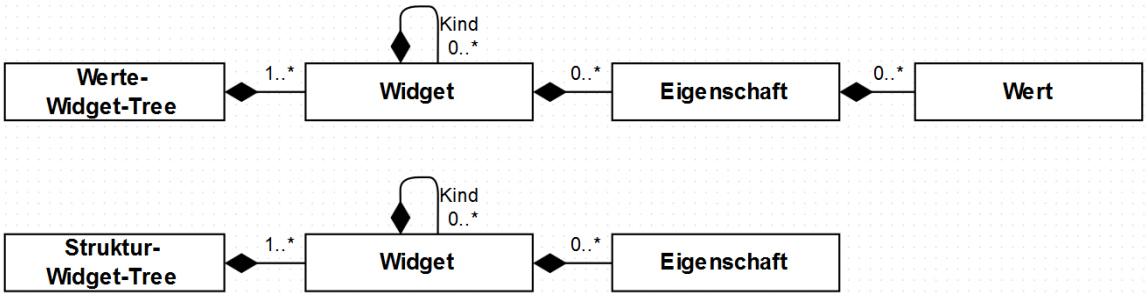


Abbildung 29: Werte-Widget-Trees (oben) sind Widget-Trees, mit Widgets, Eigenschaften und Werten. Im Vergleich dazu enthalten Struktur-Widget-Trees (unten) keine Werte.

Jedem Werte-Widget-Tree kann man also eindeutig einen Struktur-Widget-Tree zuordnen, indem man alle Werte-Knoten weglässt. Zu jedem so ermittelten Struktur-Widget-Tree kann es mehrere Werte-Widget-Trees geben. Dieser Zusammenhang zwischen Struktur- und Werte-Widget-Trees ist in Abbildung 28 dargestellt.

Beispiel

Ein Beispiel für zwei solche Widget-Trees ist in Abbildung 30 dargestellt.

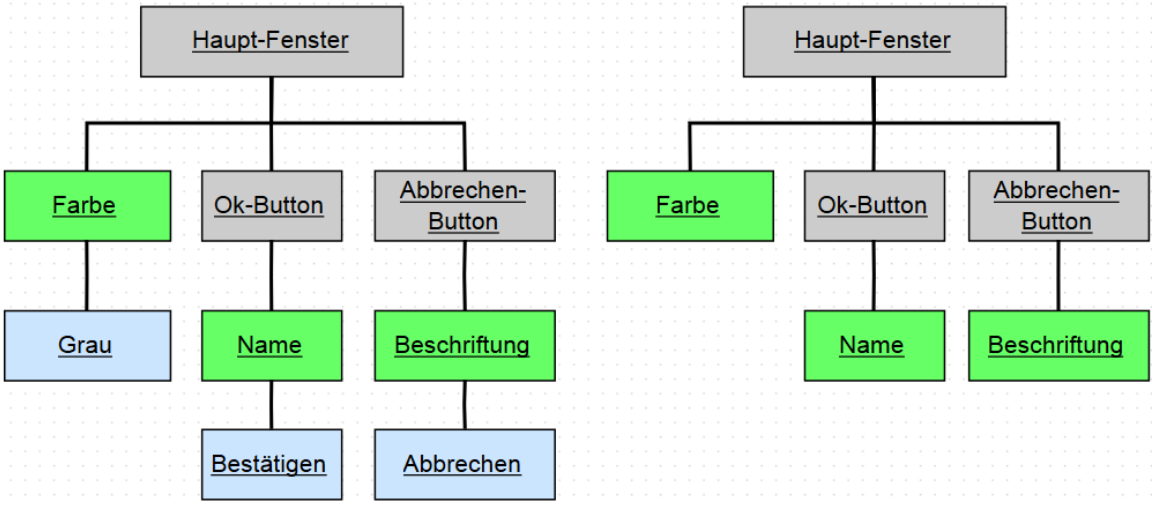


Abbildung 30: Beispiel: Vergleich zwischen einem Werte-Widget-Tree (links) und dem ihm zugeordneten Struktur-Widget-Tree (rechts).

4.2.3.2 Orthogonale Widget-Trees

In dieser Arbeit sollen unter anderem die Konzepte Widget-Tree und Zustandsautomat miteinander kombiniert werden. Bisher wurde dazu der Zusammenhang hergestellt, dass ein Zustand stets einem Widget-Tree entspricht. Ein Problem ist nun, das Konzept der orthogonalen Zustände (vgl. Abschnitt 2.2.3) sinnvoll mit den Widget-Trees zu vereinen. Dazu muss das Konzept der Widget-Trees erweitert werden.

Definitionen

Diese erweiterten Widget-Trees, welche im Rahmen dieser Arbeit entstanden sind, nennen wir auch *orthogonal*. Wir sprechen nachfolgend also zum Beispiel von einem stabilen *orthogonalen* Werte-Widget-Tree. Zur besseren Differenzierung werden nichtorthogonale Widget-Trees nachfolgend auch *klassische Widget-Trees* genannt.

Zusätzlich zu den bereits vorhandenen Knotentypen (Widget-, Eigenschafts- und Werte-Knoten) führen wir noch einen weiteren Knotentyp *Verbindungspunkt* ein. Dieser kann einen beliebigen Teilbaum des Widget-Trees substituieren, sofern der Verbindungspunkt

dadurch entweder Blatt- oder Wurzelknoten ist. Einen Widget-Tree mit Verbindungspunkten nennen wir einen *Widget-Tree-Faktor*, oder im klaren Kontext auch nur *Faktor*. Jeder Verbindungspunkt eines Faktors hat ein *Gegenstück* (welches ein spezieller Verbindungspunkt ist) in mindestens einem anderen Faktor. Ist dabei ein Verbindungspunkt Wurzelknoten seines Faktors, so sind seine Gegenstücke Blattknoten ihrer Faktoren und umgekehrt. Die Menge der Faktoren aller Gegenstücke nennen wir – analog zu den Zustandsautomaten – *Bereiche*. Ein Verbindungspunkt *verbindet* seinen Faktor mit allen Faktoren, in denen er ein Gegenstück hat. Die Menge aller Faktoren, welche auf diese Art direkt oder indirekt verbunden sind nennen wir *orthogonalen Widget-Tree.*[23]

Ein orthogonaler Widget-Tree repräsentiert einen oder mehrere klassische Widget-Trees. Zu jedem so repräsentieren klassischen Widget-Tree kann eine Kombination von Widget-Tree-Faktoren gefunden werden, welche diesen klassischen Widget-Tree eindeutig identifiziert. Dabei enthält die Menge der identifizierenden Faktoren maximal einen Faktor aus jedem Bereich.

Beispiel

Abbildung 31 zeigt ein Beispiel eines orthogonalen Widget-Trees und der damit zusammenhängenden Begriffe. Dieser orthogonale Widget-Tree repräsentiert mehrere klassische Widget-Trees. Diese sind in Abbildung 32 dargestellt. Jeder dieser klassischen Widget-Trees ist eindeutig durch Faktoren der verschiedenen Bereiche identifiziert. Diese Faktoren und Bereiche sind auch in Abbildung 32 zu sehen.

[23] Vergleichbar mit einer Zusammenhangskomponente in Graphen.

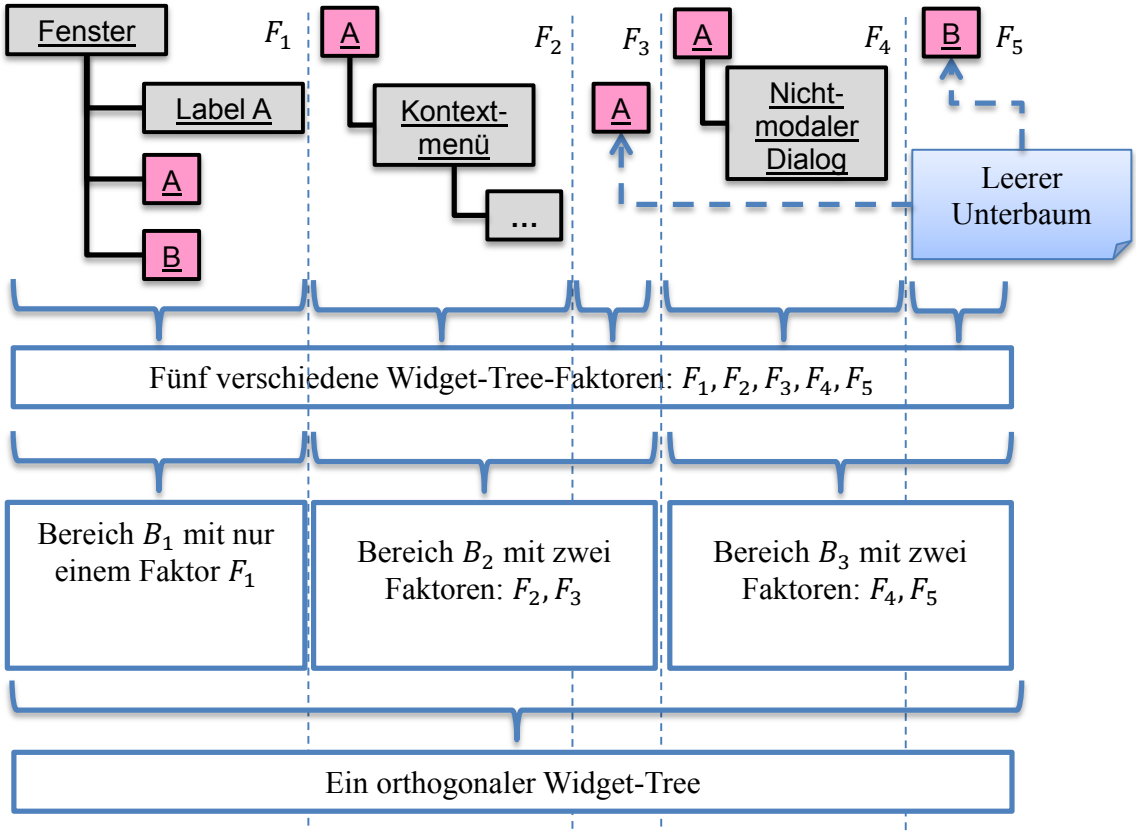


Abbildung 31: Ein orthogonaler Widget-Tree, bestehend aus 3 verschiedenen Bereichen mit insgesamt 5 Faktoren. Verbindungspunkte werden hier und im Folgenden pink dargestellt.

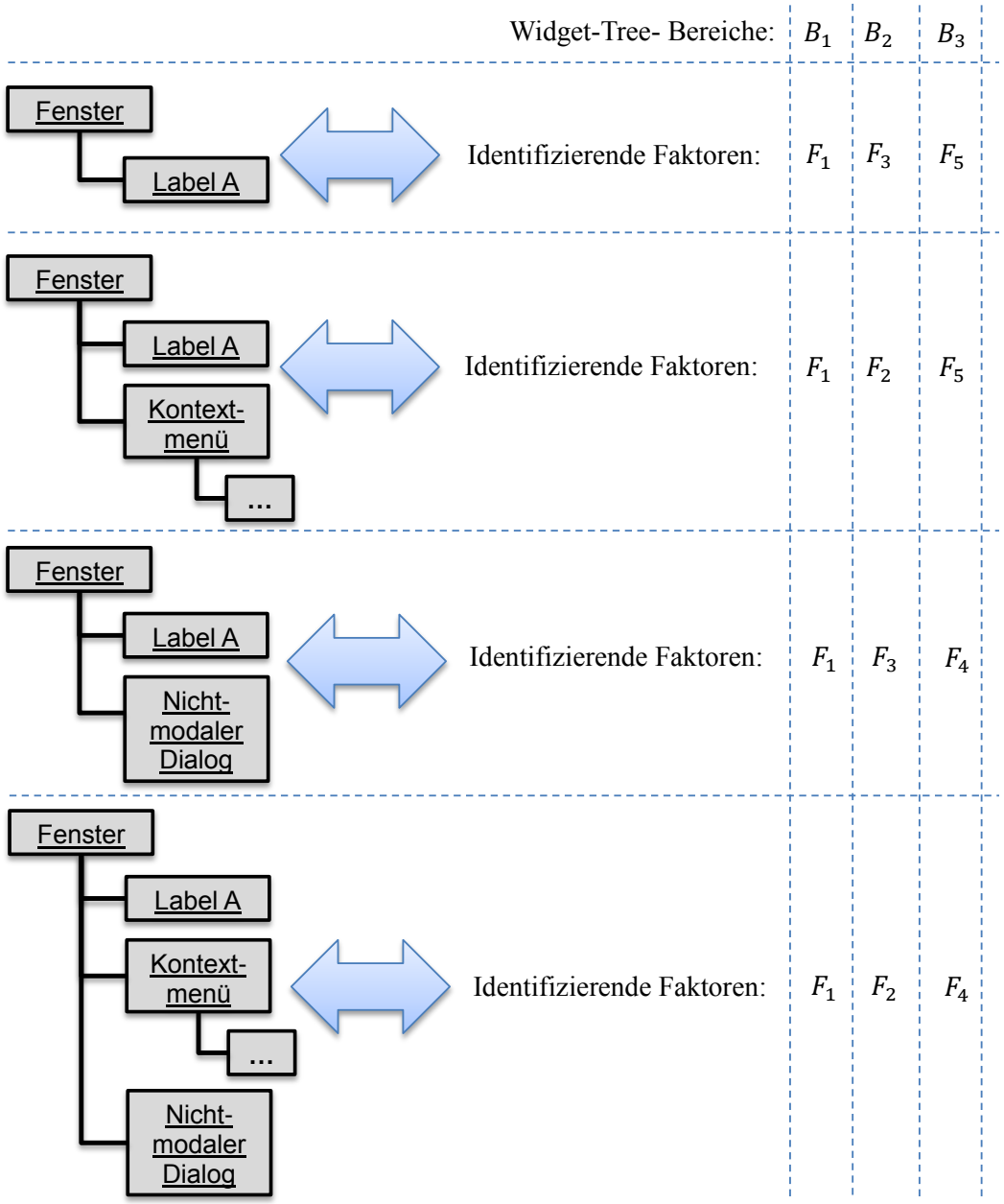


Abbildung 32: Zu sehen sind die vier klassischen Widget-Trees welche durch den orthogonalen Widget-Tree repräsentiert werden. Außerdem sind die Bereiche und deren jeweils repräsentierende Faktoren dargestellt.

Ein konkreteres Beispiel für einige Widget-Tree-Faktoren ist später in Abschnitt „Orthogonale Zustandsautomaten“ zu sehen.

4.2.3.3 Hierarchische Zustandsautomaten

Um die modellierten Zustandsautomaten hierarchisch zu verfeinern (vgl. Abschnitt 2.2.3.2), wurden Regeln dafür aufgestellt, wie eine konkrete Hierarchie modelliert wird.

Die Strukturierung erfolgt in drei Hierarchieebenen. Wir nennen die Zustände auf den drei Ebenen *Fenster-Zustände*, *Struktur-Zustände* und *Werte-Zustände* (vgl. auch Abbildung 33):

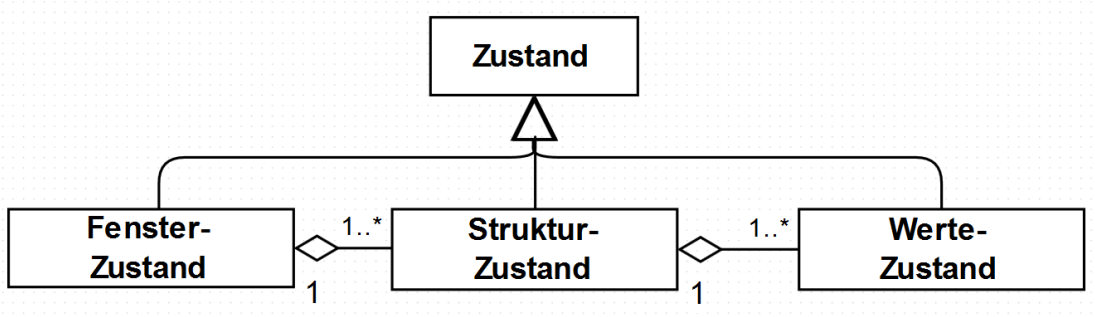


Abbildung 33: Es wird zwischen drei Granularitäten von Zuständen unterschieden. Werte-Zustände sind in Struktur-Zuständen gruppiert und Struktur-Zustände sind wiederum in Fenster-Zuständen gruppiert.

1) Auf der obersten Ebene werden die Zustände nach den Namen der GUI-Fenster gruppiert. Diese Überzustände werden nachfolgend *Fenster-Zustände* genannt. Zustände mit gleichem Fensternamen werden dem gleichen Fenster-Zustand zugeordnet, bzw. zu jedem Fensternamen existiert genau ein Fenster-Zustand. Bei modalen Dialog-Fenstern wird der Name des aktiven Fensters genutzt. Nichtmodale Dialog-Fenster erhalten keine eigenen Fenster-Zustände, sie werden dem Fenster-Zustand ihres (modalen) Haupt-Fensters zugeordnet.[24]
2) Innerhalb der Fenster-Zustände gibt es *Struktur-Zustände.* Für jeden stabilen Struktur-Widget-Tree-Faktor existiert genau ein solcher Struktur-Zustand.
3) Innerhalb der Struktur-Zustände befinden sich wiederum *Werte-Zustände*. Es existiert je ein solcher *Werte-Zustand* für jeden stabilen Werte-Widget-Tree-Faktor.

Nachfolgend sprechen wir auch von einer *Zuordnung*, immer dann wenn etwas hierarchisch unter etwas anderem angeordnet ist. So ist zum Beispiel ein Werte-Zustand im-

[24] Die Fenster-Zustände entsprechen den Knoten der GUI-Trees aus dem GUI-Forest-Modell von Memon et al. [MeBN03a]. Der Zustandsautomat aller Fenster-Zustände und den Transitionen dazwischen entspricht etwa einem GUI-Forest, allerdings ohne eine Baumstruktur zu erzwingen.

mer einem Struktur-Zustand *zugeordnet*. Analog funktioniert es für deren Widget-Trees: Ein Werte-Widget-Tree ist immer einem Struktur-Widget-Tree *zugeordnet*.

Beispiel

Ein Beispiel der hierarchischen Einteilung der Zustandsautomaten ist in Abbildung 34 zu sehen.

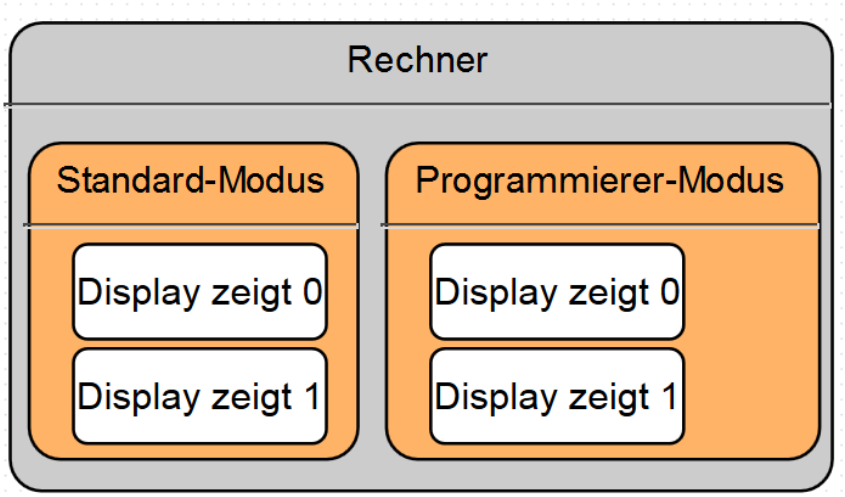


Abbildung 34: Beispiel: Die hierarchische Unterteilung des Zustandsautomaten erfolgt in drei Ebenen. Auf der obersten Ebene existiert ein Fenster-Zustand (grau) für jeden Fensternamen („Rechner“ im Beispiel). Darunter existiert ein Struktur-Zustand (orange) für jeden stabilen Struktur-Widget-Tree-Faktor (zum Beispiel der Widget-Tree-Faktor für den Programmierer-Modus des Taschenrechners). Diesen Struktur-Zuständen sind wiederrum Werte-Zustände (weiß) zugeordnet, welche von den Werten der assoziierten Widget-Trees abhängen.

4.2.3.4 Orthogonale Zustandsautomaten

Um eine Zustandsexplosion zu vermeiden wird vom Konzept der Orthogonalität Gebrauch gemacht. Die Zustandsautomaten des abstrahierten GUI-Modells werden nicht nur hierarchisch untergliedert, sondern auch in orthogonale Bereiche zergliedert. So wird das Verhalten bestimmter Regionen der GUI als orthogonal, das heißt weitestgehend unabhängig, zu anderen Regionen der GUI angenommen.[25] So ändert beispielsweise das Öffnen eines Kontextmenüs nichts an der restlichen GUI. Analog ändert das Markieren

[25] Hierbei handelt es sich um eine Heuristik, welche zwar nicht zwingend ist, aber bei vielen SUTs so zu beobachten ist und welche dabei hilft das Idealmodell aus Abschnitt 3 gut anzunähern.

einer Checkbox oft nichts an der restlichen GUI. Für jeden dieser „unabhängigen“ Teile – welche durch je einen eigenen Widget-Tree-Bereich identifiziert sind – existiert ein eigener orthogonaler Bereich im Fenster-Zustand.

Da diese Bereiche nicht komplett unabhängig voneinander sind, erfolgt die Kommunikation zwischen den Bereichen mittels Aktionen und Ereignissen.

Beispiel

Ein Beispiel für orthogonales Verhalten im Windows-Taschenrechner ist in Abbildung 35 zu sehen. Wie dieses Verhalten mittels orthogonaler Bereiche modelliert wird, zeigt Abbildung 36. Ein Ausschnitt der assoziierten Widget-Tree-Faktoren ist in Abbildung 37 zu sehen.

Das abstrahierte GUI-Modell abstrahiert das GUI-System über die Capture-Sequenzen hinaus und modelliert dadurch auch zuvor nicht aufgenommenes Verhalten. So modelliert der Zustandsautomat aus Abbildung 36 auch jenen Zustand, in dem sich der Taschenrechner im Standard-Modus befindet, während das Ansicht-Menü geöffnet ist, obwohl dies nicht Teil der Capture-Sequenz aus Abbildung 35 ist.

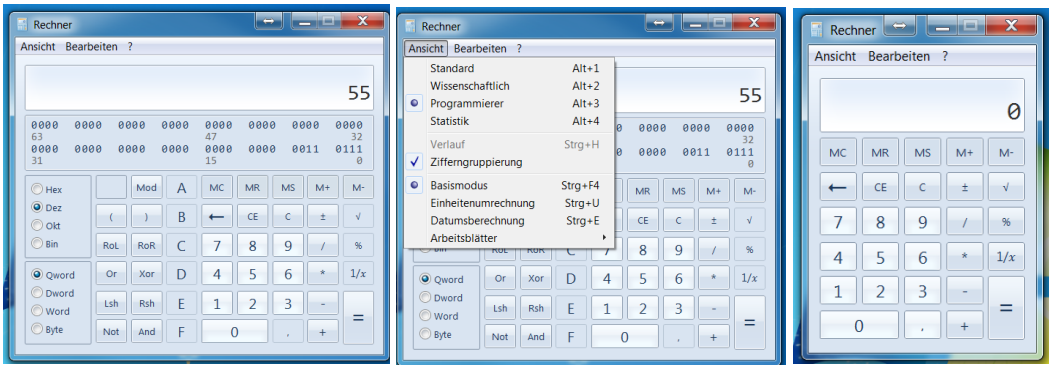


Abbildung 35: Beispiel: das Öffnen des Ansicht-Menüs ändert nichts daran, dass der Rechner sich im Programmierer-Modus befindet oder „55“ im Display steht. Das Auswählen des Menüeintrags „Standard“ ändert dies allerdings schon.

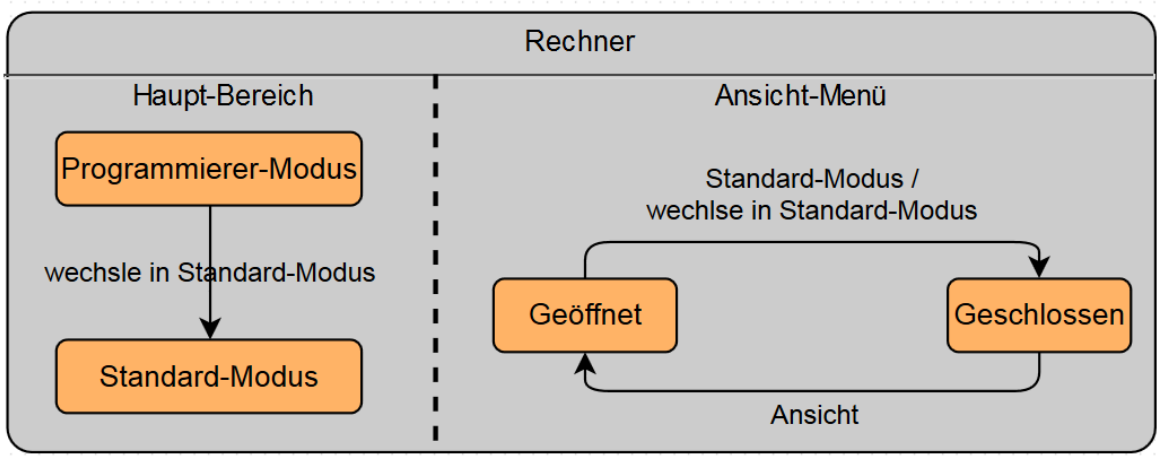


Abbildung 36: Voneinander unabhängige Teile der GUI werden in verschiede orthogonale Bereiche aufgeteilt. Beispielsweise erhält das teils unabhängige Ansicht-Menü einen eigenen orthogonalen Bereich. Die Effekte, die durch die Auswahl des Menüeintrags „Standard“ im restlichen Taschenrechner („Haupt-Bereich“) auftreten, werden durch eine Aktion und ein Event („wechsle in Standard-Modus“) modelliert.

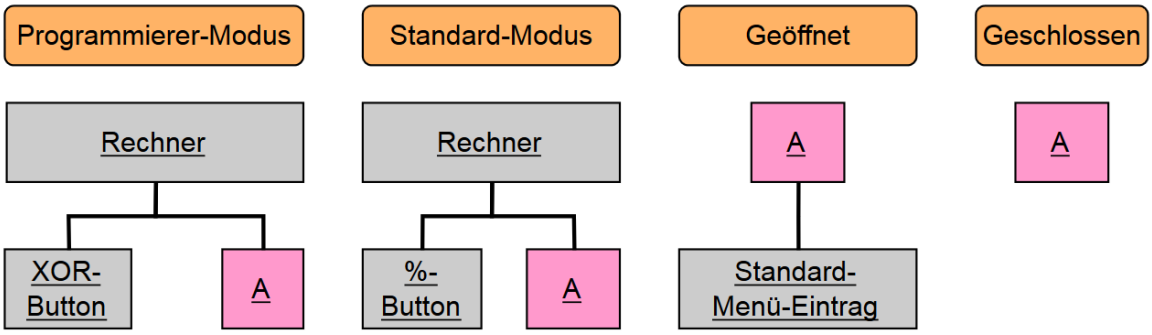


Abbildung 37: In diesem Bild ist ein kleiner Ausschnitt der Widget-Tree-Faktoren zu den vier Struktur-Zuständen aus dem Beispiel (Abschnitt 4.2.3.4) zu sehen. Die Widget-Trees verschiedener orthogonaler Bereiche sind durch Verbindungspunkte (pink) verbunden.

4.2.3.5 Klassifikationsbaum für GUIs

Zu dem orthogonalen, hierarchischen Zustandsautomat wird auch ein Klassifikationsbaum modelliert. Dabei wird – wie in Kruse und Wegener [Krus11, KrWe12] – jedem Zustand eine Klasse und jedem Bereich eine Klassifikation zugeordnet. Während bei Kruse und Wegener eine allgemeine Abbildungsvorschrift zwischen orthogonalen, hierarchischen Zustandsautomaten und Klassifikationsbäumen beschrieben wird, behandelt diese Arbeit konkrete Automaten und Bäume zur Beschreibung von GUIs. Der Inhalt der „GUI-Zustandsautomaten“ ist klar definiert. Zum Beispiel hat er genau drei Hierarchieebenen:

Fenster-, Struktur- und Werte-Zustände (vgl. Abschnitte 4.2.3.3 und 4.2.3.4). Analog hat der modellierte Klassifikationsbaum drei Arten von Klassen – *Fenster-*, *Struktur-* und *Werte-Klassen* (vgl. Abbildung 38) – und Klassifikationen für die Regionen im Zustandsautomaten.[26] Es gelten folgende konkrete Konventionen für die Benennung der Klassen und Klassifikationen:

Die Klassifikationen jedes Bereichs heißen genau wie der Bereich. Die Klassen heißen wie die mit ihnen assoziierten Zustände.

Die oberste Klassifikation heißt immer *windows* (engl. Fenster), da unter ihr die Fenster-Klassen angeordnet sind.

Die Klassifikation unter einer Struktur-Klasse heißt immer *values* (engl. Werte), da unter ihnen die Werte-Klassen angeordnet sind.

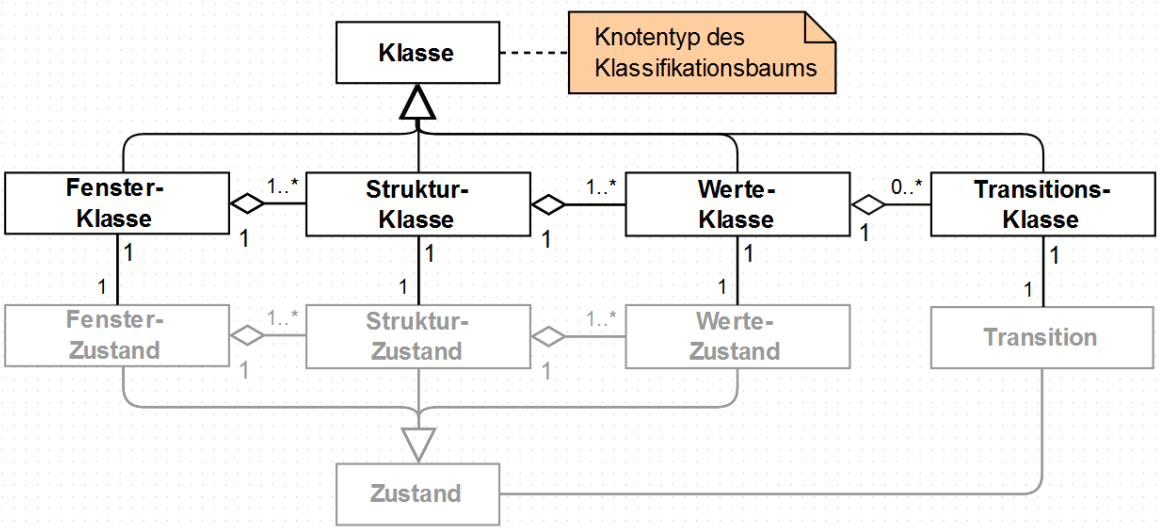


Abbildung 38: Analog zu den verschiedenen Typen von Zuständen (grau) werden auch verschiedene Typen von (Klassifikationsbaum-)Klassen (schwarz) modelliert. Zusätzlich werden auch Transitionen als Klassen modelliert.

[26] In unserem Modell werden nur die Regionen unterhalb der Fenster-Zustände explizit modelliert. Struktur- und Werte-Zustände werden nicht orthogonal zergliedert und enthalten daher direkt ihre Unterzustände, anstatt indirekt über eine Region. Dieses „Überspringen der Indirektion“ ist im Klassifikationsbaum nicht möglich, weshalb auch Klassifikationen ohne äquivalente Region existieren. Man könnte hier zusätzliche Regionen unter den Struktur- und Werte-Zuständen modellieren, um eine höhere Konsistenz, allerdings auch mehr Indirektionen und unnütze Klassen, zu erhalten.

Im Rahmen dieser Arbeit wird die Abbildungsvorschrift von [Krus11, KrWe12] noch folgendermaßen erweitert:

Existiert von einem Ausgangszustand mehr als eine Transition zu einem Folgezustand, dann ist unterhalb der Klasse des Ausgangszustands eine Klassifikation *transitions* (engl. Transitionen), welche eine Transitions-Klasse für jede dieser Transitionen enthält (vgl. Abbildung 38).

Durch diese Erweiterung ist die Abbildung vom Klassifikationsbaum auf den Zustandsautomaten nicht mehr wie in [Krus11, KrWe12] möglich. Klassen repräsentieren nicht mehr ausschließlich Zustände, sie können jetzt auch Transitionen repräsentieren. Klassifikationen repräsentieren nicht mehr ausschließlich Bereiche, sie können jetzt auch Mengen ausgehender Transitionen repräsentieren. Deshalb unterscheidet unser Modell zwischen den verschiedenen Klassen: Fenster-Klasse, Struktur-Klasse, Werte-Klasse und Transitions-Klasse.

Beispiel

Ein Beispiel, wie ein Klassifikationsbaum mit einem Zustandsautomaten zusammenhängt ist in Abbildung 39 und Abbildung 40 zu sehen.

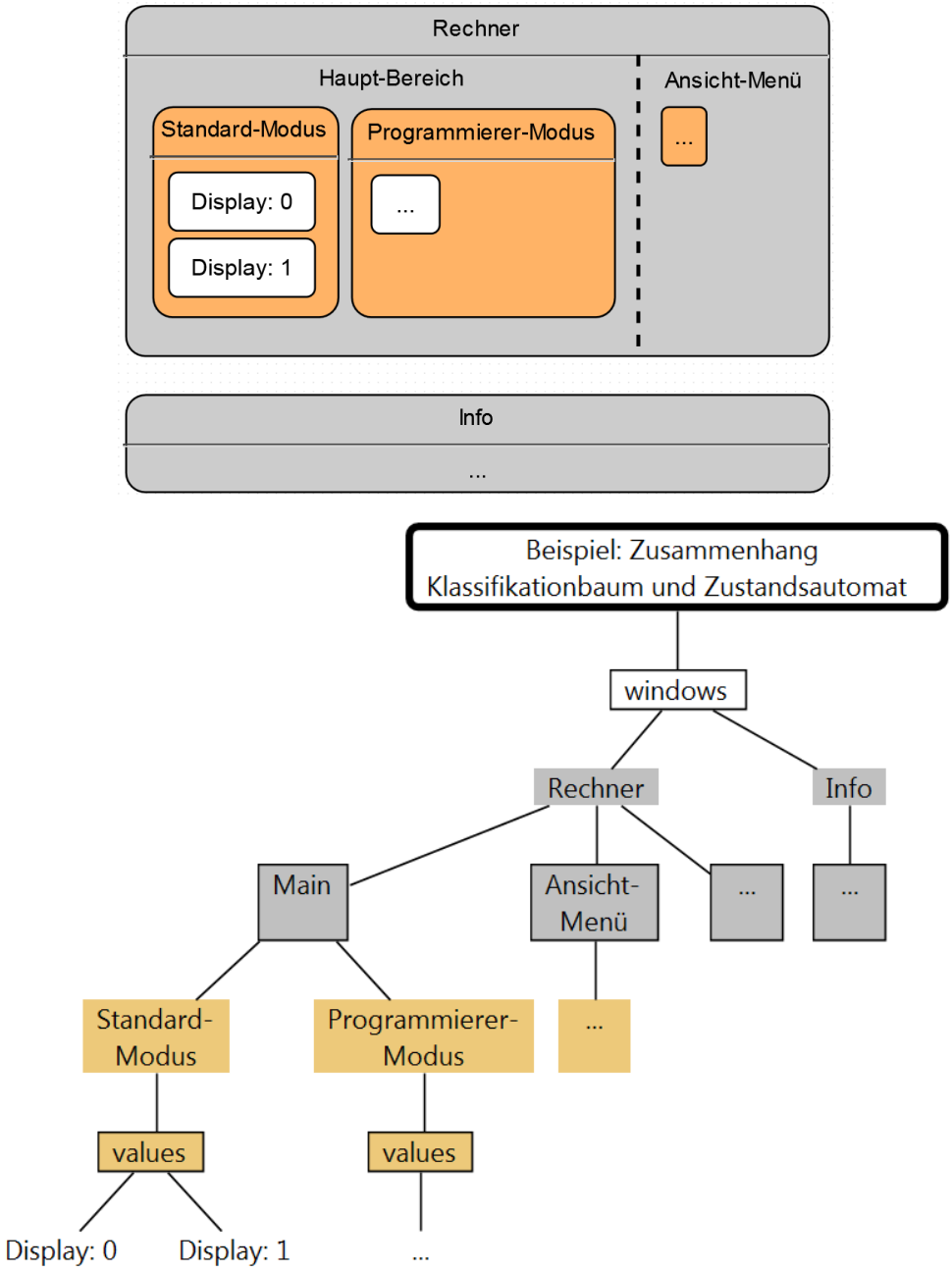


Abbildung 39: Betrachtet man nur die Zustände, ist die Abbildungsvorschrift zwischen Zustandsautomaten (oben) und Klassifikationsbaum (unten) ist die gleiche wie in [Krus11, KrWe12]: Zustände entsprechen Klassen, Bereiche entsprechen Klassifikationen. Besondere Konventionen für die Namensgebung werden angewendet.

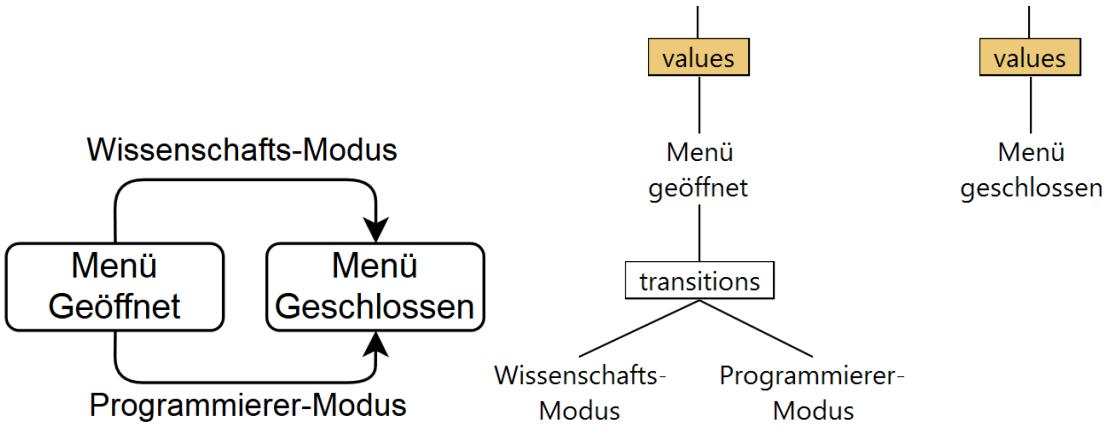


Abbildung 40: Existiert im Zustandsautomat (links) von einem Ausgangszustand zu einem Folgezustand mehr als eine Transition, dann werden diese Transitionen als Transitions-Klassen in einer eigenen Klassifikation unterhalb des Ausgangszustands im Klassifikationsbaum (rechts) abgebildet.

4.2.3.6 Erhöhter Abstraktionsgrad

Durch die zusätzliche Abstraktion ist das *abstrahierte GUI-Modell* weiter vom realen GUI-System entfernt als das (einfache) *GUI-Modell* (vgl. Abschnitt 4.2.1). Es werden Verallgemeinerungen getroffen, wodurch zusätzliches Verhalten der GUI modelliert wird, welches zuvor nicht unbedingt mittels Capture aufgenommen wurde. Anhand des abstrahierten GUI-Modells kann getestet werden, ob die GUI sich wirklich so verhält, wie das neu modellierte Verhalten es vorhersagt.

Die Richtigkeit des durch Abstraktion entstehenden Verhaltens hängt von verschiedenen Faktoren ab:

Die zugrundeliegenden Heuristiken: Die Abstraktionsschritte basieren auf Annahmen und Heuristiken wie GUIs im Allgemeinen funktionieren. So wird zum Beispiel angenommen, dass Kontextmenüs sich orthogonal zur restlichen GUI verhalten. Je besser die Heuristiken das allgemeingültige Verhalten von GUIs beschreiben, desto besser wird das resultierende abstrahierte GUI-Modell die reale GUI beschreiben. Im Rahmen dieser Arbeit werden einfache Heuristiken vorgestellt, um die Grundidee des Abstraktionsschrittes zu veranschaulichen.

Das Standard-Konformität der GUI: Das entwickelte Verfahren soll für alle GUIs funktionieren. Daher müssen die Modelle allgemeingültig formuliert werden. Hat eine GUI nicht eingeplante Elemente (zum Beispiel eine Zeichenebene oder Radiobuttons die sich wie Checkboxen verhalten), so wird das abstrahierte Modell deren Verhalten vielleicht falsch vorhersagen. Diese falsche Vorhersage kann ein Mangel im Abstraktionsprozess sein, oder sie kann für den Tester als Indikator dafür dienen, dass die GUI zu weit von Standards abweicht.

Nicht sichtbare Variablen: Für gewöhnlich haben Software-Systeme Zustände, die über den sichtbaren Zustand hinausgehen. Durch eine teilweise Analyse der GUI können diese Zustände nicht immer vollständig modelliert werden. Dieses Problem ist Bestandteil jedes Black-Box-Tests und kann nur auf zwei Arten umgangen werden. Entweder ergänzt der Tester mit seinem Kontextwissen manuell die Modelle, oder die GUI muss – durch Capture aller möglichen Sequenzen – vollständig analysiert werden. Die vollständige Analyse ist bereits für einfachste GUIs impraktikabel.

4.2.4 Testsequenzen

Das Modell der Testsequenzen spezifiziert, wie das GUI-System getestet wird. Es wird im Teilprozess Phase 3 „Testsequenzen generieren“ aus dem abstrahierten GUI-Modell hergeleitet und dient dem Teilprozess Phase 4 „Tests ausführen“ als Spezifikation der auszuführenden Eingaben.

Die Testsequenzen werden – wie in Abschnitt 2.1.3 beschrieben – in einer Testmatrix zum Klassifikationsbaum des abstrahierten GUI-Modells spezifiziert. Dabei wird in jeder Sequenz eine Folge von anzunehmenden Zuständen definiert. Jeder Zwischenzustand in dieser Sequenz wird nachfolgend *Test-Zustand* genannt.[27] Der Übergang von einem Test-Zustand in den nächsten wird *Test-Schritt* genannt. Jeder Test-Zustand beschreibt einen (sichtbaren) Zustand in dem das GUI-System sich befinden kann.

[27] Kruse und Wegener [Krus11, KrWe12] nennen diese Test-Zustände stattdessen Test-Schritte. In dieser Arbeit beschreibt der Begriff Test-Schritt stattdessen den Übergang von einem Test-Zustand in den nächsten.

Die konkreten Widgets und darauf auszuführenden Aktionen sind als Teil der Test-Schritte definiert. Abbildung 41 zeigt das Modell der Testsequenzen und wie diese mit dem Klassifikationsbaum des abstrahierten GUI-Modells zusammenhängen.

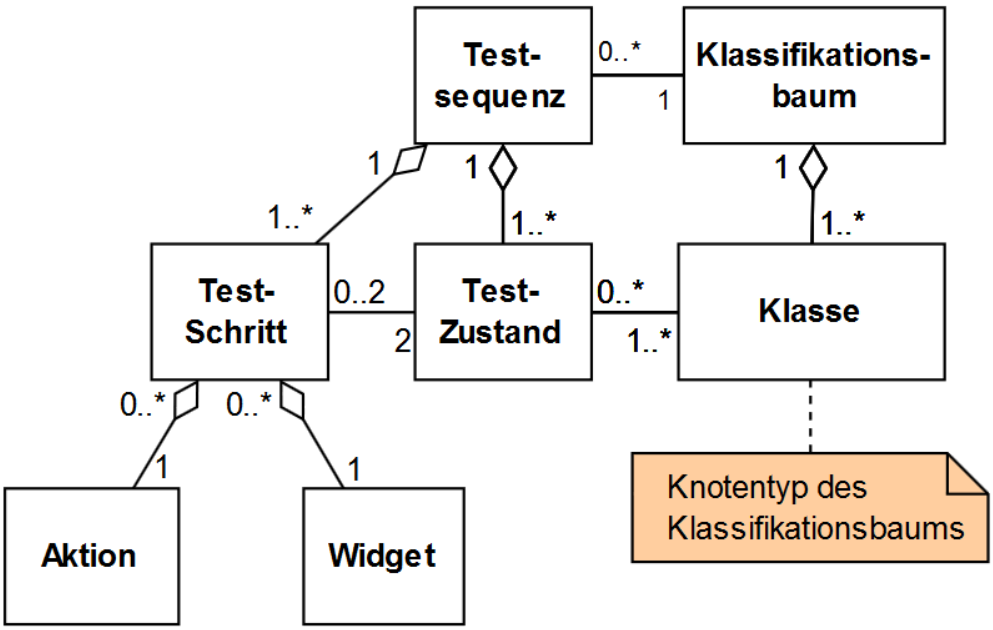


Abbildung 41: Testsequenzen sind Kompositionen aus Test-Zuständen und Test-Schritten. Test-Schritte beschreiben ein Widget und eine darauf auszuführende Aktion. Jede Testsequenz mit dem Klassifikationsbaum des abstrahierten GUI-Modells assoziiert. Test-Zustände sind mit Klassen dieses Baums assoziiert.

4.2.4.1 Beispiel

Ein Beispiel für eine modellierte Testsequenz ist in Abbildung 42 zu sehen. In dieser Testmatrix ist nicht zu sehen, welche Test-Schritte die Zustandswechsel verursachen. Diese Test-Schritte sind in Abbildung 43 dargestellt.

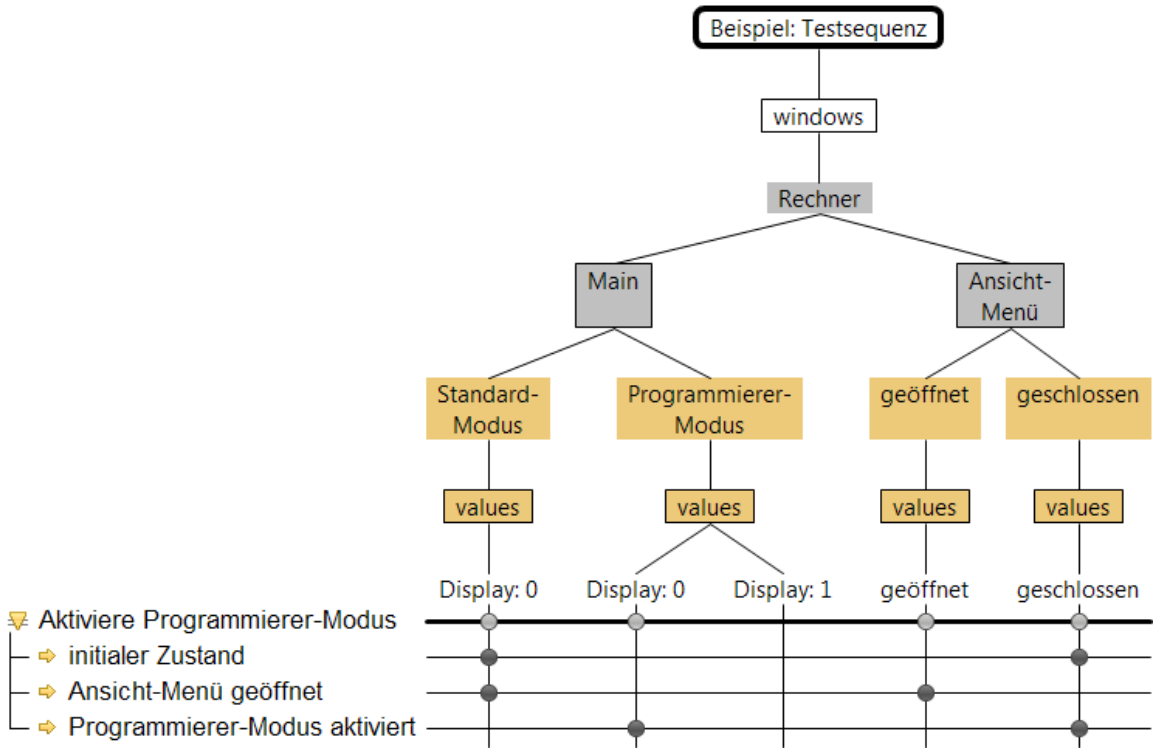


Abbildung 42: Im Bild ist beispielhaft eine Testsequenz dargestellt, welche den Programmierer-Modus aktiviert, indem drei Test-Zustände durchlaufen werden. Zuerst befindet sich der Taschenrechner im initialen Zustand, dann ist das Ansicht-Menü geöffnet und zuletzt ist der Programmierer-Modus aktiviert.

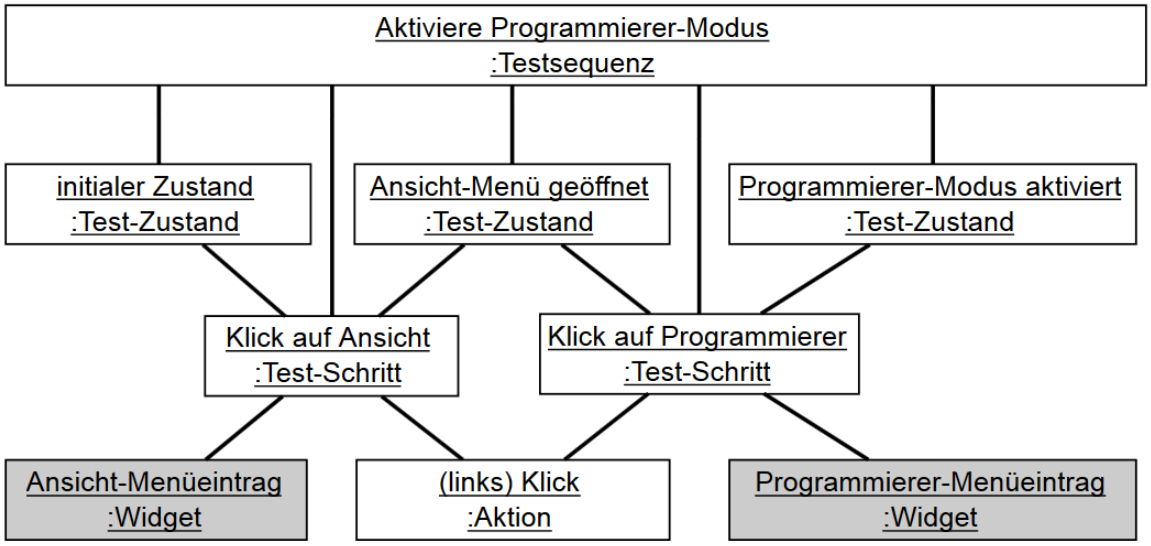


Abbildung 43: Im Bild ist beispielhaft das Objektdiagramm einer Testsequenz dargestellt, welche den Programmierer-Modus aktiviert. Zusätzlich zu den Informationen der Testmatrix ist hier zu sehen, dass zum Durchlaufen dieser Zustände ein Klick auf den Menüeintrag „Ansicht“ und ein Klick auf den Menüeintrag „Programmierer“ notwendig sind.

4.2.5 Grenzen der Modelle

Unsere Methode und die erzeugten Modelle haben Grenzen und können nicht alle Klassen von GUIs beschreiben. Sie müssten erweitert werden, um auch andere Klassen beschreiben zu können. Memon et al. [8] zählen beispielhaft einige Grenzen ihres Modells auf, welche gleichermaßen auf unsere Methode und unsere Modelle zutreffen:

- Web-Benutzeroberflächen mit Synchronisations- bzw. Timing Constraints
- Videoplayer mit einem kontinuierlichen Video-Stream, anstatt einer Sequenz von diskreten Frames
- Nichtdeterministische GUIs, in welchen die Konstruktion eines Modells via Reverse Engineering nicht ohne weiteres möglich ist, da die Auswirkungen von Ereignissen nicht vorhergesagt werden können

4.3 Phase 1 „Capture“

Die erste Phase des Verfahrens sorgt für die Erstellung eines Modells mittels Capture.

4.3.1 Überblick

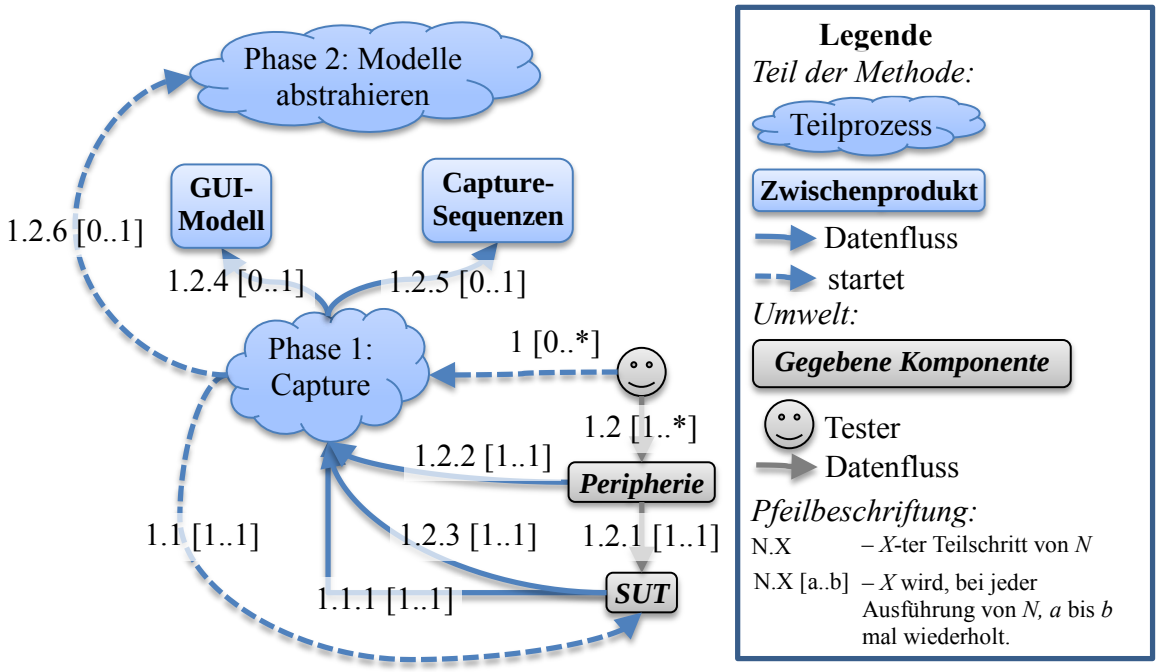


Abbildung 44: Überblick der einzelnen Interaktionen von Phase 1 „Capture“ und deren Reihenfolge.

Ein Überblick über die Reihenfolge der einzelnen Arbeitsschritte der Capture-Phase ist in Abbildung 44 zu sehen. Der Tester startet beliebig oft die Aufnahme von Capture-Sequenzen (1).[28] Mit der Aufnahme jeder Sequenz wird das SUT gestartet (1.1). Nach dem Starten wird die GUI des SUTs gescannt, um den Anfangszustand zu ermitteln (1.1.1). Der Tester tätigt während jeder Aufnahme beliebig viele Eingaben auf Tastatur und Maus (1.2). Jeder dieser Eingaben beeinflusst das SUT (1.2.1). Danach nutzt das Verfahren die gerade getätigte Eingabe (1.2.2) und liest die GUI aus (1.2.3). Das Verfahren fasst – mittels Heuristiken – mehrere Eingaben zu atomaren Aktionen zusammen. Wird durch eine Eingabe eine atomare Aktion vervollständigt, dann aktualisiert das Verfahren die Modelle von GUI (1.2.4) und Capture-Sequenz (1.2.5). Immer wenn die Modelle aktualisiert werden, dann startet das Verfahren den Teilprozess Phase 2 „Modelle abstrahieren“ (1.2.6). Nach dem Beenden einer Testsequenz kann der Tester die Aufnahme einer weiteren Capture-Sequenz beginnen (1).

[28] Die Nummern in diesem Absatz beziehen sich auf Abbildung 44.

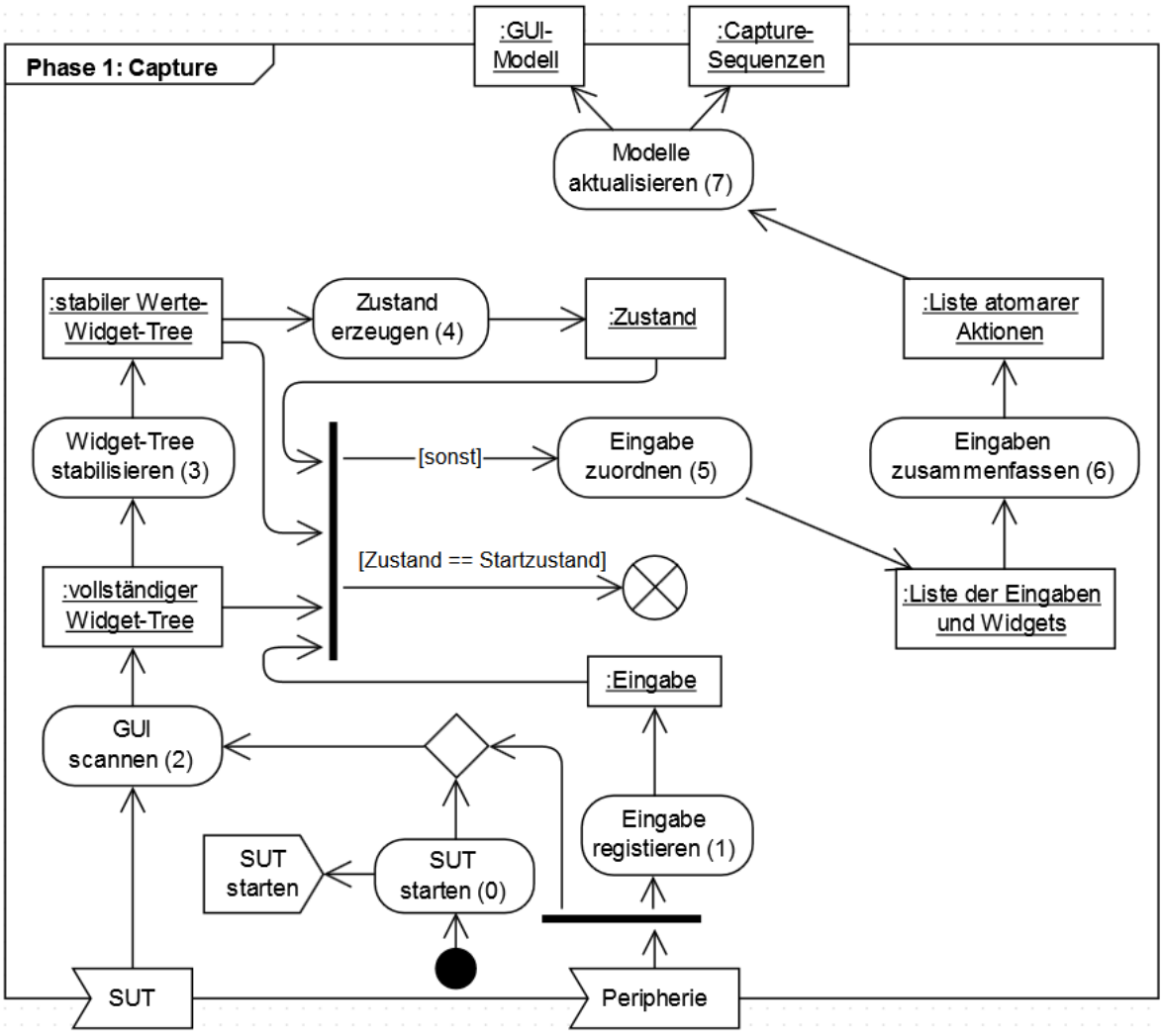


Abbildung 45: Überblick der Subaktivitäten und Zwischenprodukte von Phase 1 „Capture“ und deren Zusammenhänge. Einige Zusammenhänge wurden zur besseren Übersichtlichkeit vereinfacht dargestellt.

Abbildung 45 gibt einen Überblick der Subaktivitäten und Zwischenprodukte von Phase 1 „Capture“ und stellt die Zusammenhänge zwischen ihnen dar. Tätigt der Tester während der Aufnahme eine Eingabe, so wird diese registriert (1)[29] und der vollständige Widget-Tree der GUI ermittelt (2). Dieser wird in einen stabilen Widget-Tree umgewandelt (3). Zu dem stabilen Widget-Tree wird dann ein Zustand generiert (4). Der registrierten Eingabe wird ein Widget zugeordnet und das Eingabe-Widget-Paar wird einer Liste hinzugefügt (5). Danach wird eine Subaktivität angestoßen, um die bisherigen Eingaben – mittels Heuristiken – zu atomaren Aktion zusammenzufassen (6). Lassen sich die Eingaben zu mindestens einer neuen atomaren Aktion zusammenfassen, so werden die relevanten

[29] Die Nummern in diesem Absatz beziehen sich auf Abbildung 45.

Daten in die Modelle gespeichert (7). Der Teilprozess Phase 1 „Capture“ ist danach bereit, die nächste Eingabe zu verarbeiten.

Zur Ermittlung des Anfangszustands werden beim ersten Durchlauf durch Phase 1 das SUT gestartet (0) und die initialen Widget-Trees und der Anfangszustand ermittelt (2 bis 4), ohne auf ein Signal der Peripherie zu warten (1).

In den folgenden Unterabschnitten werden die einzelnen Subaktivitäten und die erzeugten Zwischenprodukte genauer erläutert.

4.3.2 Subaktivität „SUT starten“

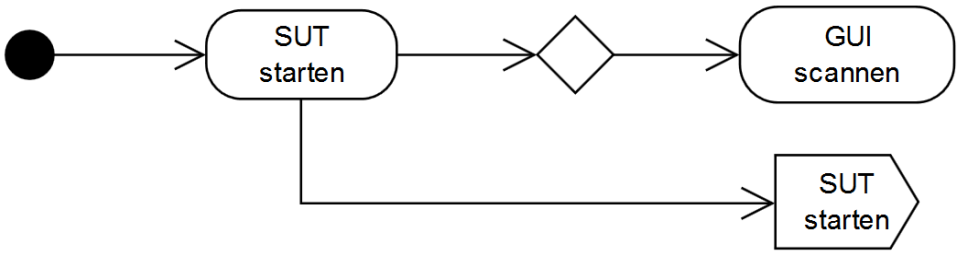


Abbildung 46: Phase 1 „Capture“ – Subaktivität „SUT starten“

Die Subaktivität „SUT starten“ (Abbildung 46) beginnt, sobald das Aufnehmen (Capture) einer Sequenz begonnen wurde. Diese Phase startet das SUT. Danach geht es weiter zur nächsten Subaktivität.

Diese Subaktivität beinhaltet gegebenenfalls auch das Zurücksetzen des SUTs auf einen konsistenten Startzustand. Dieses Zurücksetzen wird aufgrund des *Controllability Problems* (vgl. Abschnitt 1.5.4) nicht näher beschrieben.

Die Subaktivität „SUT starten“ ist konzeptionell einfach. Die komplexere technische Umsetzung wird später in Abschnitt 5.3.3 beschrieben.

4.3.2.1 Beispiel

Der Windows-Taschenrechner wird gestartet und ist auf dem Bildschirm sichtbar.

4.3.3 Subaktivität „Eingabe registrieren“

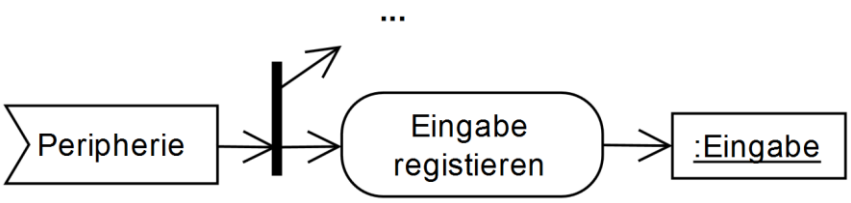


Abbildung 47: Phase 1 „Capture“ – Subaktivität „Eingabe registrieren“

Die Subaktivität „Eingabe registrieren“ (Abbildung 47) wird ausgelöst sobald eine Eingabe auf der Peripherie erfolgt. Sie nimmt die Nutzereingaben wahr und macht sie für den weiteren Prozess nutzbar.

Diese Subaktivität ist konzeptionell einfach. Die komplexere technische Umsetzung wird später in Abschnitt 5.3.3.2 beschrieben.

4.3.3.1 Beispiel

Die ausgeführten Eingaben werden mittels einfacher Bezeichner und ggf. mittels Koordinaten modelliert. Zwei Beispiele für solche Modelle sind

- Tastatureingabe „Strg“ + „S“ oder
- MouseDown der linken Maustaste über Bildschirmkoordinate (123, 456).

4.3.4 Subaktivität „GUI scannen“

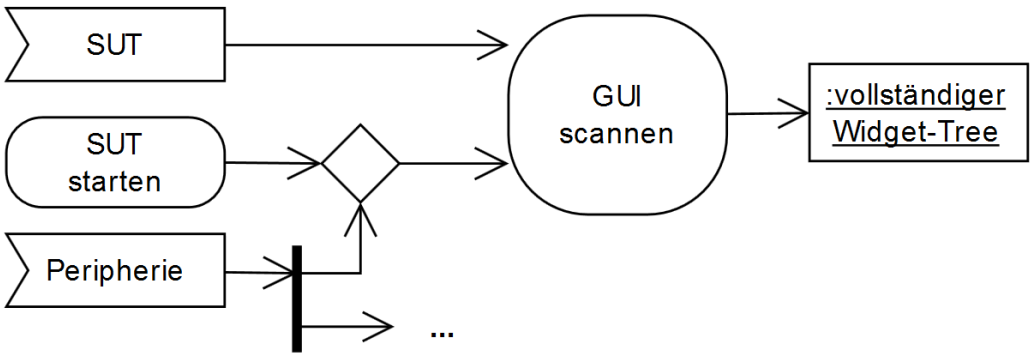


Abbildung 48: Phase 1 „Capture“ – Subaktivität „GUI scannen“.

Die Subaktivität „GUI scannen“ (Abbildung 48) wird ausgelöst, sobald eine Eingabe registriert wurde oder direkt nachdem das SUT gestartet wurde. Sie scannt die GUI und erzeugt einen vollständigen (instabilen) Widget-Tree (vgl. Abschnitt 2.4).

Diese Subaktivität ist konzeptionell einfach. Die komplexere technische Umsetzung wird später in Abschnitt 5.3.3.1 beschrieben.

4.3.4.1 Beispiel

Ein Beispiel für einen vollständigen Widget-Tree ist in Abbildung 13 (oben) zu sehen.

4.3.5 Subaktivität „Widget-Tree stabilisieren“

Abbildung 49: Phase 1 „Capture“ – Teilprozess „Widget-Tree stabilisieren“.

Die Subaktivität „Widget-Tree stabilisieren“ (Abbildung 49) erzeugt aus einem vollständigen (instabilen) Widget-Tree einen stabilen Werte-Widget-Tree oder identifiziert gegebenenfalls einen bereits vorhandenen (vgl. Abschnitt 2.4). Dazu werden alle instabilen Eigenschaften des vollständigen Widget-Trees weggelassen. Die vollständigen Widget-Trees werden mit ihren stabilen Werte-Widget-Trees assoziiert (Abbildung 50).

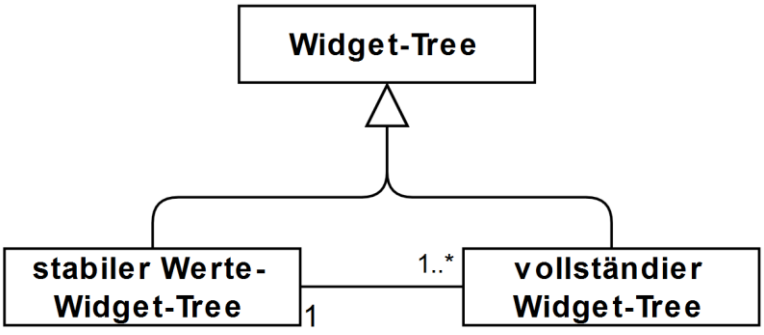


Abbildung 50: Nachdem die Subaktivität „Widget-Tree stabilisieren“ den stabilen Werte-Widget-Tree ermittelt hat, assoziiert sie ihn mit dem zugrundeliegenden vollständigen Widget-Tree.

Zusätzlich zum Weglassen von instabilen Eigenschaften werden die Widgets noch mit einer zusätzlichen Eigenschaft assoziiert: mit den *stabilen IDs*. Durch diese können sie

später bei der automatischen Testausführung (Phase 4) identifiziert werden. Zur Ermittlung dieser stabilen IDs sind verschiedene Strategien möglich, bei denen man zwischen Stabilität und Genauigkeit abwägen muss. Im Rahmen dieser Arbeit ist die stabile ID ein Hashwert, welcher die folgenden Informationen über ein Widget enthält: die stabile ID des Eltern-Widgets, den Namen des Widgets und den Typ des Widgets.

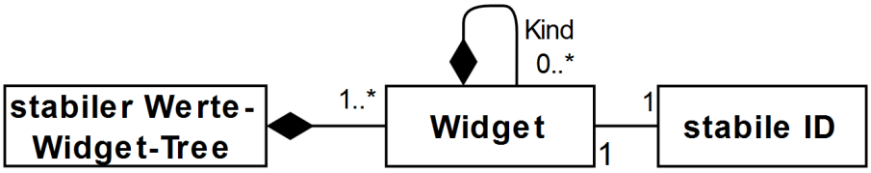


Abbildung 51: Die Subaktivität „Widget-Tree stabilisieren“ ordnet jedem Widget des stabilen Werte-Widget-Trees eine stabile ID zu.

4.3.5.1 Beispiel

Ein Beispiel eines stabilen Widget-Trees ist in Abbildung 13 (unten) zu sehen. Zusätzlich werden die Widgets aus Abbildung 13 (unten) in dieser Subaktivität noch mit jeweils einer *stabilen ID* assoziiert. So könnte der OK-Button beispielsweise die stabile ID „widget_474749874“ erhalten.

4.3.6 Subaktivität „Zustand erzeugen“

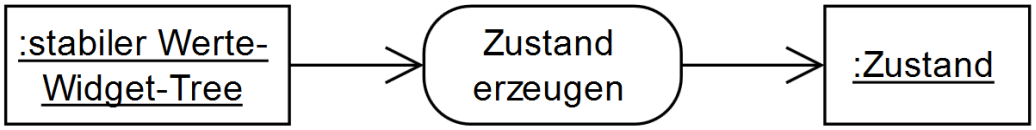


Abbildung 52: Phase 1 „Capture“ – Subaktivität „Zustand erzeugen“

Die Subaktivität „Zustand erzeugen“ (Abbildung 52) erzeugt zu einem stabilen Werte-Widget-Tree einen Zustand, oder identifiziert ggf. einen schon vorhandenen Zustand. Der Zustand wird mit dem stabilen Werte-Widget-Tree assoziiert (Abbildung 53).

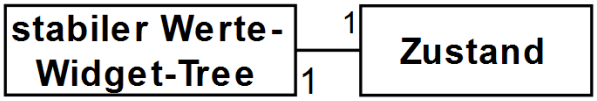


Abbildung 53: Subaktivität „Zustand erzeugen“ verlinkt den erzeugten Zustand mit dem stabilen Werte-Widget-Tree.

Dazu wird zum stabilen Werte-Widget-Tree eine ID ermittelt. Diese ID identifiziert einen eventuell vorhandenen Zustand oder erzeugt andernfalls einen neuen Zustand mit dieser ID.

Die ermittelte ID entspricht der Summe der rekursiv ermittelten Hashwerte der Serialisierung der Kindknoten des stabilen Werte-Widget-Trees (siehe Beispiel).[30]

4.3.6.1 Beispiel

Abbildung 54 zeigt beispielhaft einen stabilen Werte-Widget-Tree, aus dem ein Zustand erzeugt werden soll.

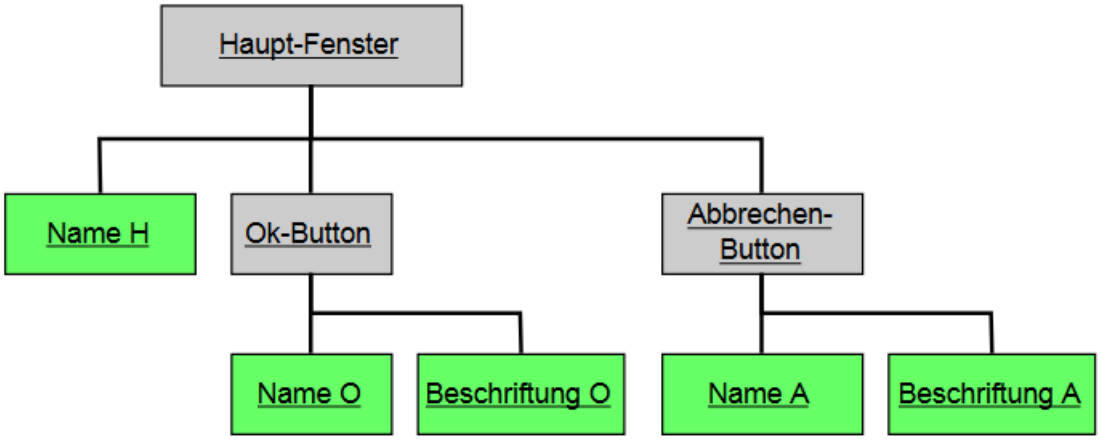


Abbildung 54: Beispiel: Widget-Tree w_1 aus dem ein Zustand erzeugt werden soll

Zuerst werden die Blattknoten serialisiert und ein Hashwert[31] ermittelt:

- Hash(Serialisiere(Name H)) = 1000
- Hash(Serialisiere(Name O)) = 2000
- Hash(Serialisiere(Beschriftung O)) = 3000

[30] Ähnlich machen es auch Bauersfeld und Vos [BaVo12a]. Sie weisen auch darauf hin, dass die Hashwerte theoretisch kollidieren können, es jedoch in der Praxis äußerst unwahrscheinlich ist. Bauersfeld und Vos serialisieren allerdings den gesamten Widget-Trees auf einmal, was nicht der Eigenschaft gerecht wird, dass Kindknoten kommutativ sind. Hier wäre es angemessener, die rekursiv ermittelten Hashwerte der Kindknoten zu addieren, wie wir es in dieser Arbeit tun.

[31] Die Hashwerte in diesem Beispiel sind fiktiv und dienen der besseren Anschauung. Echte Hashwerte schöpfen den kompletten Wertbereich aus und sind besser verteilt.

- Hash(Serialisiere(Name A)) = 4000
- Hash(Serialisiere(Beschriftung A)) = 5000

Dann werden rekursiv ihre Elternknoten (ohne Kindknoten) serialisiert und ein Hashwert aus Serialisierung und Serialisierung der addierten Hashwerte aller Kindknoten gebildet:

- Hash(Serialisiere(Ok-Button ohne Kindknoten) + Serialisiere(2000 + 3000)) = 600
- Hash(Serialisiere(Abbrechen-Button ohne Kindknoten) + Serialisiere(4000 + 5000)) = 700

Bis schließlich ein Hashwert zum Wurzelknoten ermittelt wird, welcher dann als ID fungiert:

- ID :=
  Hash(Serialisiere(Haupt-Fenster ohne Kindknoten) + Serialisiere(1000 + 600 + 700)) = 42

Existiert bereits ein Zustand mit dieser ID, dann kann der vorhandene Zustand und der vorhandene zugehörige Widget-Tree weiterverwendet werden. Existiert noch kein Zustand mit dieser ID, dann wird ein neuer Zustand erzeugt.

Der stabile Werte-Widget-Tree wird mit dem ermittelten Zustand assoziiert, wodurch auch alle zugehörigen vollständigen Widget-Trees indirekt damit verbunden sind (Abbildung 55).

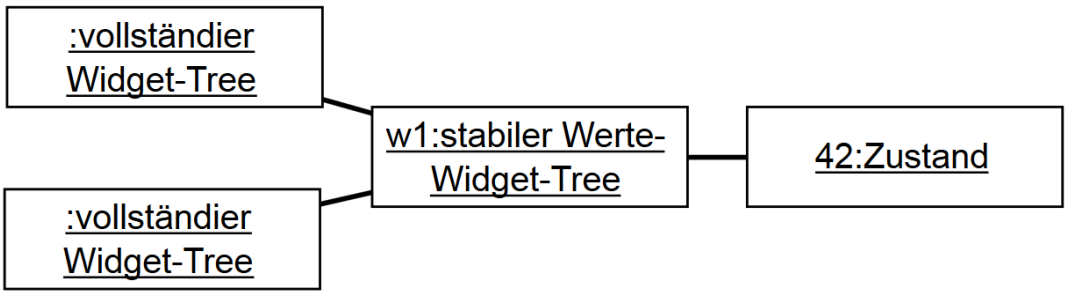


Abbildung 55: Beispiel: Die in Subaktivität „Zustand erzeugen“ ermittelten Zustände werden mit ihren stabilen Werte-Widget-Trees verlinkt und dadurch indirekt auch mit allen zugehörigen vollständigen Widget-Trees.

4.3.7 Subaktivität „Eingabe zuordnen“

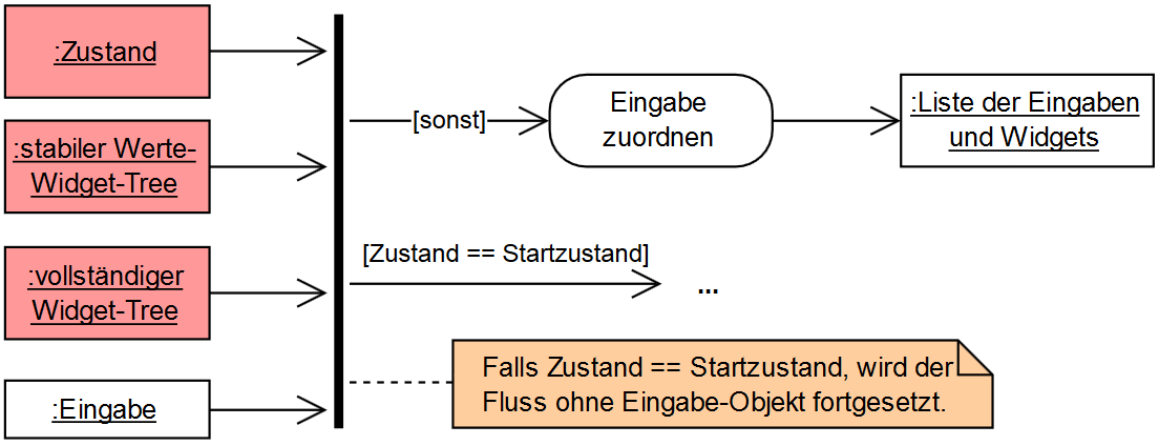


Abbildung 56: Phase 1 „Capture“ – Subaktivität „Eingabe zuordnen“. Die rot markierten Objekte stammen aus der vorherigen Iteration von Phase I, so dass die Eingabe ihrem Ausgangszustand und nicht dem Folgezustand zugeordnet wird. Das Zusammenspiel zwischen verschiedenen Iterationen wurde zur besseren Übersichtlichkeit in Abbildung 45 nicht dargestellt.

Der Teilprozess „Eingabe zuordnen“ (Abbildung 56) ordnet einer Eingabe das zugehörige Widget zu. Dazu wird ein Widget in einem stabilen Werte-Widget-Tree referenziert. Der Widget-Tree selbst wird über den Zustand referenziert. Die Eingabe-Widget-Paare werden einer Liste hinzugefügt. Die erzeugte Liste enthält dann Einträge der Art *<Eingabe> über <Widget> in Zustand <Zustand>* (vgl. Abbildung 57).

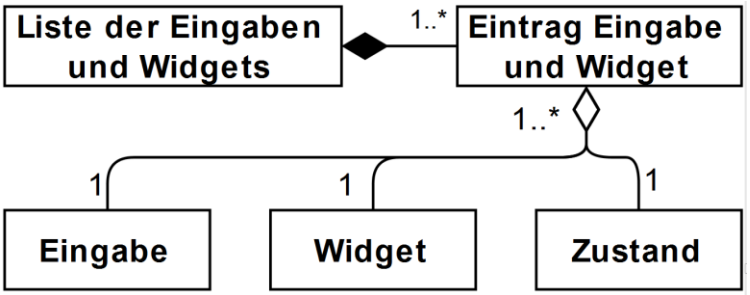


Abbildung 57: Subaktivität „Eingabe zuordnen“ erzeugt Listen-Einträge welche Eingaben, Widgets und Zustände verknüpfen.

Mauseingaben werden über die Bildschirmkoordinaten mit den Widgets assoziiert. Tastatureingaben haben keine Bildschirmkoordinaten und werden deshalb mit dem Widget assoziiert, welches im vollständigen Widget-Tree den Fokus hat.[32]

Diese Subaktivität wird nicht in der ersten Iteration von Phase 1 „Capture“ ausgeführt, da in dieser Iteration nur der Anfangszustand ermittelt wird und noch keine Eingabe vorliegt.

4.3.7.1 Beispiel

Beispielsweise könnten diese Informationen als Eingabe dienen (siehe auch Abbildung 58):

- *MouseDown* der linken Maustaste über Bildschirmkoordinate *(123, 456)*.
- Widget *OK-Button (instabil)* im vollständigen Widget-Tree v_1 umschließt die Koordinaten *(123, 456).*
- Der stabile Widget-Tree w_1 zum Zustand *42* ist mit dem vollständigen Widget-Tree v_1 assoziiert.
    - Das Widget *Ok-Button (instabil)* ist dem Widget *Ok-Button* aus dem stabilen Widget-Trees w_1 zugeordnet.

[32] Fokus ist – in dieser Arbeit – eine instabile Eigenschaft eines Widgets. Mögliche Werte sind vom Typ Boolean.

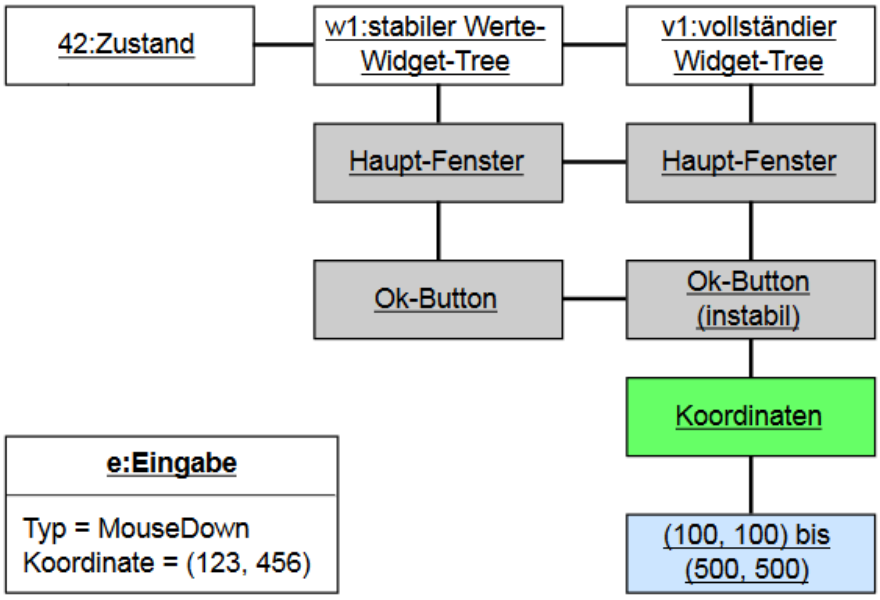


Abbildung 58: Die gegeben Objekte, welche von Subaktivität „Eingabe zuordnen“ genutzt werden, könnten aussehen wie in der Darstellung. Man sieht auch wie die Objekte, teils indirekt, miteinander verlinkt sind.

Diese Informationen werden in der Liste mit dem Eintrag *MouseDown über OK-Button in Zustand 42* zusammengefasst (vgl. Abbildung 59).

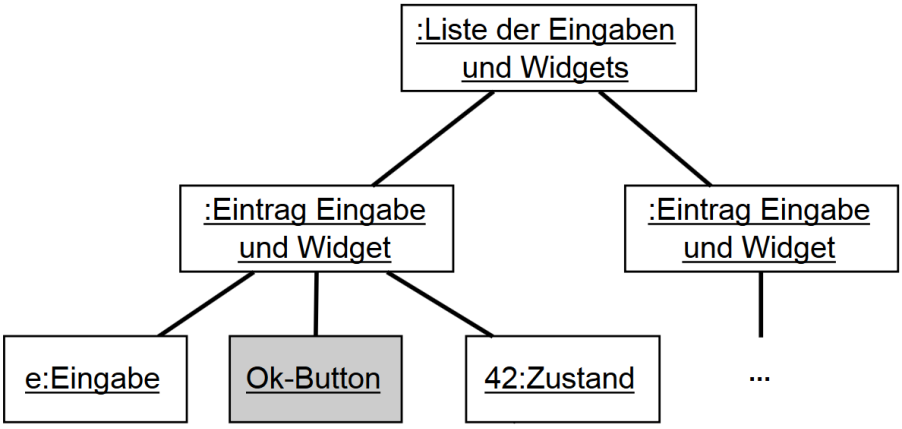


Abbildung 59: Beispielergebnis von Subaktivität „Eingabe zuordnen“

4.3.8 Subaktivität „Eingaben zusammenfassen“

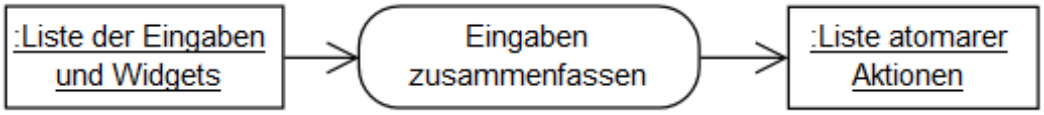


Abbildung 60: Phase 1 „Capture“ – Subaktivität „Eingaben zusammenfassen“.

Die Subaktivität „Eingaben zusammenfassen“ (Abbildung 60) fasst die einzelnen Nutzereingaben zu *atomaren Aktionen* zusammen. Atomare Aktionen sind dabei ähnlich den *Einträgen mit Eingaben und Widgets*, allerdings sind Aktionen grobgranularer als Eingaben. Die erzeugte Liste enthält dann Einträge der Art *<Aktion> über*[33] *<Widget> in Zustand <Zustand>* (vgl. Abbildung 61).

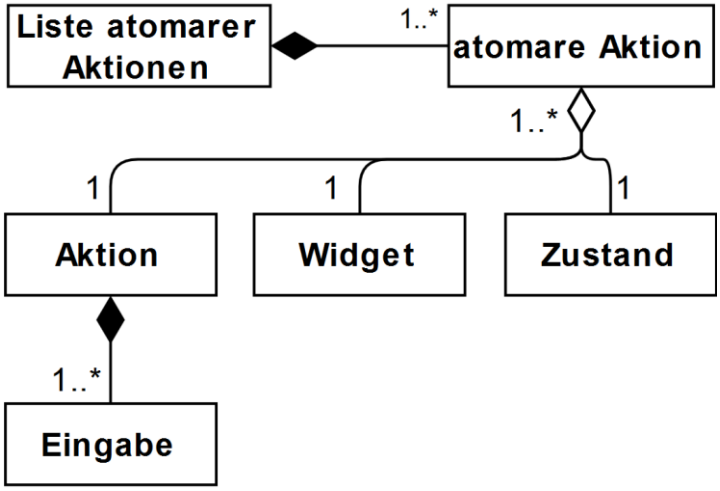


Abbildung 61: Atomare Aktionen fassen mehrere Eingaben zusammen, wenn sie auf einem Widget und in einem Zustand ausgeführt werden.

Die Details dieser Subaktivität werden im Rahmen dieser Arbeit nur informell beschrieben. Es wird stets eine Sequenz von Eingaben des aktuellen Capture-Prozesses betrachtet. Ein Ziel ist es, auch in unfertigen Eingabesequenzen die ausgeführten atomaren Aktionen möglichst früh zu erkennen. Dazu folgt das Zusammenfassen von Eingabesequenzen festen Zuordnungsvorschriften. Die in dieser Arbeit verwendeten Vorschriften sind Tabelle 2 zu entnehmen. Eine längere Liste oder formellere Zuordnung (mittels formaler Sprachen) ist denkbar, wird aber im Rahmen dieser Arbeit nicht entworfen.

[33] Die Präposition wird später angepasst, um den Text lesbarer zu machen. Zum Beispiel wird „Klick über Button A“ zu „Klick auf Button A“.

Aktion	**Folge von Eingaben**	**Mögliche Widgets**
<elementare Benutzer Aktion>	1. <beliebig>	Alle
Klick	1. MouseDown 2. MouseUp	Alle
Tippe <Text>	1. -> Klick 2. Tippe Buchstabe, Ziffer oder Sonderzeichen beliebig oft.	Textfeld
Ersetze Text durch <Text>	1. -> Klick 2. Strg + A 3. Tippe Buchstabe, Ziffer oder Sonderzeichen beliebig oft.	Textfeld

Tabelle 2: Die Zuordnungsvorschriften zum Zusammenfassen elementarer Benutzereingaben zu Aktionen.

Will man einer (unvollständigen) Sequenz von Eingaben eine Sequenz von atomaren Aktionen zuordnen, so muss man entscheiden, welche Zuordnungsvorschriften genommen werden. Diese Entscheidung wird durch dieses Vorgehen getroffen:

- Es wird unterschieden zwischen „nicht zugeordneten Eingaben“, welchen noch keine Aktion zugeordnet wurde, und „zugeordneten Eingaben“, welche bereits Teil einer atomaren Aktion sind.
- Es wird stets nur die Teilsequenz der nicht zugeordneten Eingaben betrachtet und Aktionen daraus geformt. Wurde eine Eingabe einmal zugeordnet, so wird sie keiner anderen Aktion mehr zugeordnet.
- Die getätigten Eingaben $e_1, \dots, e_i$ werden solange nicht zugeordnet, wie theoretisch noch Eingaben $e_j, \dots, e_k$ folgen könnten, so dass eine Zuordnungsvorschrift für die Eingaben $e_1, \dots, e_i, e_j, \dots, e_k$ existiert.

- Die getätigten Eingaben $e_1, \dots, e_i$ werden zugeordnet, sobald eine Eingabe e_j erfolgt und theoretisch keine Eingaben $e_k, \dots, e_l$ mehr folgen können, so dass eine Zuordnungsvorschrift für die Eingaben $e_1, \dots, e_i, e_j, e_k, \dots, e_l$ existiert.
- Sobald eine Eingabe auf einem anderen Widget als die vorherige Eingabe stattfindet, beginnt auch eine neue atomare Aktion. Das heißt alle vorherigen Eingaben, die bisher nicht zugeordneten wurden, werden nun zugeordnet.
- Wurde entschieden, dass einem Eingabesequenzteil $e_1, \dots, e_k$ (mit fester Länge) Aktionen zugeordnet werden, so wird „greedy“ vorgegangen. Das heißt es wird mit der ersten Eingabe begonnen und die Aktion gewählt, welche die meisten Eingaben zusammenfassen kann. Analog werden die übrigen Eingaben (das heißt der Postfix der Eingabesequenz) rekursiv zusammengefasst, bis keine Eingabe mehr übrig ist.

4.3.8.1 Beispiel

Tabelle 3 zeigt beispielhaft, welche Elemente einer Liste von Eingaben zu welchen atomaren Aktionen zusammengefasst werden.

Eingabe	**Widget**	**Widget-Tree**	**Atomare Aktionen**
MouseDown	über Button A	in Zustand 42	Klick auf Button A in Zustand 42
MouseUp	über Button A	in Zustand 43	
MouseDown	über Textfeld B	in Zustand 44	Tippe “Hi” in Textfeld B in Zustand 44
MouseUp	über Textfeld B	in Zustand 45	
Tippe “H”	über Textfeld B	in Zustand 46	
Tippe “i”	über Textfeld B	in Zustand 47	
MouseDown	über Button C	in Zustand 48	Vorerst keine atomare Aktion

Tabelle 3: Beispiel: Zusammenfassung von Eingaben zu atomaren Aktionen.

Das Vorgehen zur Herleitung der atomaren Aktionen soll zum besseren Verständnis an einem zweiten Beispiel demonstriert werden. Dieses Beispiel benutzt die Zuordnungsvorschriften aus Tabelle 2. Im Beispiel wird ein Textfeld A angeklickt, „Max“ getippt und danach auf den OK-Button geklickt. Tabelle 4 zeigt welche Aktionen nach jeder Eingabe zugeordnet wurden und welche Aktionen für die noch nicht zugeordneten Eingaben in Frage kommen.

#	Eingabe	Widget	Potentielle Aktionen nach der Eingabe (A) = abgeschlossen	Gewählte atomare Aktionen
1	MouseDown	Textfeld A	• <elementare Benutzer Aktion> (A) • Klick • Tippe <Text> • Ersetze Text durch <Text>	• keine
2	MouseUp	Textfeld A	• Klick (A) • Tippe <Text> • Ersetze Text durch <Text>	• keine
3	Tippe „M“	Textfeld A	• Tippe “M” (A)	• keine
4	Tippe “a”	Textfeld A	• Tippe “Ma” (A)	• keine
5	Tippe “x”	Textfeld A	• Tippe “Max” (A)	• keine
6	MouseDown	OK-Button	• <elementare Benutzer Aktion> (A) • Klick	• Tippe “Max” in Textfeld A
7	MouseUp	OK-Button	• Klick (A)	• Tippe “Max” in Name-Textfeld
8	beliebig	Auf einem dritten Widget	• …	• Tippe “Max” in Name-Textfeld • Klick auf OK-Button

Tabelle 4: Beispiel: Einzelne Schritte des Zusammenfassens von Eingaben zu atomaren Aktionen.

4.3.9 Subaktivität „Modelle aktualisieren“

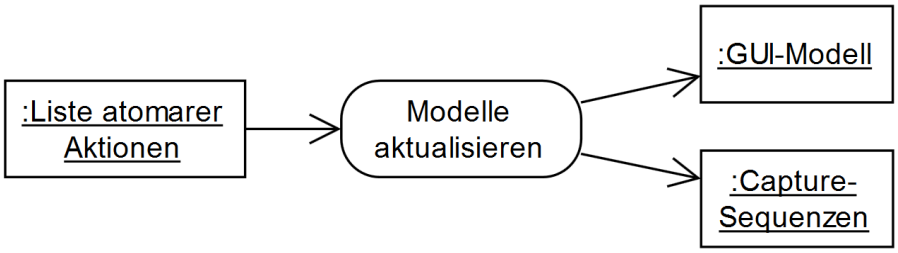


Abbildung 62: Phase 1 „Capture“ – Subaktivität „Modelle aktualisieren“.

Immer wenn eine neue atomare Aktion aus den Eingaben zusammengefasst wurde (vgl. Abschnitt 4.3.8), dann wird die Subaktivität „Modelle aktualisieren“ (Abbildung 62) gestartet. Diese Subaktivität strukturiert die vorhandenen Daten im Wesentlichen nur neu, um die Capture-Sequenzen (vgl. Abschnitt 4.2.2) und das GUI-Modell (vgl. Abschnitt 4.2.1) zu aktualisieren.

Der *Zustand* und der zugehörige *stabile Werte-Widget-Tree* der *atomaren Aktion* werden unverändert ins *GUI-Modell* übernommen, falls sie dort noch nicht vorhanden sind. In jedem Fall wird ein *durchlaufener Zustand* in der *Capture-Sequenz* angelegt. Eine Assoziation zwischen *durchlaufenem Zustand* und *Zustand* wird hergestellt.

Die zuvor ermittelten atomaren Aktionen werden als *Transition* im *klassischen Zustandsautomaten* des *GUI-Modells* gespeichert, falls noch nicht vorhanden. Eine Referenz auf diese *Transition* wird als *ausgeführte Aktion* zur aktuellen *Capture-Sequenz* hinzugefügt.

Die Ausgangs- und Folgezustände werden entsprechend der gerade ausgeführten Capture-Sequenz mit den Transitionen verknüpft.

4.3.9.1 Beispiel

Ein Beispiel für ein GUI-Modell befindet sich in Abschnitt 4.2.1.1. Ein Beispiel für Capture-Sequenzen befindet sich in Abschnitt 4.2.2.1.

4.3.10 Zusammenfassung von Phase 1

Phase 1 „Capture“ adaptiert den Capture-Mechanismus, um ein systemnahes einfaches Modell der GUI zu erstellen. Dazu startet der Tester für jede Capture-Sequenz den automatischen Prozess von Phase 1 und bedient dann über die gewohnte Peripherie das GUI-System. Im Gegensatz zur klassischen Capture/Replay-Methode wird daraus nicht sofort eine auszuführende Testsequenz erzeugt, sondern ein Modell. Das Modell besteht aus einem einfachen Zustandsautomaten des SUTs. Die aufgenommenen Sequenzen werden als Pfade durch den Zustandsautomaten modelliert. Wie diese Modelle aus den Nutzereingaben ermittelt werden, wurde in diesem Abschnitt beschrieben.

Da einfache Zustandsautomaten für komplexe Systeme schlecht wartbar sind, sieht das angestrebte Zielmodell des SUTs neben dem klassischen Zustandsautomaten noch weitere Modellelemente und -untergliederungen vor. Dazu wird das systemnahe einfache *GUI-Modell* weiter abstrahiert und ein *abstrahiertes GUI-Modell* daraus entwickelt. Dieser Abstraktionsprozess wird im nächsten Abschnitt beschrieben.

4.4 Phase 2 „Modelle abstrahieren“

Der Teilprozess Phase 2 „Modelle abstrahieren“ wird automatisch ausgelöst sobald Phase 1 abgeschlossen ist. Er wandelt das GUI-Modell und die Capture-Sequenzen in ein *abstrahiertes GUI-Modell* (vgl. Abschnitte 4.2.1, 4.2.2 und 4.2.3) um. In den folgenden Unterabschnitten wird dieser Prozess detailliert beschrieben.

4.4.1 Überblick

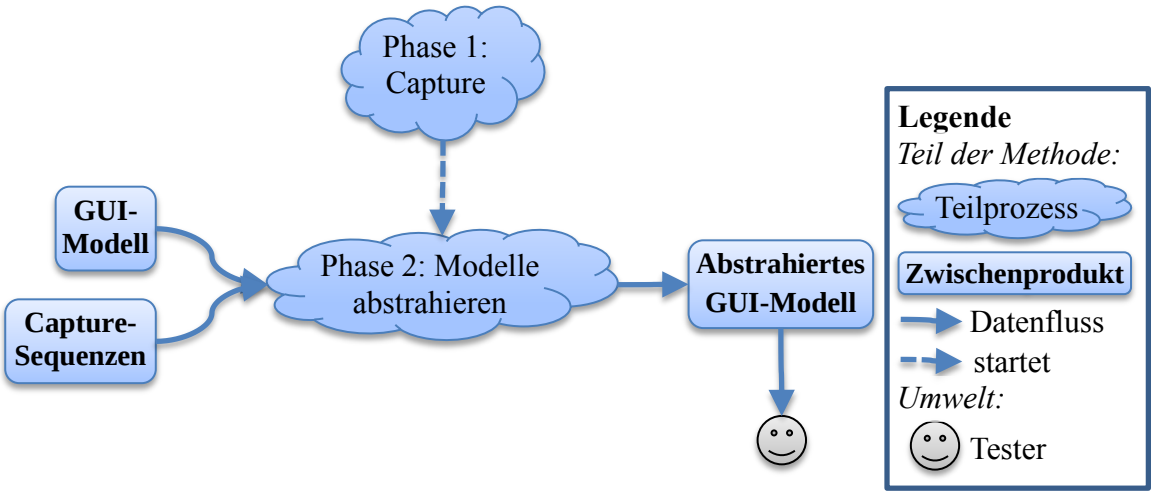


Abbildung 63: Überblick der Ein- und Ausgaben von Phase 2 „Modelle abstrahieren“.

Die bisherigen Modelle (GUI-Modell und Capture-Sequenzen) wären aufgrund ihres geringen Abstraktionsgrads nur schwer vom Tester zu beherrschen. Außerdem haben die Modelle noch einige Schwächen: Dem GUI-Modell fehlt die Historie und den Capture-Sequenzen fehlt ein Zusammenhang zwischen den Sequenzen und die Verzweigungen (vgl. Abschnitte 4.2.1 und 4.2.2). Der Teilprozess Phase 2 „Modelle abstrahieren“ (Abbildung 63) nutzt Heuristiken, um die Modelle so zusammenzuführen, dass ihre bisherigen Nachteile verschwinden. Außerdem werden sie derart abstrahiert, dass der Tester das resultierende Modell beherrschen kann. Das resultierende *abstrahierte GUI-Modell* besteht aus einem Klassifikationsbaum, einem Zustandsautomaten und einer Menge von Widget-Trees (vgl. Abschnitt 4.2.3).

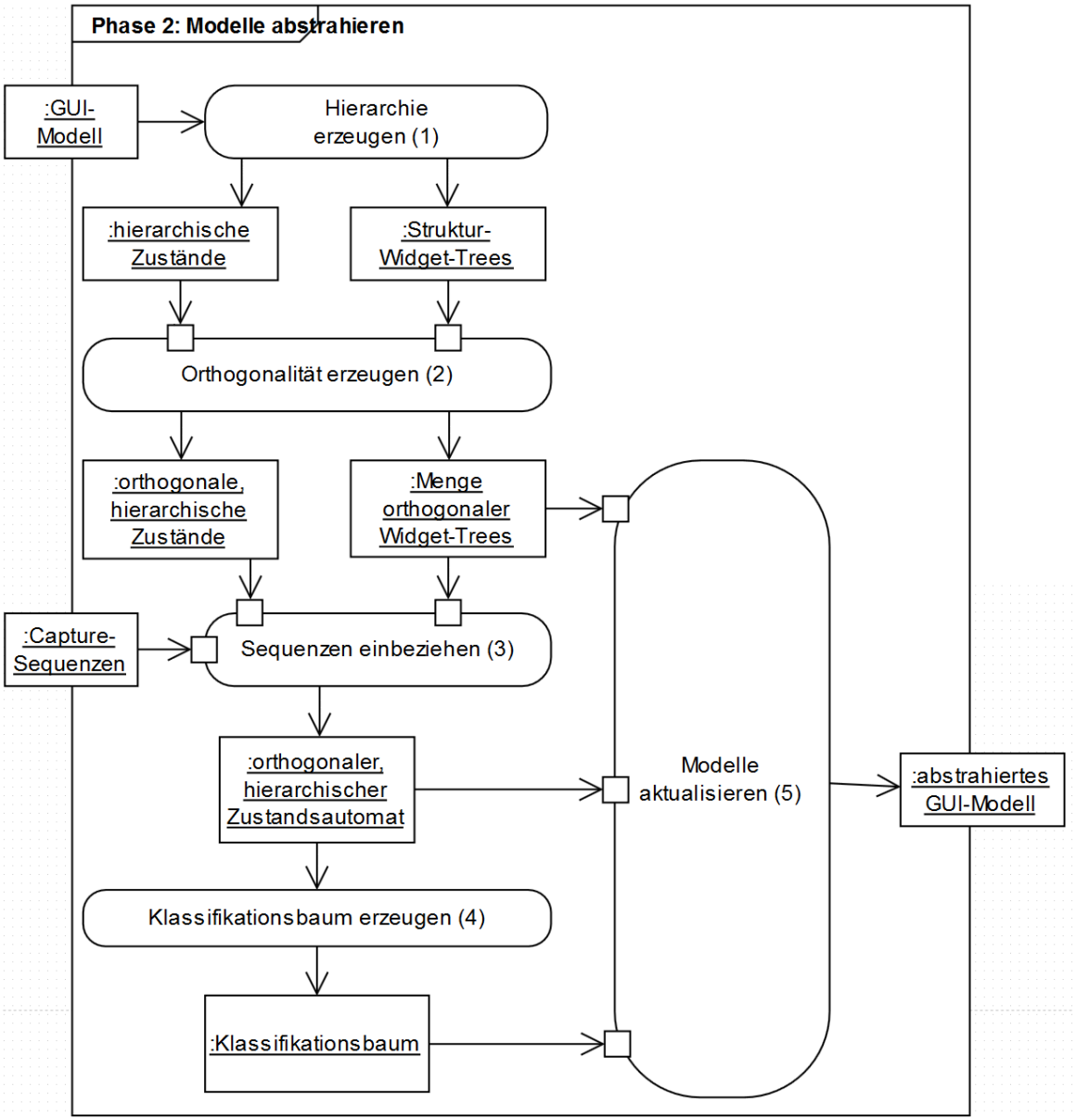


Abbildung 64: Überblick der Subaktivitäten und Zwischenprodukte von Phase 2 „Modelle abstrahieren“ und deren Zusammenhänge.

Abbildung 64 gibt einen Überblick über die einzelnen Subaktivitäten von Phase 2 „Modelle abstrahieren“. Zuerst werden die Zustände des GUI-Modells hierarchisch in neuen Überzuständen angeordnet und Struktur-Widget-Trees ermittelt (1). Diese hierarchischen Zustände werden danach in orthogonalen Bereichen aufgeteilt und zusammengeführt und die Widget-Trees werden orthogonal zergliedert (2). Als nächstes werden die orthogonalen, hierarchischen Zustände mithilfe der Capture-Sequenzen verbunden (3). Dabei entstehen Transitionen mit Events und Aktionen. Aus dem so abstrahierten Zustandsautomat wird ein Klassifikationsbaum erzeugt (4). Die so entstehenden Modelle

werden dem Tester und dem nächsten Teilprozess Phase 3 „Testsequenzen generieren“ bereitgestellt (5).

In den folgenden Unterabschnitten werden die einzelnen Subaktivitäten und die erzeugten Zwischenprodukte genauer erläutert.

4.4.2 Subaktivität „Hierarchie erzeugen“

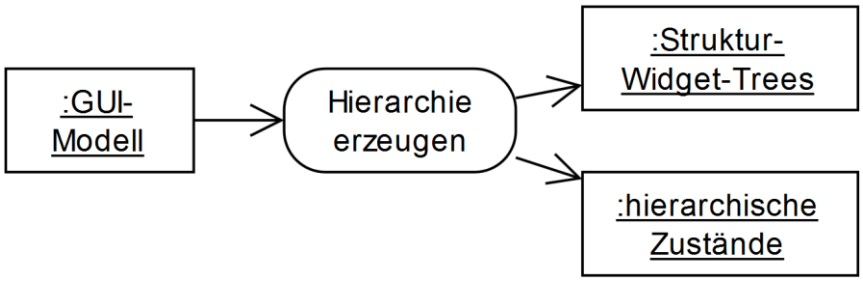


Abbildung 65: Phase 2 „Modelle abstrahieren“ – Subaktivität „Hierarchie erzeugen“

In dieser Subaktivität (siehe Abbildung 65) werden die Zustände aus dem GUI-Modell in übergeordneten Zuständen hierarchisch zusammengefasst (vgl. Abschnitt 4.2.3.3). Außerdem werden zu den gegebenen Werte-Widget-Trees die Struktur-Widget-Trees[34] ermittelt und ihnen zugeordnet. Die Transitionen werden in dieser Subaktivität ignoriert. Die Strukturierung erfolgt in drei Hierarchieebenen: Fenster-, Struktur- und Werte-Zustand. Das resultierende Modell der hierarchischen Zustände ist eine Sammlung der ermittelten Fenster-Zustände.

Um aus den gegebenen Zuständen und Werte-Widget-Trees eine Hierarchie zu erzeugen werden folgende Schritte ausgeführt:

1) Zu jedem Werte-Widget-Tree wird ein Struktur-Widget-Tree ermittelt und – falls noch nicht vorhanden – erzeugt (vgl. Abschnitt 4.2.3.1). Die Werte-Widget-Trees werden mit ihren Struktur-Widget-Trees assoziiert.
2) Zu jedem Struktur-Widget-Tree wird ein Struktur-Zustand erzeugt. Diese werden miteinander assoziiert. Die Werte-Zustände werden analog zu ihren assoziierten

[34] Die Unterscheidung zwischen stabilen und instabilen Widget-Trees spielte nur in Phase 1 „Capture“ eine Rolle. In allen anderen Phasen werden ausschließlich stabile Widget-Trees genutzt, dies wird ab hier nicht mehr explizit erwähnt.

Widget-Trees als Unterzustände des entsprechenden Struktur-Zustands gruppiert.

3) Zu jedem Struktur-Zustand wird der Name des aktiven modalen Fensters aus dem assoziierten Widget-Tree ermittelt. Falls noch nicht vorhanden, wird ein Fenster-Zustand mit diesem Namen erzeugt. Der Struktur-Zustand wird dem Fenster-Zustand mit diesem Namen als Unterzustand zugeordnet.

4.4.2.1 Beispiel

Ein Beispiel für so entstehende hierarchische Zustände ist in Abbildung 34 (Seite 67) zu sehen. Abbildung 30 (Seite 62) zeigt ein anderes Beispiel für Werte- und Struktur-Widget-Trees.

4.4.3 Subaktivität „Orthogonalität erzeugen“

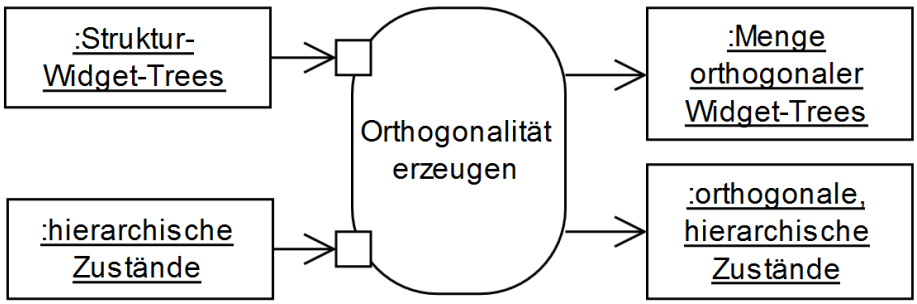


Abbildung 66: Phase 2 „Modelle abstrahieren“ – Subaktivität „Orthogonalität erzeugen“

Sobald die hierarchischen Zustände und Struktur-Widget-Trees vorliegen, werden diese orthogonal zergliedert (siehe Abbildung 66). Dabei wird angenommen, dass bestimmte Widgets (zum Beispiel Kontextmenüs) oder Eigenschaften (zum Beispiel „checked“ einer Checkbox) ein Verhalten orthogonal zum restlichen Verhalten der GUI aufweisen.

Eine Herausforderung dieses Schrittes ist, dass sowohl Harels [Hare87] Statecharts als auch die hier entwickelten orthogonalen Widget-Trees eine Transformation in die umgekehrte Richtung definieren. Beide Definitionen nehmen eine orthogonales Modell als gegeben und definieren dann, wie zugehöriges nichtorthogonales Modell aussieht. Diese Subaktivität muss dagegen eine Menge nichtorthogonaler Modelle orthogonalisieren, was einige Heuristiken und Annahmen erfordert.

Die Details dieser Subaktivität werden in den folgenden Unterabschnitten genauer beschrieben.

4.4.3.1 Überblick „Orthogonalität erzeugen“

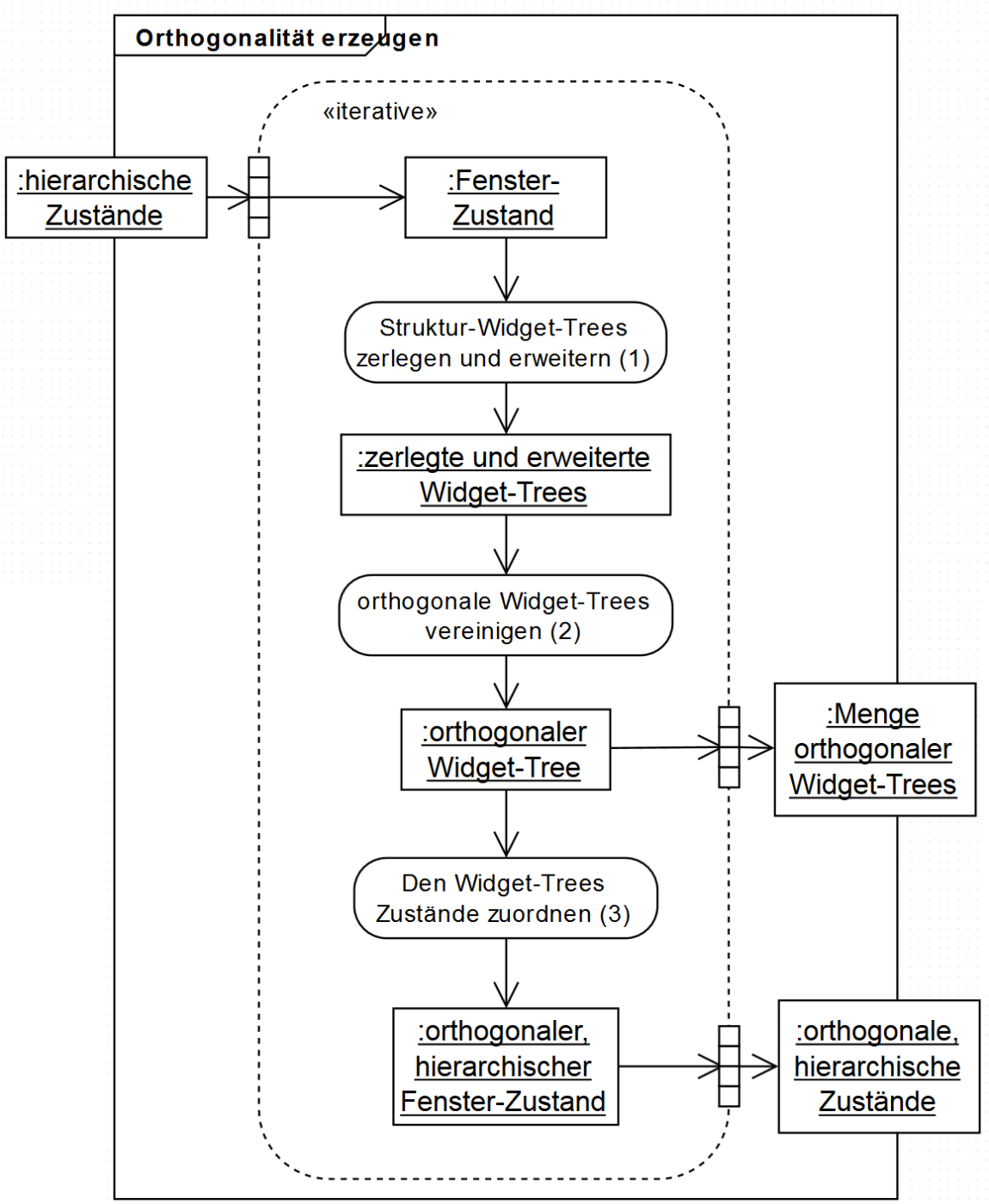


Abbildung 67: Überblick der Subaktivitäten und Zwischenprodukte von „Orthogonalität erzeugen“ und deren Zusammenhänge.

Um die hierarchischen Zustände und deren Widget-Trees orthogonal zu zergliedern, werden folgende Schritte für jeden einzelnen Fenster-Zustand durchgeführt. Zuerst werden alle Widget-Trees des Fenster-Zustands in orthogonale Faktoren zerlegt (1). Dabei werden auch Verallgemeinerungen getroffen und die Widget-Trees um neue Faktoren erweitert. Ähnliche Faktoren der so entstandenen, zerlegten und erweiterten Widget-Trees werden im nächsten Schritt teilweise wieder vereinigt (2), wodurch ein orthogonaler Widget-Tree entsteht. Zu diesem orthogonalen Widget-Tree werden nun Zustände generiert (3), welche alle Teil eines orthogonalen, hierarchischen Fenster-Zustands sind.

Auf diese Weise entstehen für jeden gegebenen Fenster-Zustand ein neuer orthogonaler Fenster-Zustand und ein zugehöriger orthogonaler Widget-Tree. Die Menge all dieser orthogonalen Fenster-Zustände und orthogonalen Widget-Trees ist das Zwischenergebnis, welches von der nächsten Aktivität weiterverwendet werden kann.

Wie diese Subaktivitäten im Detail aussehen, ist in den folgenden Unterabschnitten beschrieben.

4.4.3.2 Subaktivität „Struktur-Widget-Trees zerlegen und erweitern“

In dieser Subaktivität geht es darum, zu jedem klassischen Widget-Tree (vgl. Abschnitt 2.4) des gegebenen Fenster-Zustands und seiner Unterzustände einen orthogonalen Widget-Tree (vgl. Abschnitt 4.2.3.2) zu finden, welcher den gegebenen Widget-Tree repräsentiert.

Bei dieser Subaktivität findet eine Verallgemeinerung statt, durch welche sich die Faktoren der neuen orthogonalen Widget-Trees vielfältig kombinieren lassen. Daher repräsentieren die resultierenden orthogonalen Widget-Trees mehr klassische Widget-Trees als nur den zerlegten gegebenen Widget-Tree.

Vorgehen

Zuerst werden die Struktur-Widget-Trees nach bestimmten Unterbäumen durchsucht, welche als eigene Bereiche ausgegliedert werden sollen. Die Faktoren der ausgegliedert-

ten Bereiche werden dann rekursiv[35] selbst wieder nach auszugliedernden Bereichen durchsucht.

Welche Unterbäume ausgegliedert werden sollen hängt von der verwendeten Heuristik ab. Die in dieser Arbeit verwendeten Regeln sind in Tabelle 5 zu sehen. Eine umfassendere Regelsammlung ist zwar denkbar, wird jedoch im Rahmen dieser Machbarkeitsuntersuchung nicht weiter betrachtet.

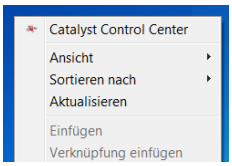
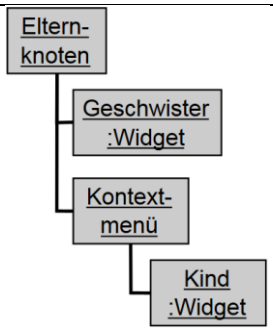
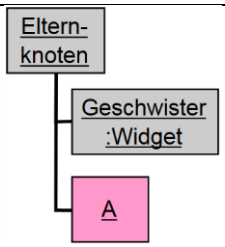
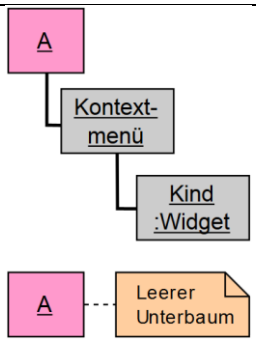
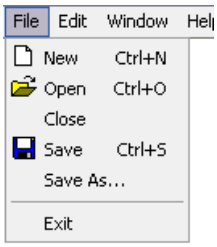
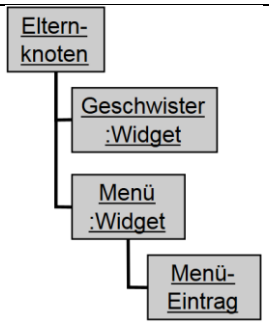
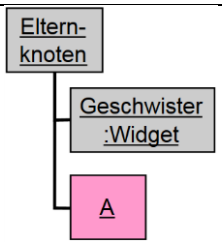
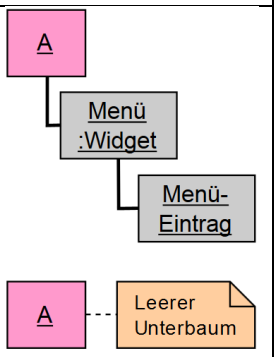
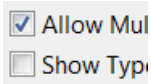
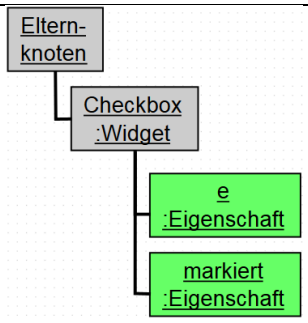
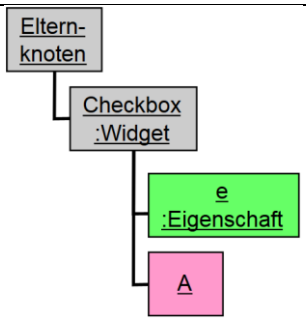
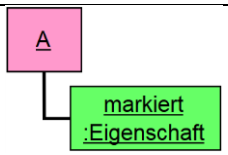
Widget	**Ausschnitt ursprünglicher Struktur-Widget-Tree**	**Resultierende Faktoren**	
		Bereich A:	**Bereich B:**
Kontextmenü 			
Menü 			
Checkbox 			

Tabelle 5: Verwendete Regeln zum Ausgliedern von Faktoren aus Widget-Trees.

[35] Dabei wird der oberste ausgegliederte Knoten übersprungen, um unendliche Wiederholungen zu vermeiden.

Immer wenn ein Struktur-Widget-Tree so in einzelne Faktoren zerlegt wird, dann werden die ihnen zugeordneten Werte-Widget-Trees gleichermaßen in Werte-Widget-Tree-Faktoren zerlegt. Dadurch ist jeder Werte-Widget-Tree-Faktor einem Struktur-Widget-Tree-Faktor zugeordnet, welcher genau dem Werte-Widget-Tree-Faktor ohne die Werte-Knoten entspricht.

Beispiel

Als Beispiel soll ein Ausschnitt des Menüs zum Windows-Taschenrechner dienen (siehe Abbildung 68), welcher durch die Widget-Trees aus Abbildung 69 repräsentiert wird.

Wendet man die Regeln aus Tabelle 5 rekursiv auf den Struktur-Widget-Tree (A) an, so erhält man die fünf Widget-Tree-Faktoren aus Abbildung 70. Hier findet bereits die erste Verallgemeinerung statt. Denn obwohl ursprünglich nur ein mittels Capture aufgenommener Struktur-Widget-Tree gegeben war, repräsentieren die fünf Faktoren bereits drei verschiedene Struktur-Widget-Trees. Die fünf Faktoren lassen sich folgendermaßen kombinieren:

- (1)-(2)-(4): ursprünglicher Struktur-Widget-Tree, beide Menüs geöffnet,
- (1)-(2)-(5): nur das erste Menü ist geöffnet,
- (1)-(3): kein Menü ist geöffnet.

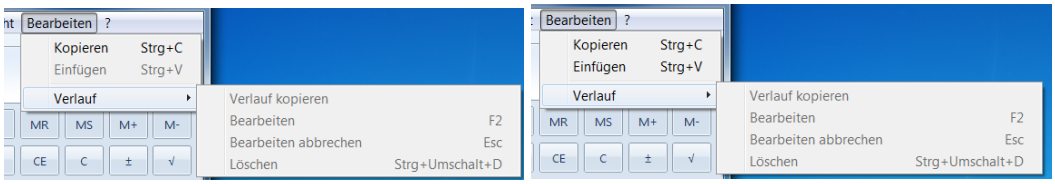


Abbildung 68: Beispiel eines zweistufigen Menüs des Windows-Taschenrechners. Der Menüeintrag „Einfügen“ kann zwei verschiedene Werte für die Eigenschaft „aktiviert“ annehmen: Wahr (rechts) und Unwahr (links).

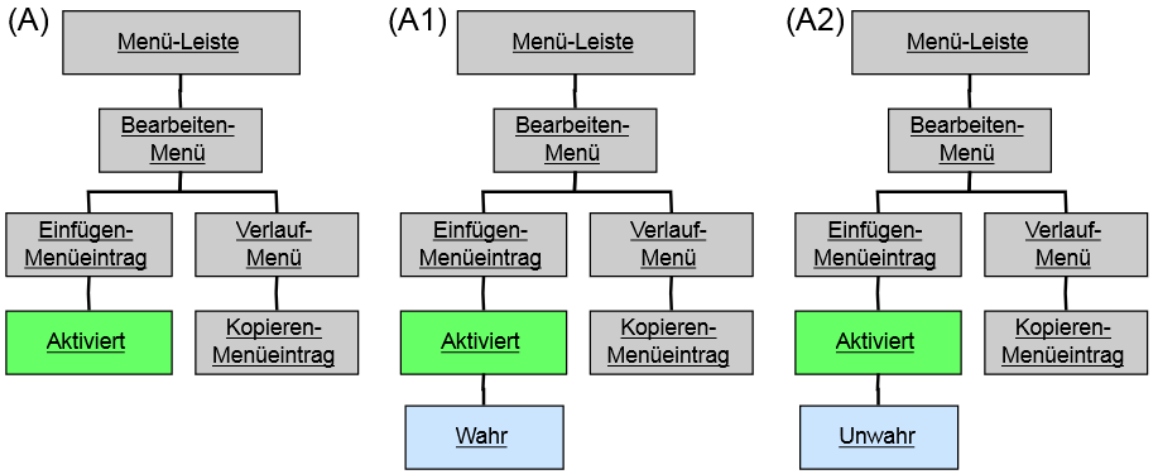


Abbildung 69: Beispiel: Ausschnitt des Struktur-Widget-Trees des zweistufigen Menüs vom Taschenrechner (A) und zwei zugehörige Werte-Widget-Trees (A1 und A2).

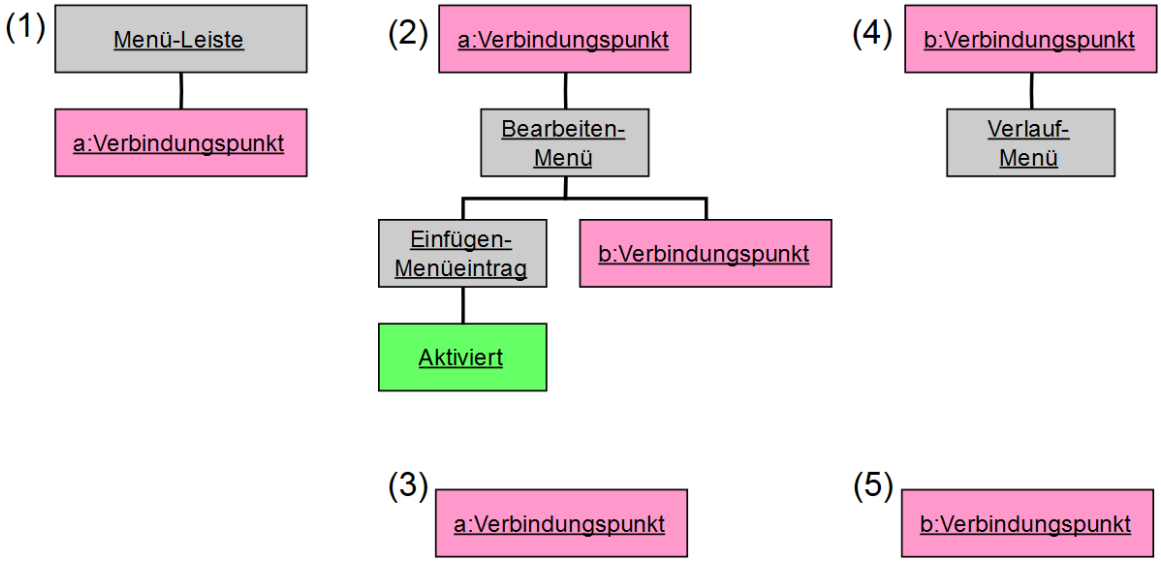


Abbildung 70: Beispiel: Ausschnitt des zerlegten und erweiterten Struktur-Widget-Trees des zweistufigen Taschenrechner-Menüs in fünf Widget-Tree-Faktoren.

Analog zum Struktur-Widget-Tree (A) werden auch die zwei gegebenen Werte-Widget-Trees (A1 und A2) in je fünf Faktoren zerlegt und erweitert. Dabei unterscheiden sie sich im Beispielausschnitt nur in einem der Faktoren, wie in Abbildung 71 dargestellt. Obwohl ursprünglich nur zwei Werte-Widget-Trees gegeben waren (A1 und A2), repräsentieren die zerlegten und erweiterten Widget-Trees nun fünf verschiedene Werte-Widget-Trees. Die Faktoren lassen sich zu folgenden Werte-Widget-Trees kombinieren:

- (1)-(2a)-(4): gegebener Widget-Tree A1, beide Menüs geöffnet, Einfügen aktiviert,
- (1)-(2b)-(4): gegebener Widget-Tree A2, beide Menüs geöffnet, Einfügen deaktiviert,

- (1)-(2a)-(5): nur das erste Menü ist geöffnet, Einfügen aktiviert,
- (1)-(2b)-(5): nur das erste Menü ist geöffnet, Einfügen deaktiviert,
- (1)-(3): kein Menü ist geöffnet.

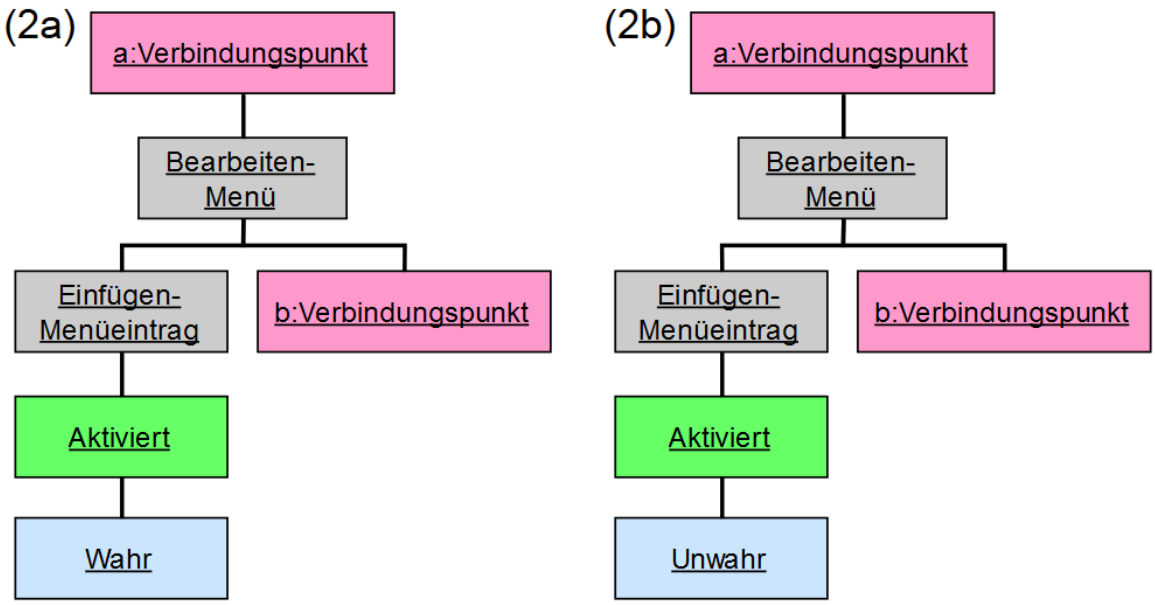


Abbildung 71: Die ursprünglichen Werte-Widget-Trees werden analog zu ihren Struktur-Widget-Trees zerlegt und erweitert.

4.4.3.3 Subaktivität „orthogonale Widget-Trees vereinigen“

In dieser Subaktivität werden die einzelnen orthogonalen Widget-Trees des vorherigen Teilprozesses zu neuen orthogonalen Widget-Trees vereinigt.

Bei diesem Schritt findet eine Verallgemeinerung statt, durch welche sich die Faktoren der neuen orthogonalen Widget-Trees vielfältiger kombinieren lassen. Somit repräsentieren die neuen orthogonalen Widget-Trees quantitativ mehr klassische Widget-Trees als die nichtvereinigten Widget-Trees.

Vorgehen

In den orthogonalen Struktur-Widget-Trees des vorherigen Teilprozesses werden ähnliche Struktur-Widget-Tree-Faktoren gesucht und vereinigt. Dazu wird wie folgt vorgegangen:

(1) Es werden alle Struktur-Widget-Tree-Faktoren eines Fenster-Zustandes miteinander verglichen, ohne dabei ihre Verbindungspunkte, welche Blattknoten sind, zu beachten.
(2) Es wird nach gleichen Struktur-Widget-Tree-Faktoren gesucht und diese als gleich markiert.
(3) Die Verbindungspunkte werden nun nicht mehr ignoriert, aber zuvor als gleich erkannte Faktoren bleiben als gleich markiert.
(4) Als gleich markierte Faktoren werden miteinander vereinigt (Vereinigung der Knoten- und Kanten-Mengen) und bilden dann einen vereinigten Faktor. Die so durch Vereinigung entstandenen Faktoren bilden die neuen orthogonalen Struktur-Widget-Trees.

Immer, wenn Struktur-Widget-Tree-Faktoren so vereinigt werden, dann werden die ihnen zugeordneten Werte-Widget-Tree-Faktoren gleichermaßen vereinigt. Dadurch ist jeder neue Werte-Widget-Tree-Faktor weiterhin einem Struktur-Widget-Tree-Faktor zugeordnet, welcher genau dem Werte-Widget-Tree-Faktor ohne die Werte-Knoten entspricht.

Beispiel

Ein Beispiel, wie zwei gegebene orthogonale Struktur-Widget-Trees vereinigt werden, ist in Abbildung 72 dargestellt. Die gegebenen Widget-Tree-Faktoren repräsentieren drei klassische Widget-Trees:

- Kontextmenü geöffnet,
- nichtmodaler Dialog geöffnet,
- nichts von beiden geöffnet.

Nach dem Vereinigen der Widget-Trees wurden die sechs Faktoren auf fünf Faktoren reduziert. Durch die Verallgemeinerung kann nun noch ein weiterer klassischer Widget-Tree repräsentiert werden:

- Kontextmenü und nichtmodaler Dialog geöffnet.

Die zugehörigen Werte-Widget-Trees werden analog vereinigt, ähnlich wie im vorherigen Beispiel (vgl. Abschnitt 4.4.3.2).

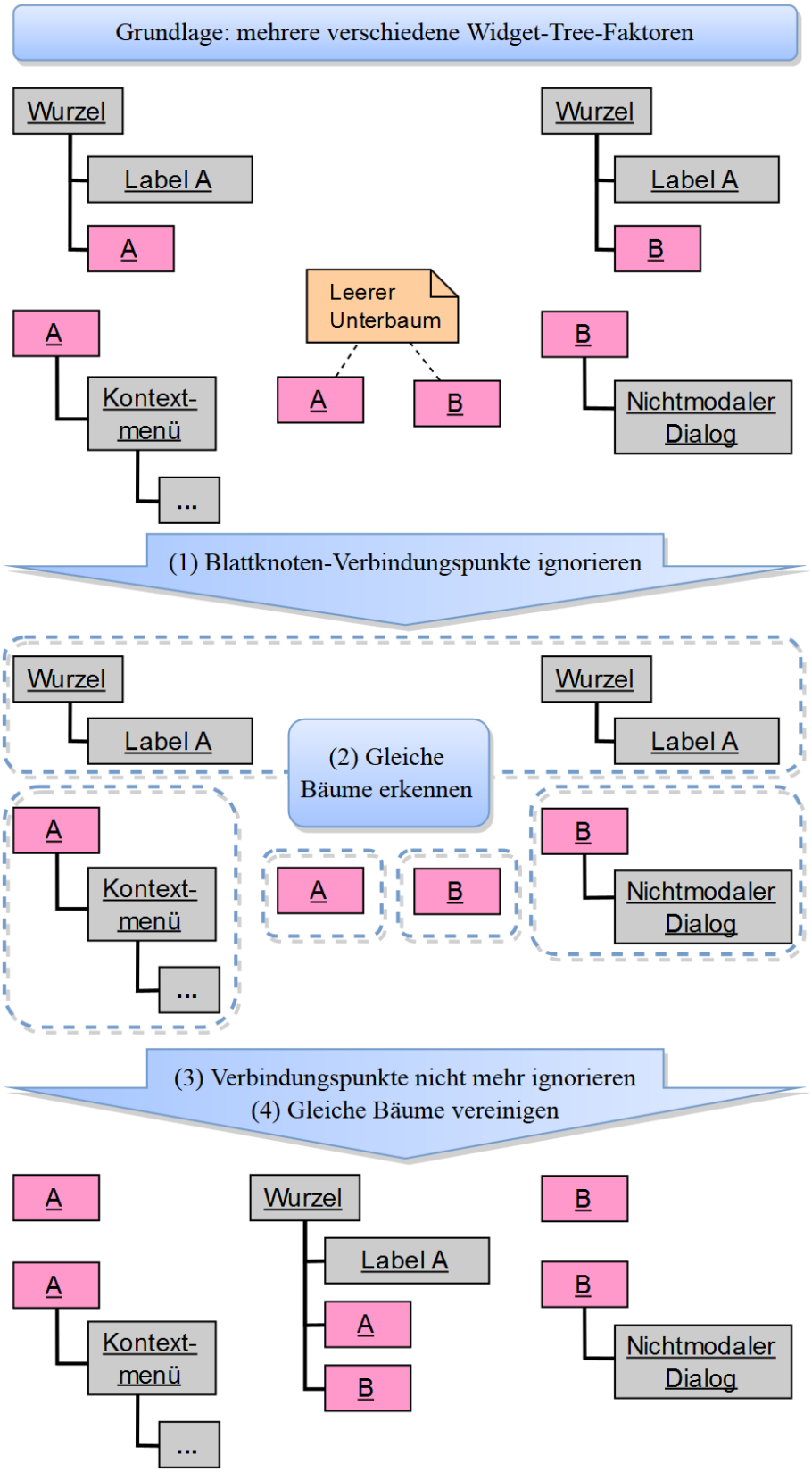


Abbildung 72: Beispiel: Subaktivität „orthogonale Widget-Trees zusammenführen“

4.4.3.4 Subaktivität „den Widget-Trees Zustände zuordnen“

In dieser Subaktivität werden allen zuvor erstellten orthogonalen Widget-Trees eines Fenster-Zustands neue Zustände zugeordnet. Dazu wird wie folgt vorgegangen:

(1) Im Fenster-Zustand wird ein orthogonaler Bereich B_{Main} angelegt.

(2) Für jeden orthogonalen Struktur-Widget-Tree werden die folgenden Schritte ausgeführt:

   a. Zu dem Struktur-Widget-Tree-Faktor F_{Wurzel} welcher den Wurzelknoten enthält, wird ein neuer Struktur-Zustand S_{Wurzel} im Bereich B_{Main} angelegt.
   b. Zu jedem Widget-Tree-Bereich B_{WT}, außer dem Bereich welcher F_{Wurzel} enthält, wird ein neuer orthogonaler Bereich B_{FSM} im Fenster-Zustand angelegt.
   c. Zu jedem Struktur-Widget-Tree-Faktor $F_{Faktor,i}$ welcher dem Bereich B_{WT} zugeordnet ist, wird ein neuer Struktur-Zustand $S_{Faktor,i}$ im Bereich B_{FSM} angelegt.
   d. Zu jedem Werte-Widget-Tree-Faktor $F_{i,j}$ welcher Struktur-Widget-Tree F_i zugeordnet ist, wird ein neuer Werte-Zustand $S_{i,j}$ im Struktur-Zustand S_i angelegt.

Beispiel

Ein Beispiel, wie aus orthogonalen Widget-Trees ein orthogonaler, hierarchischer Zustandsautomat entsteht, ist in Abbildung 73 und Abbildung 74 zu sehen.

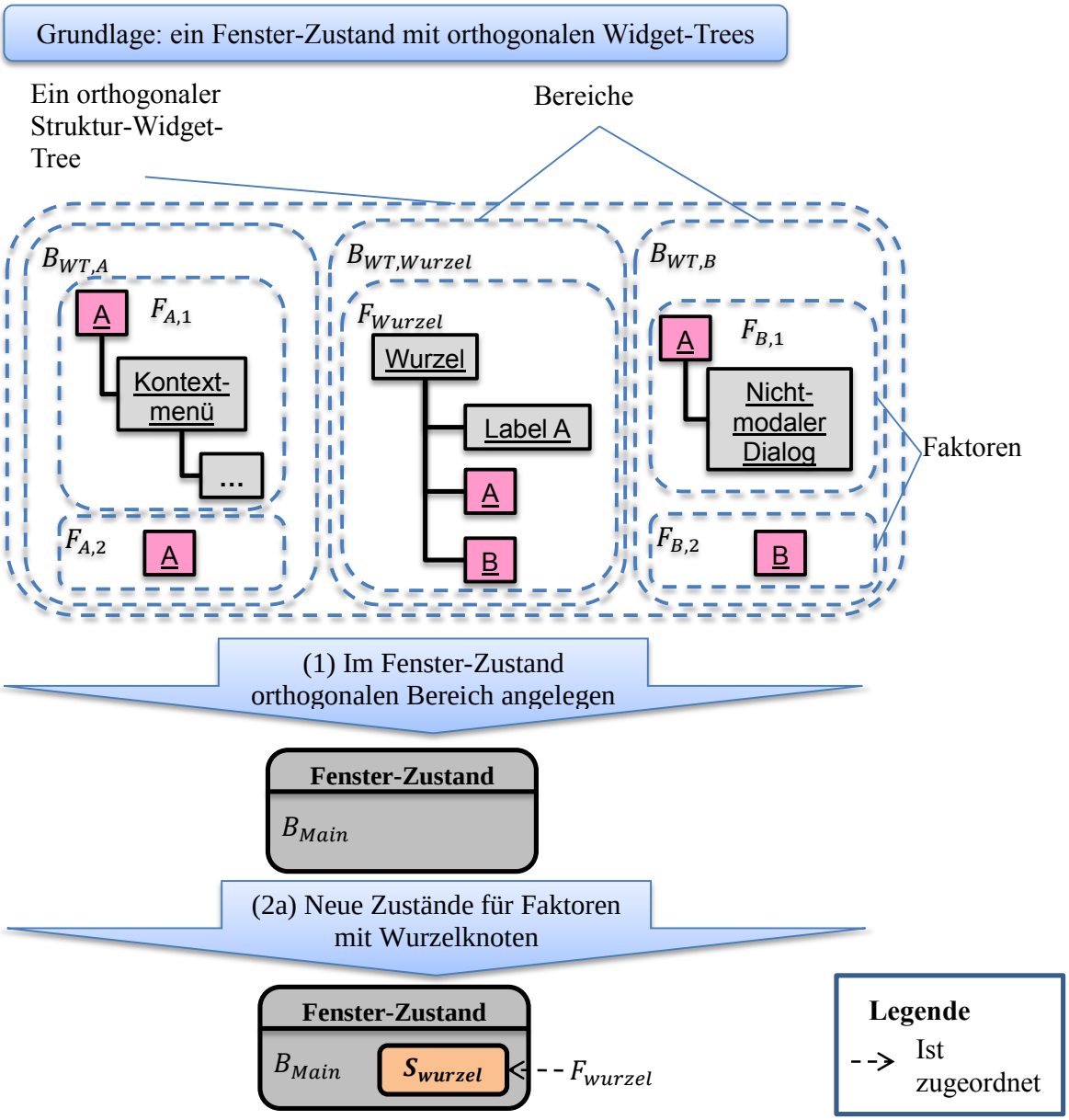


Abbildung 73: Subaktivität „den Widget-Trees Zustände zuordnen“ – Beispiel Teil 1: aus einem orthogonalen Widget-Tree wird ein orthogonaler, hierarchischer Zustandsautomat konstruiert.

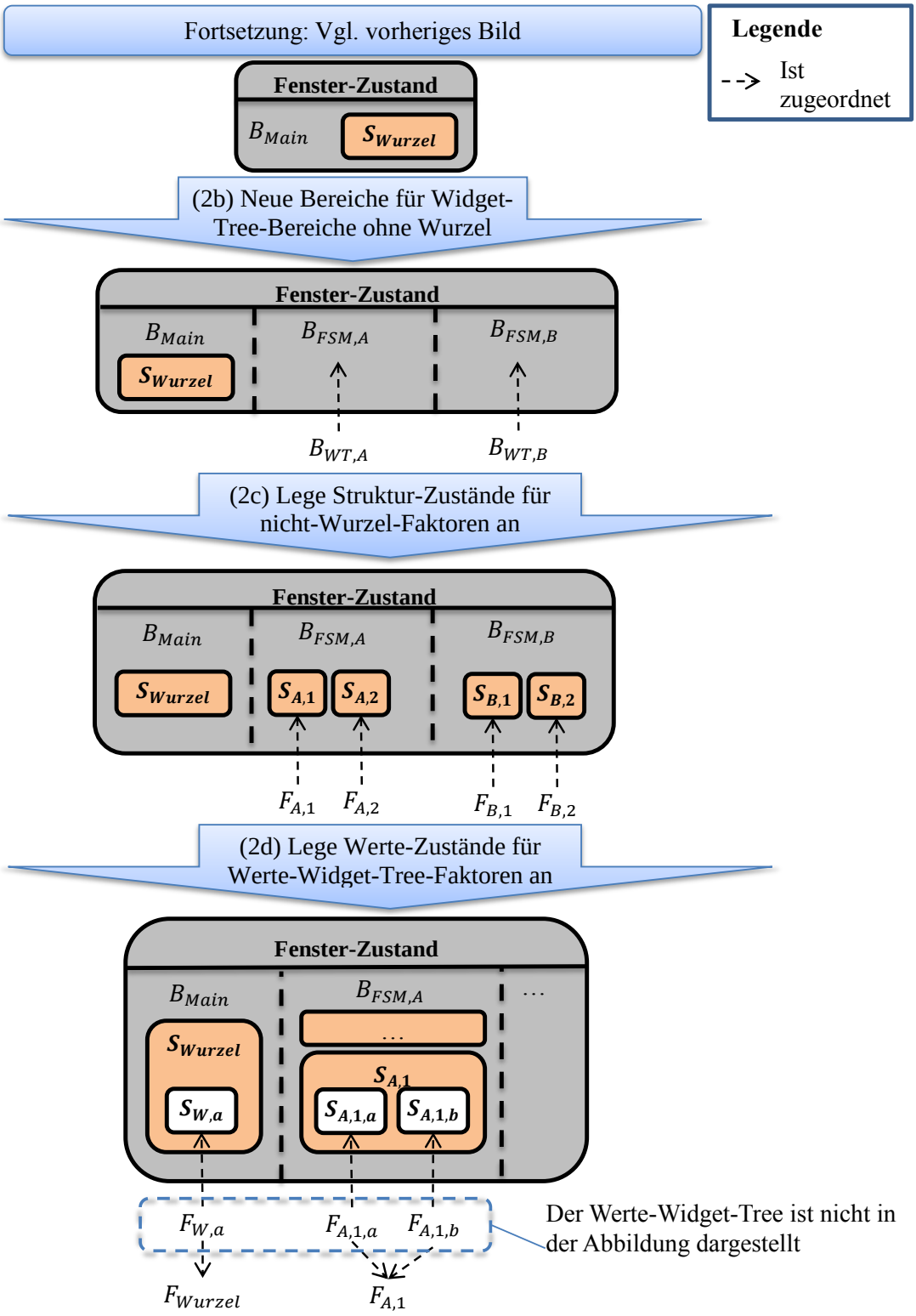


Abbildung 74: Subaktivität „den Widget-Trees Zustände zuordnen“ – Beispiel Teil 2: aus einem orthogonalen Widget-Tree wird ein hierarchischer orthogonaler Zustandsautomat konstruiert.

4.4.4 Subaktivität „Sequenzen einbeziehen“

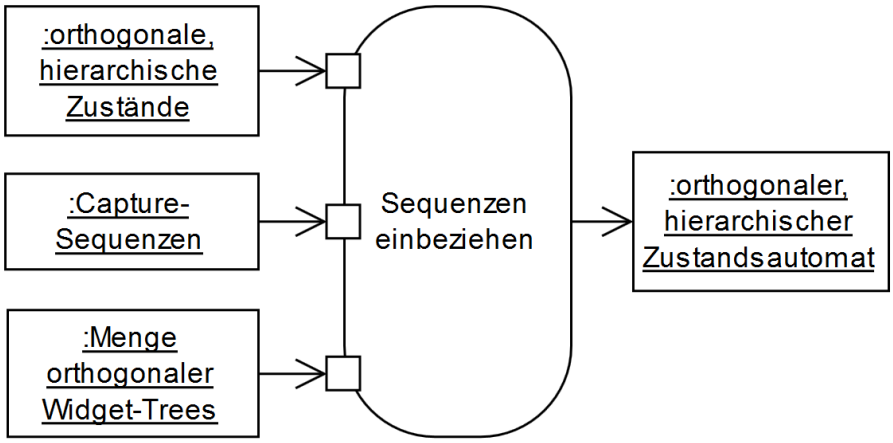


Abbildung 75: Phase 2 „Modelle abstrahieren“ – Subaktivität „Sequenzen einbeziehen“

In diesem Schritt werden die ausgeführten Aktionen aus den Capture-Sequenzen genutzt, um die Zustände des orthogonalen, hierarchischen Zustandsautomaten mit Transitionen zu verbinden (siehe Abbildung 75). Um den Zustandsautomaten zu vervollständigen, werden nacheinander alle Capture-Sequenzen genutzt. Für jede Capture-Sequenz werden die folgenden Schritte ausgeführt:

1) Für jeden durchlaufenen Zustand S_i innerhalb der Sequenz und seinen zugehörigen klassischen Werte-Widget-Tree W_i werden folgende Schritte ausgeführt (vgl. Abbildung 76):
    a. Es wird im Zustandsautomat der Fenster-Zustand $S_{Window,i}$ mit dem gleichen Fensternamen ausgewählt.
    b. Innerhalb des Fenster-Zustands $S_{Window,i}$ wird aus jedem Bereich genau ein Werte-Zustand $S_{Window,i,j}$ gewählt, so dass dessen zugehörige Werte-Widget-Tree-Faktoren $W_{Window,i,j}$ zusammen genau W_i repräsentieren. Die so gewählten Werte-Zustände $S_{Window,i,j}$ werden mit S_i assoziiert. Deren zugehörige Werte-Widget-Tree-Faktoren $W_{Window,i,j}$ werden entsprechend mit W_i assoziiert.
2) Für jede Aktion A_k mit dem Ausgangszustand S_k und Folgezustand S_{k+1} innerhalb der Sequenz werden folgende Schritte ausgeführt:
    a. Es wird ermittelt, auf welchem Widget die Aktion A_k ausgeführt wurde.

b. Aus den mit dem Werte-Widget-Tree W_k assoziierten Werte-Widget-Tree-Faktoren wird der Faktor $W_{k,Widget}$ gewählt, welcher den Knoten enthält, der das Widget repräsentiert. Repräsentiert keiner der Faktoren das Widget, dann wird der Faktor aus dem Bereich B_{Main} gewählt (vgl. Abbildung 77).

c. Im Zustandsautomaten wird eine Transition t_k hinzugefügt, welche $S_{k,Widget}$ als Ausgangszustand hat. Die Transition wird ausgelöst durch das Ereignis A_k. Der Folgezustand und die ausgelösten Aktionen der neuen Transition hängen davon ab, ob sich die mit Ausgangszustand S_k und Folgezustand S_{k+1} der Sequenz assoziierten Zustände im gleichen Fenster-Zustand befinden.

    i. Ist das der Fall, so ist der Folgezustand von t_k der Werte-Zustand, welcher sich im gleichen Bereich wie $S_{k,Widget}$ befindet und mit S_{k+1} assoziiert ist. Die Transition t_k löst selbst die Aktion A_{k*} aus. Die neue Transition wird also mit „A_k/A_{k*}“ beschriftet (vgl. Abbildung 78).

    ii. Andernfalls ist der Folgezustand von t_k ein neuer Gabelungs-Knoten. In diesem Gabelungs-Knoten starten weitere neue Transitionen, welche in den mit S_{k+1} assoziierten Werte-Zuständen enden. Die Transition t_k löst selbst keine Aktion aus, wird also mit „A_k“ beschriftet (vgl. Abbildung 79).

d. Dieser Schritt wird nur ausgeführt, wenn sich die mit S_k und S_{k+1} assoziierten Zustände im gleichen Fenster-Zustand befinden: Für alle mit dem Werte-Widget-Tree W_k assoziierten Werte-Widget-Tree-Faktoren $W_{k,i}$, außer dem Faktor $W_{k,Widget}$, werden folgende Schritte ausgeführt (vgl. Abbildung 78):

    i. Der Werte-Zustand $S_{k+1,i}$ welcher sich im gleichen Bereich wie $S_{k,i}$ befindet und mit W_{k+1} assoziiert ist wird ermittelt.

    ii. Sind $S_{k,i}$ und $S_{k+1,i}$ verschieden, dann wird eine Transition von $S_{k,i}$ nach $S_{k+1,i}$ erzeugt, welche durch das Ereignis A_{k*} ausgelöst wird.

4.4.4.1 Beispiel

Ein Beispiel, wie die Sequenzen einbezogen werden, um den Zustandsautomaten zu vervollständigen, ist in Abbildung 76 bis Abbildung 79 zu sehen.

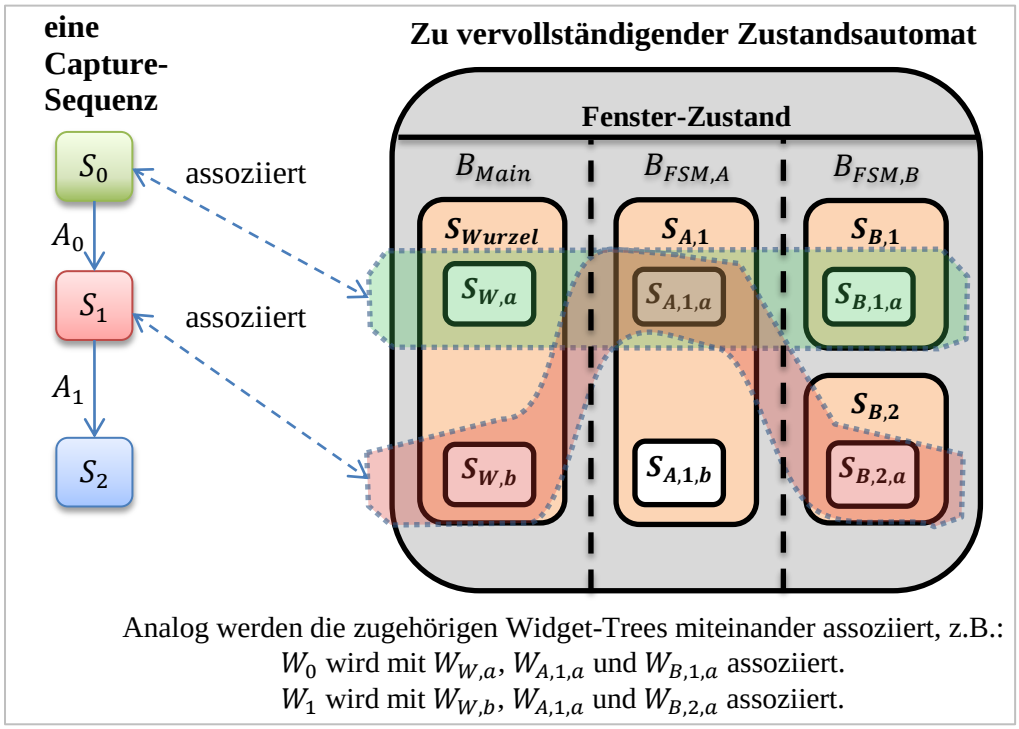


Abbildung 76: Jeder durchlaufene Zustand der Capture-Sequenzen wird mit den ihn repräsentierenden Zuständen im orthogonalen, hierarchischen Zustandsautomaten assoziiert.

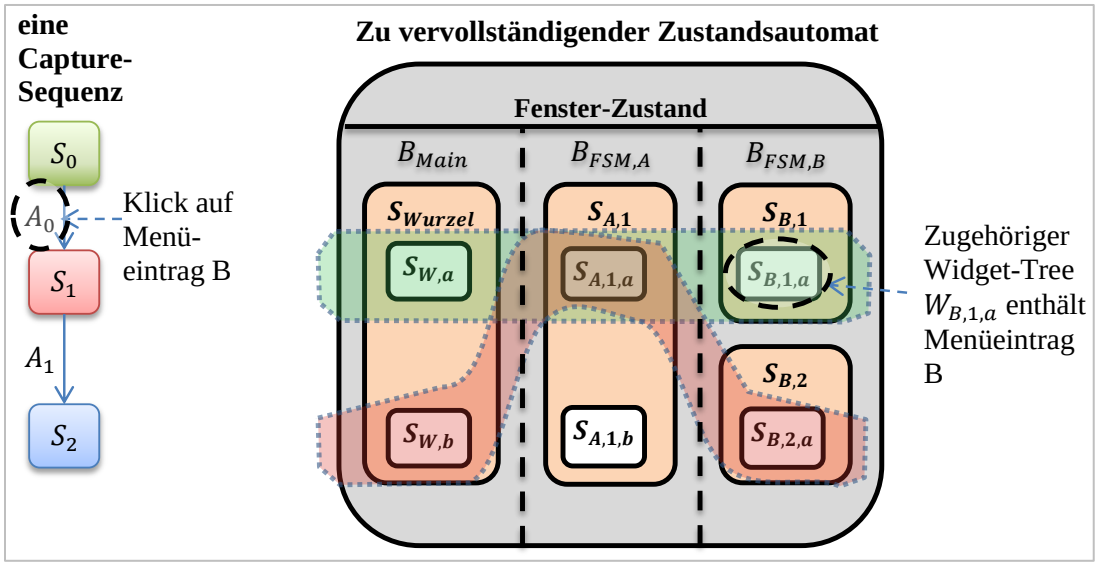


Abbildung 77: Für jede Aktion wird der Werte-Zustand gesucht, der das Widget repräsentiert, auf dem die Aktion durchgeführt wurde.

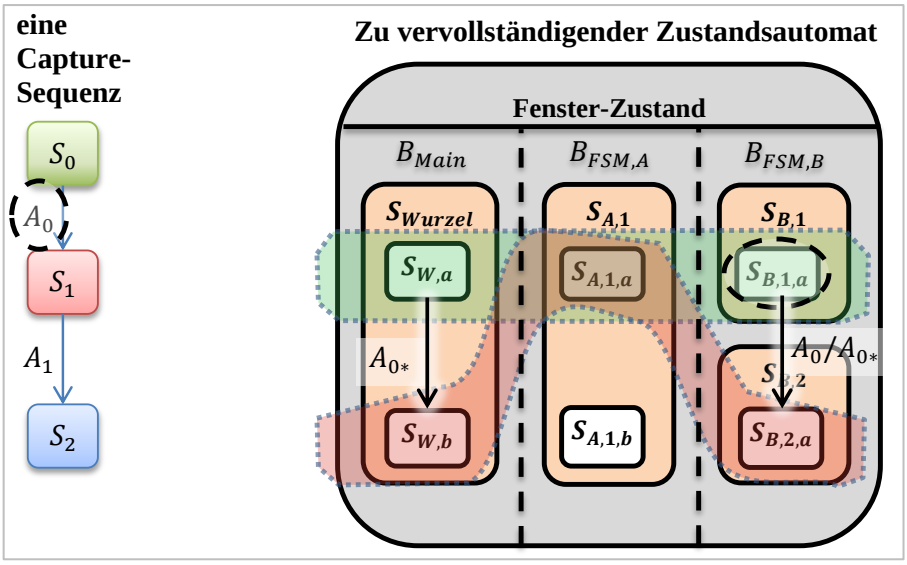


Abbildung 78: Die Transitionen werden entsprechend der Aktion zum Zustandsautomaten hinzugefügt. Dabei wird der Zustand, welcher das benutzte Widget repräsentiert, zuerst verlassen, und löst dabei wiederrum die restlichen Transitionen aus.

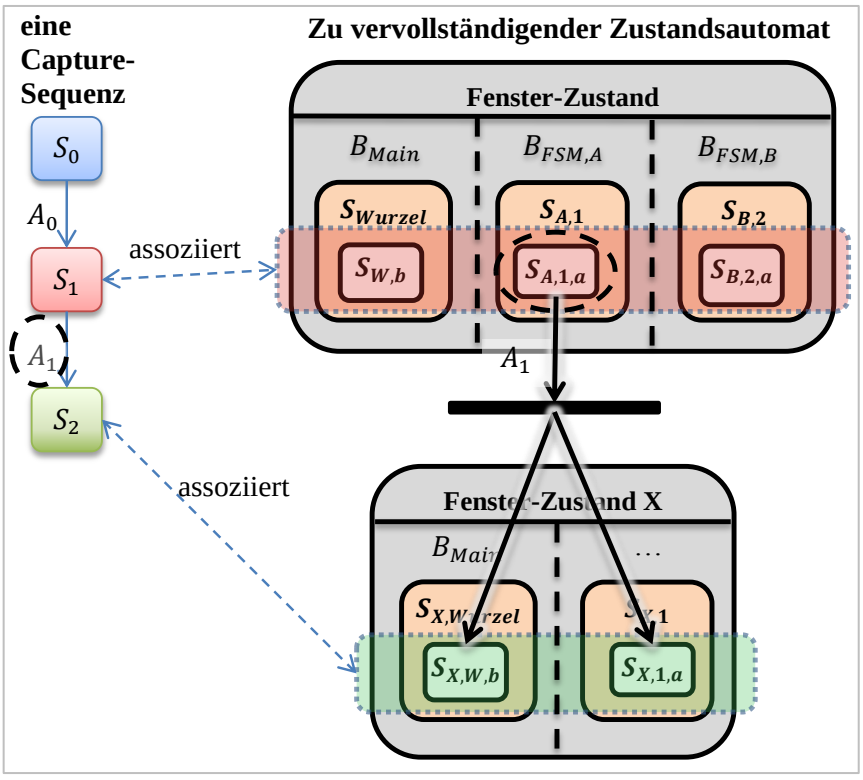


Abbildung 79: Sorgt eine Aktion dafür, dass der Fenster-Zustand verlassen wird, dann muss ein Zustand aus jedem Bereich des neuen Fenster-Zustands betreten werden. Dazu wird ein Gabelungs-Knoten eingeführt.

4.4.5 Subaktivität „Klassifikationsbaum erzeugen“

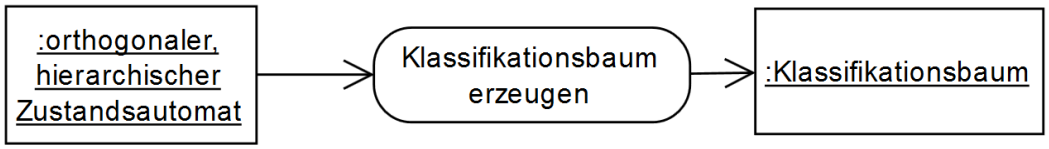


Abbildung 80: Phase 2 „Modelle abstrahieren“ – Subaktivität „Klassifikationsbaum erzeugen“

In der Subaktivität „Klassifikationsbaum erzeugen“ wird aus dem hierarchischen Zustandsautomat ein Klassifikationsbaum erstellt (siehe Abbildung 80). Die Abbildungsvorschrift, um aus einem Zustandsautomaten einen Klassifikationsbaum herzuleiten wurde in Abschnitt 4.2.3.5 beschrieben. Diese Subaktivität erstellt den Klassifikationsbaum und assoziiert seine Elemente mit denen des gegebenen Zustandsautomaten.

4.4.5.1 Beispiel

Ein Beispiel zur Herleitung eines Klassifikationsbaumes aus einem Zustandsautomaten wurde in Abschnitt 4.2.3.5 beschrieben.

4.4.6 Subaktivität „Modelle aktualisieren“

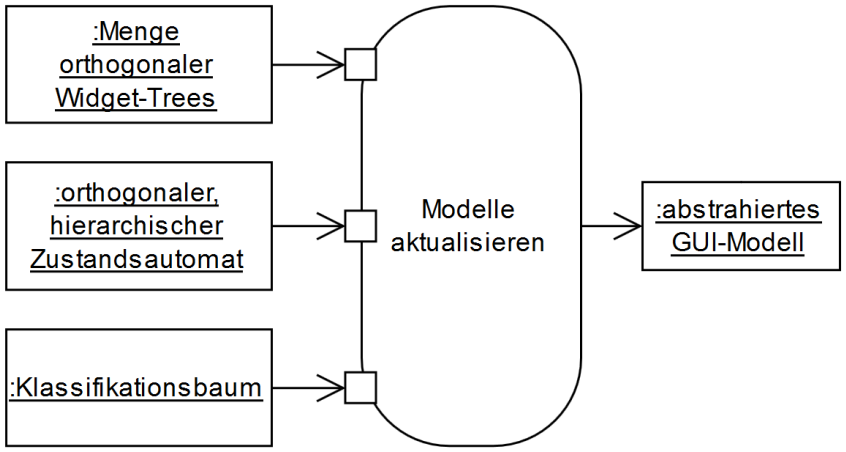


Abbildung 81: Phase 2 „Modelle abstrahieren“ - Subaktivität „Modelle aktualisieren“

Phase 2 endet mit der Subaktivität „Modelle aktualisieren“, welche das abstrahierte GUI-Modell aktualisiert. Diese Subaktivität stellt die in Phase 2 ermittelten Teilergebnisse (siehe Abbildung 81) unverändert im abstrahierten GUI-Modell bereit. Dadurch sind sie für den Tester sichtbar.

4.4.6.1 Beispiel

Die Beispiele des abstrahierten GUI-Modells aus Abschnitt 4.2.3 zeigen, wie Ergebnisse dieser Subaktivität aussehen könnten.

4.4.7 Zusammenfassung von Phase 2

Phase 2 „Modelle abstrahieren“ nutzt die systemnahen Modelle der ersten Phase, um daraus abstraktere, benutzerfreundlichere Modelle zu erzeugen. Dazu erweitert es die Modelle um Orthogonalität und Hierarchie. Im Gegensatz zum systemnahen Modell repräsentiert das abstrahierte GUI-Modell auch Systemverhalten, welches nicht mittels Capture aufgenommen wurde, indem einige allgemeine Annahmen über GUIs getroffen werden. Das abstrahierte Modell besteht aus einem Zustandsautomaten, einem Klassifikationsbaum und einer Menge von Widget-Trees. Die detaillierte Ermittlung der Modelle wurde in diesem Abschnitt beschrieben.

Nachdem das SUT nun modelliert ist, soll es getestet werden. Dazu wird das Modell genutzt, um auszuführende Testsequenzen auf diesem Modell zu ermitteln. Der Prozess zum Ermitteln dieser Sequenzen wird im nächsten Abschnitt detailliert beschrieben.

4.5 Phase 3 „Testsequenzen generieren“

In der dritten Phase werden Testsequenzen spezifiziert.

4.5.1 Überblick

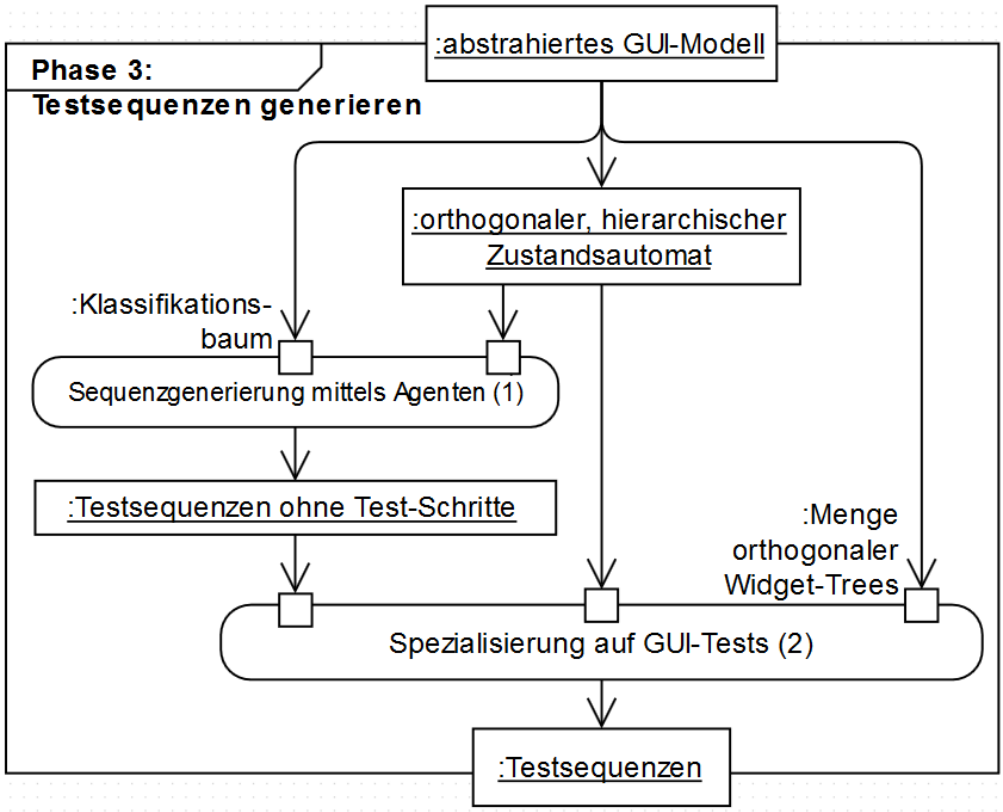


Abbildung 82: Überblick der Subaktivitäten und Zwischenprodukte von Phase 3 „Testsequenzen generieren und deren Zusammenhänge

Abbildung 82 gibt einen Überblick über die Subaktivitäten und Zwischenprodukte von Phase 3 „Testsequenzen generieren“ und stellt die Zusammenhänge zwischen ihnen dar. Zuerst werden die einzelnen Teilmodelle des abstrahierten GUI-Modells genutzt, um Testsequenzen zu ermitteln und – wie in der Klassifikationsbaum-Methode – in einer Testmatrix zu beschreiben (1).[36] Jede Sequenz repräsentiert auch einen Pfad durch den Zustandsautomaten. Die ermittelten Sequenzen genügen dabei Abdeckungskriterien wie Zustandsüberdeckung oder Pfadüberdeckung. In einer weiteren Subaktivität (2) werden die Testsequenzen „ausführbar“ gemacht. Das Ergebnis sind ausführbare Testsequenzen mit Folgen von Widgets und den darauf auszuführenden Aktionen (vgl. Abschnitt 4.2.4).

In den folgenden Unterabschnitten werden die einzelnen Subaktivitäten und die erzeugten Zwischenprodukte genauer erläutert.

[36] Die Nummern in diesem Absatz beziehen sich auf Abbildung 82.

4.5.2 Subaktivität „Sequenzgenerierung mittels Agenten“

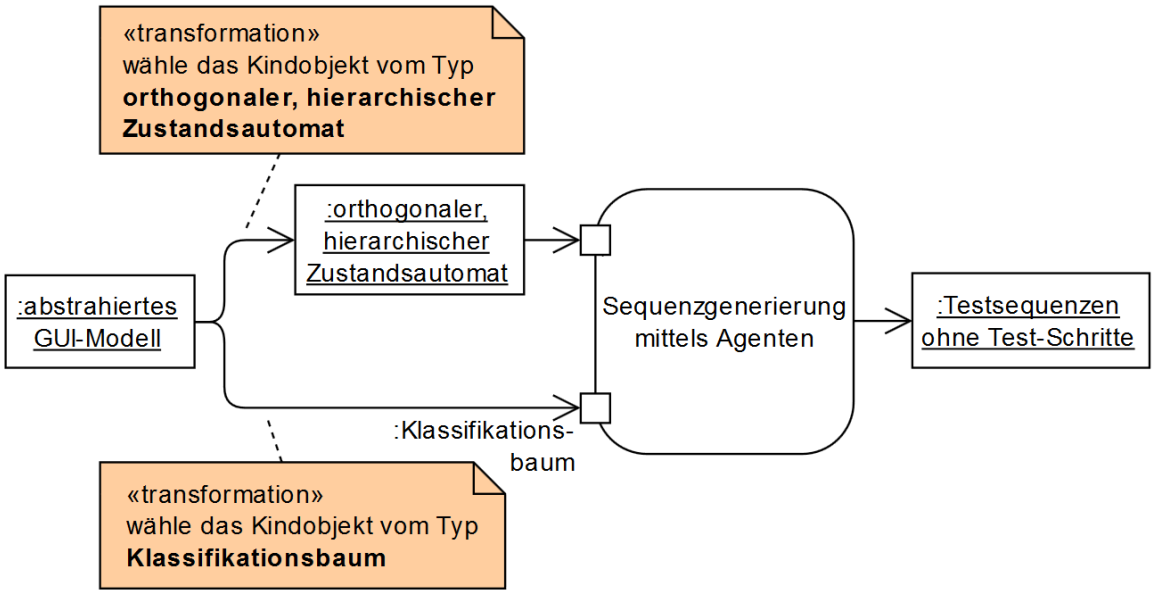


Abbildung 83: Phase 3 „Testsequenzen generieren“ – Subaktivität „Sequenzgenerierung mittels Agenten“

In dieser Subaktivität (vgl. Abbildung 83) werden der Klassifikationsbaum und der Zustandsautomat aus dem abstrahierten GUI-Modell genutzt, um Testsequenzen zu spezifizieren.

Zur Erzeugung der Testsequenzen wird ein von Wegener und Kruse ([Krus11, KrWe12]) vorgestelltes Verfahren genutzt. Dieses Verfahren ist nicht im Rahmen dieser Arbeit entstanden und wird daher nur kurz erläutert. Für Details sei auf die oben genannten Quellen verwiesen.

Das Verfahren arbeitet agentenbasiert. Es benutzt zwei Typen von Agenten: *Walker Agents* und *Coverage Agents*.

Die Walker Agents sind dafür verantwortlich, durch den orthogonalen, hierarchischen Zustandsautomaten zu „laufen“. Die durchlaufenen Zustände werden dabei als Testschritte aufgezeichnet und führen so zur Erzeugung von Testsequenzen. Die Agenten sorgen anhand der Transitionen dafür, dass nur legale Zustandsänderungen stattfinden. Sie laufen im Startzustand los. *Walker Agents* können Kind-Agenten haben, wodurch hierarchische Zustände auf allen Hierarchieebenen gleichzeitig durchlaufen werden können.

Die Agenten können erzeugt und vernichtet werden, wodurch das entkoppelte Durchlaufen orthogonaler Bereiche möglich wird.

Der zweite Typ – die *Coverage Agents* – ist dafür zuständig, die *Walker Agents* intelligent durch den Zustandsautomaten zu leiten, um möglichst effektiv die gewünschten Überdeckungen zu erzielen. Dabei gibt es einen *Coverage Agent* je Überdeckungskriterium. Ihr Lebenszyklus beginnt mit der Generierung der Testsequenz und endet mit Erreichen des jeweiligen Überdeckungskriteriums. Sie haben eine feste Reihenfolge, um Konflikte zwischen den *Coverage Agents* zu behandeln.

Die resultierenden Testsequenzen enthalten noch keine Test-Schritte – wie in 4.2.4 spezifiziert – sondern bisher nur die Test-Zustände.[37] Daher können die Testsequenzen noch nicht automatisch auf der GUI ausgeführt werden.

4.5.2.1 Beispiel

Ein Beispiel für eine resultierende *Testsequenz ohne Test-Schritte* ist in Abbildung 42 dargestellt. Derartige Testsequenzen kann es am Ende dieser Subaktivität auch mehrere geben.

[37] Hinweis: *Test-Schritte* in dieser Arbeit sind nicht das Gleiche wie *Test-Schritte* in den Arbeiten von Kruse und Wegener ([Krus11, KrWe12]). Am Ende dieser Subaktivität liegen *Test-Zustände* vor, welche in den genannten Arbeiten *Test-Schritte* genannt wurden.

4.5.3 Subaktivität: „Spezialisierung auf GUI-Tests“

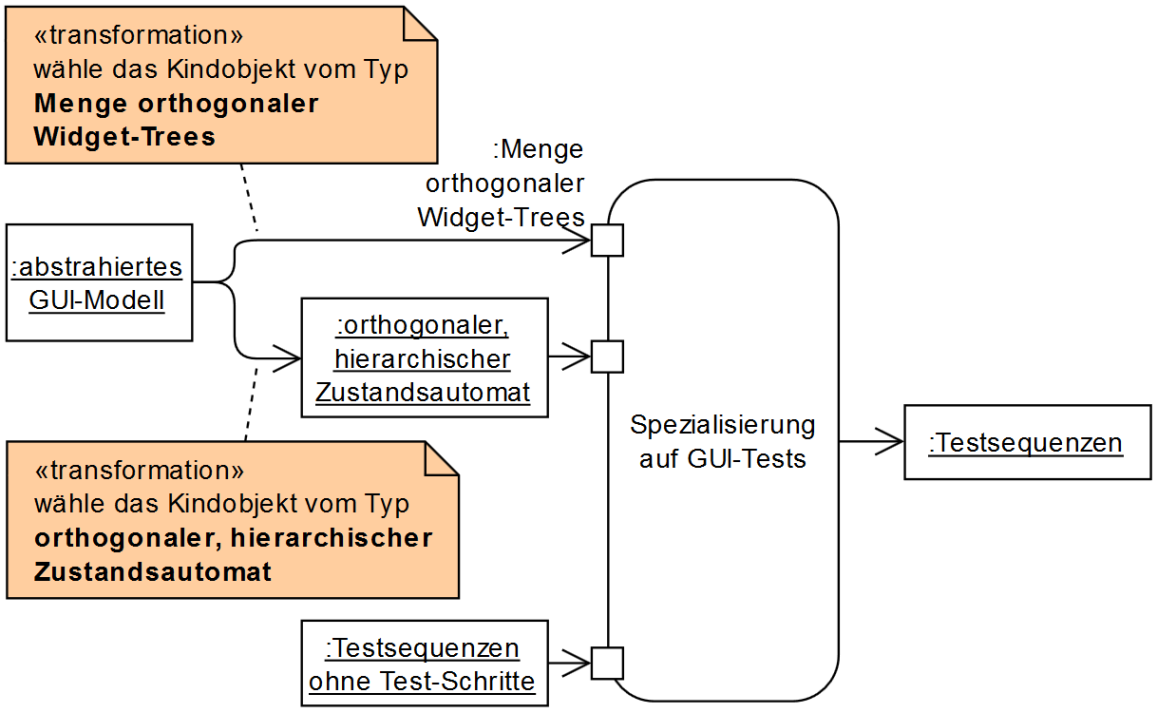


Abbildung 84: Phase 3 „Testsequenzen generieren“ – Subaktivität: „Spezialisierung auf GUI-Tests“

Die Testsequenzen von Wegener und Kruse [Krus11, KrWe12] werden allgemein für jede Art von Klassifikationsbaum und Zustandsautomat generiert. Diese Diplomarbeit spezialisiert sich wiederum auf GUI-Tests. Die Testsequenzen sollen in der nächsten Phase automatisch aus einer GUI ausgeführt werden. Dazu reicht es nicht, eine Sequenz von Test-Zuständen zu haben, vielmehr wird eine Sequenz von Widgets und Aktionen benötigt.

In dieser Subaktivität werden die Testsequenzen zusätzlich mit Test-Schritten versehen, welche wiederum das zu bedienende Widget und die darauf auszuführende Aktion enthalten (vgl. Abbildung 84).

Um zu ermitteln, welche Widgets und Aktionen für einen Test-Schritt benötigt werden, müssen diese Objekte aus den gegebenen Teilen des abstrahierten GUI-Modells herausgesucht werden. Abbildung 85 zeigt, wie das abstrahierte GUI-Modell einen Zusammenhang zwischen einem Test-Schritt und dem zugehörigen Widget und der darauf auszuführender Aktion herstellt. Dieser Zusammenhang kann genutzt werden, um den Test-Schritt mit Aktion und Widget auszustatten, wie in Abschnitt 4.2.4 beschrieben.

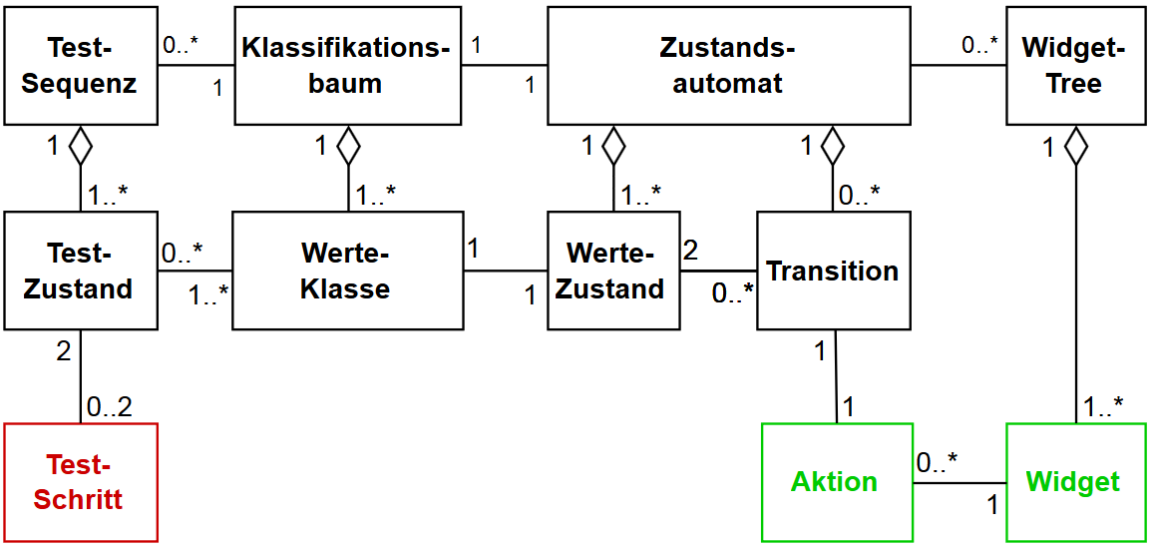


Abbildung 85: Im Modell der Testsequenzen sind die auszuführenden Aktionen und Widgets zunächst nicht direkt spezifiziert. Stattdessen besteht eine indirekte Assoziation zwischen Test-Schritten (rot) und den auszuführenden Aktionen und zu bedienenden Widgets (grün).

Bei Test-Schritten mit mehrdeutigen Zustandsübergängen (vgl. Abbildung 40) definiert jeweils die Transitions-Klasse unter dem Ausgangszustand über welche Transition man zum Folgezustand gelangt. Abbildung 86 zeigt, wie in diesem Fall Aktion und Widget für einen Test-Schritt ermittelt werden können.

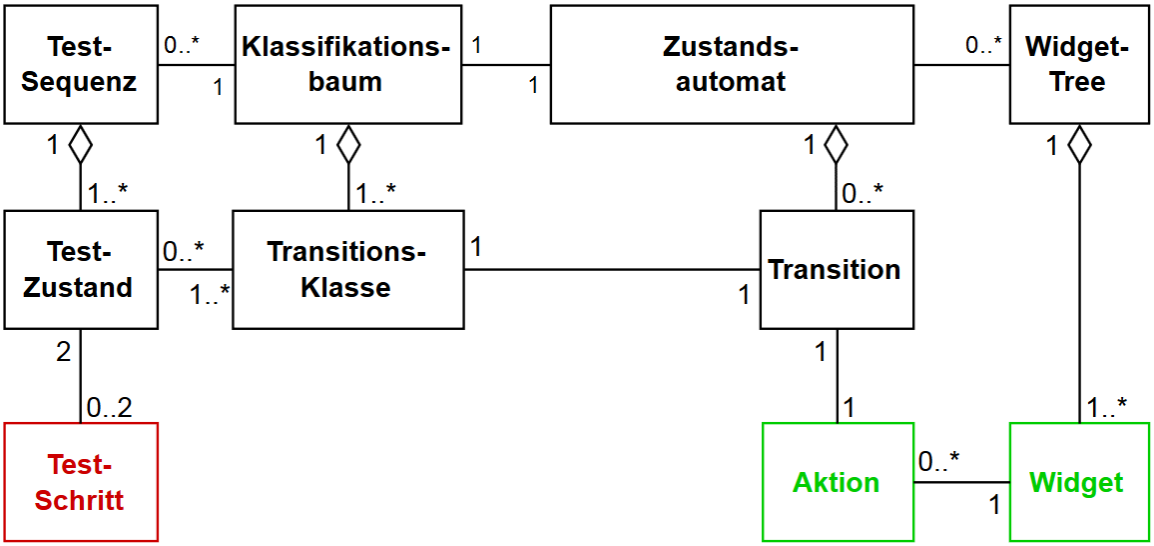


Abbildung 86: Ist nicht eindeutig, mittels welcher Transition man von einem Zustand in den nächsten gelangt, so wird der Zusammenhang zwischen Test-Schritt (rot) und den auszuführenden Aktionen und zu bedienenden Widgets (grün) mittels der Transitions-Klasse hergestellt.

4.5.3.1 Beispiel

Ein Beispiel für eine resultierende *Testsequenz* ist in 4.2.4.1 beschrieben. Derartige Testsequenzen kann es am Ende dieser Subaktivität auch mehrere geben.

4.5.4 Zusammenfassung von Phase 3

Phase 3 „Testsequenzen generieren“ nutzt das abstrahierte GUI-Modell der zweiten Phase, um daraus ausführbare Testsequenzen zu erzeugen. Dazu nutzt sie einen etablierten Algorithmus zur Generierung generischer Testsequenzen und spezialisiert diese auf die speziellen Anforderungen von GUI-Tests. Die entstehenden Testsequenzen werden durch eine Testmatrix und Test-Schritte mit Paaren von Aktion und Widget beschrieben. Wie diese Sequenzen im Detail ermittelt werden, wurde in diesem Abschnitt beschrieben.

Nachdem die Testsequenzen spezifiziert und ausführbar gemacht wurden, sollen sie in der nächsten Phase automatisch ausgeführt werden. Wie die automatische Ausführung funktioniert, wird im nächsten Abschnitt detailliert beschrieben.

4.6 Phase 4 „Tests ausführen“

In der vierten Phase werden die zuvor spezifizierten Testsequenzen automatisch ausgeführt.

4.6.1 Überblick

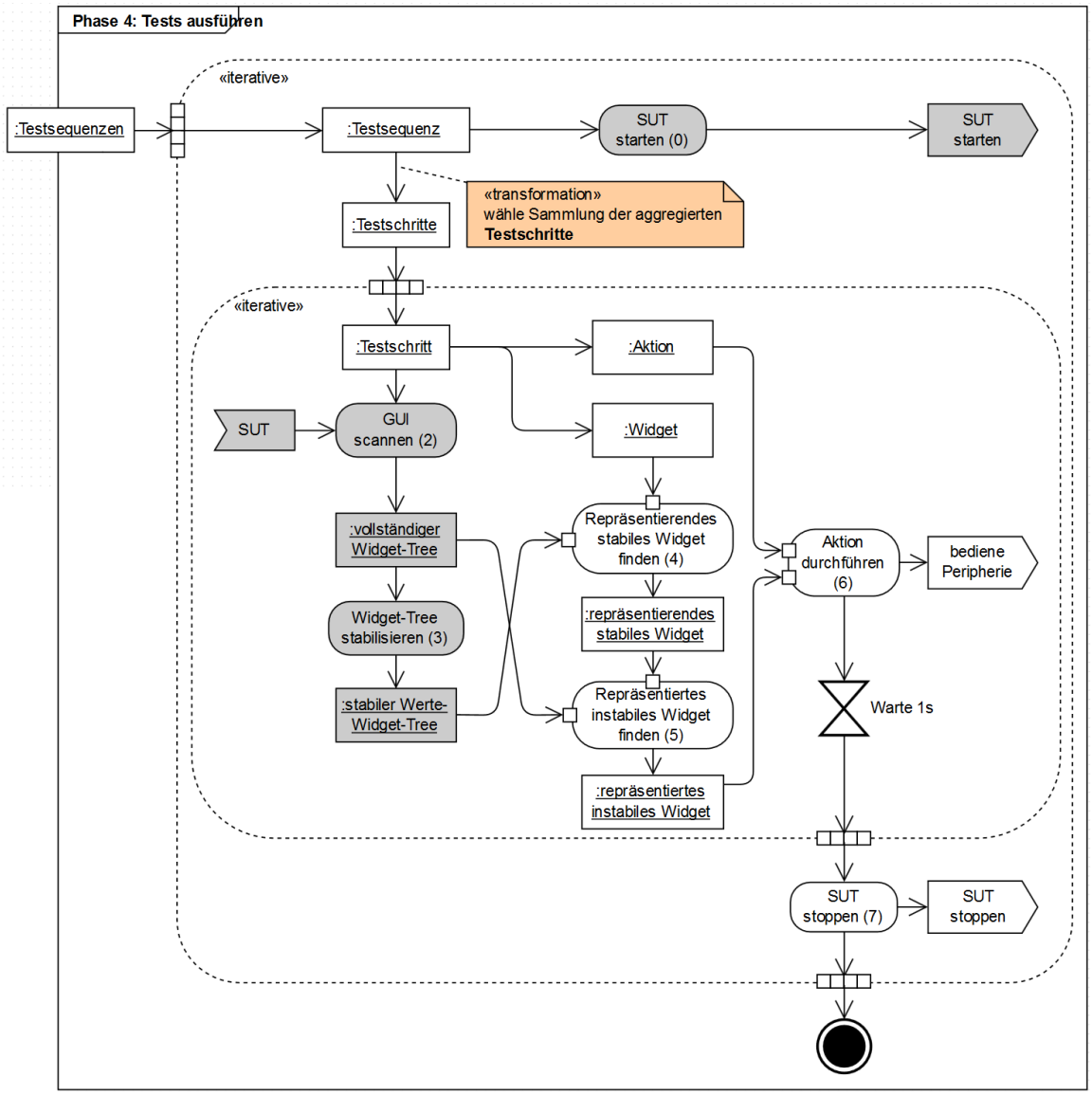


Abbildung 87: Überblick der Subaktivitäten und Zwischenprodukte von Phase 4 „Tests ausführen“ und deren Zusammenhänge. Grau hinterlegt sind die Bestandteile, die bereits aus Phase 1 „Capture“ bekannt sind und hier nur wiederverwendet werden.

Abbildung 87 gibt einen Überblick über die Subaktivitäten und Zwischenprodukte von Phase 4 „Tests ausführen“ und stellt die Zusammenhänge zwischen ihnen dar.

In dieser Phase werden alle gegebenen Testsequenzen nacheinander behandelt (0).[38] Zu Beginn jeder Testsequenz wird die GUI automatisch gestartet (vgl. Abschnitt 4.3.2).

Aus jeder Testsequenz wird dann wiederum jeder einzelne Test-Schritt behandelt. Zuerst wird die GUI gescannt, um den sowohl vollständigen (2) als auch stabilen (3) Widget-Tree der GUI zu ermitteln (vgl. Abschnitt 4.3.4 und 4.3.5). Das zu bedienende Widget wird im stabilen Widget-Tree gesucht (4) und darüber das zu bedienende Widget des laufenden SUTs ermittelt (5). Wurde so das zu bedienende Widget im Widget-Tree der laufenden GUI ermittelt, wird die auszuführende Aktion darauf ausgeführt (6), indem die Bedienung der Peripherie simuliert wird.

Nachdem alle Schritte einer Testsequenz durchgeführt wurden, wird das SUT beendet (7). Nach dem Ausführen aller Testsequenzen endet diese Phase.

In den folgenden Unterabschnitten werden die neuen Subaktivitäten etwas genauer erläutert.

4.6.2 Subaktivität „Repräsentierendes stabiles Widget finden“

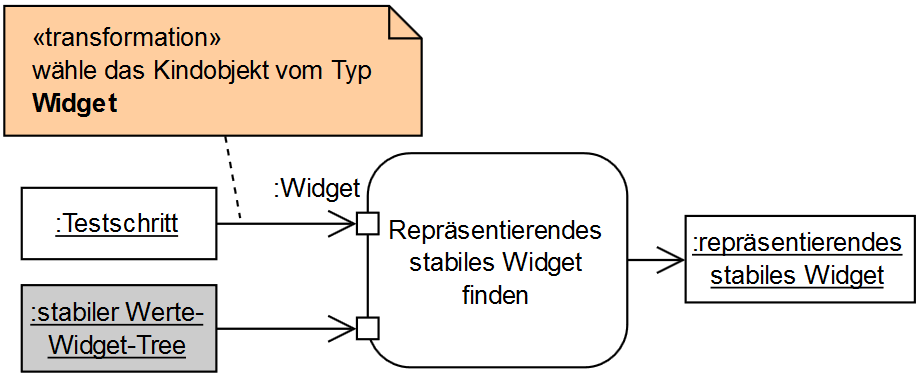


Abbildung 88: Phase 4 „Tests ausführen“ – Subaktivität „Repräsentierendes stabiles Widget finden“

Die Subaktivität „Repräsentierendes stabiles Widget finden“ (vgl. Abbildung 88) nimmt das Widget aus dem auszuführenden Testschritt und sucht ein äquivalentes Widget im stabilen Werte-Widget-Tree des laufenden SUT. Das gefundene Widget ist der stabile

[38] Die Nummern in diesem Absatz beziehen sich auf Abbildung 87. Nummer (1) fehlt dabei, weil die Teile der Nummerierung aus einem früheren Abschnitt wiederverwendet wurden, um konsistent zu bleiben.

Repräsentant des letztlich zu bedienenden Widget. Es wird für folgende Subaktivitäten bereitgestellt.

Wann zwei Widgets als äquivalent gelten, hängt von der verwendeten Strategie ab. Hier gibt es vier wesentliche Strategien, welche alle eigene Vor- und Nachteile haben:

1) Vergleiche alle Eigenschaften, aber ignoriere die Kind-Widgets.
2) Vergleiche alle Eigenschaften und ignoriere die Kind-Widgets und vergleiche außerdem die assoziierten stabilen IDs.[39]
3) Vergleiche alle Eigenschaften und rekursiv alle Kind-Widgets.
4) Vergleiche alle Eigenschaften und rekursiv alle Kind-Widgets und vergleiche außerdem die assoziierten stabilen IDs.

In dieser Arbeit wird ausschließlich Strategie 4 verwendet, welche am strengsten, aber auch am fehleranfälligsten ist.

4.6.2.1 Beispiel

Abbildung 54 (Seite 84) zeigt ein Beispiel für einen Widget-Tree, in welchem nach einem Widget gesucht werden soll. Ein Beispiel für ein zu bedienendes Widget eines gegebenen Testschrittes ist in Abbildung 89 dargestellt. Haben die beiden Ok-Button-Widgets außerdem die gleiche stabile ID, dann wird das Ok-Button-Widget des Widget-Trees an die folgenden Subaktivitäten weitergegeben.

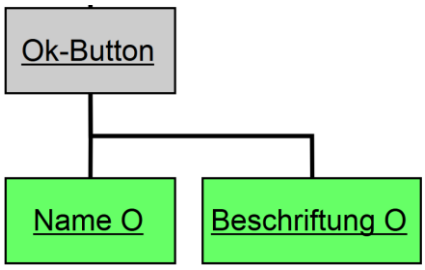


Abbildung 89: Beispiel eines zu bedienenden Widgets, wie es in einem Testschritt spezifiziert ist

[39] Die stabile ID eines Widgets repräsentiert den Pfad zum Wurzelknoten des Widget-Trees, das heißt rekursiv alle Eltern-Widgets. Ermittelt wurde die stabile ID in Phase 1 „Capture“ – Subaktivität „Widget-Tree stabilisieren“ (vgl. Abschnitt 4.3.5).

4.6.3 Subaktivität „Repräsentiertes instabiles Widget finden“

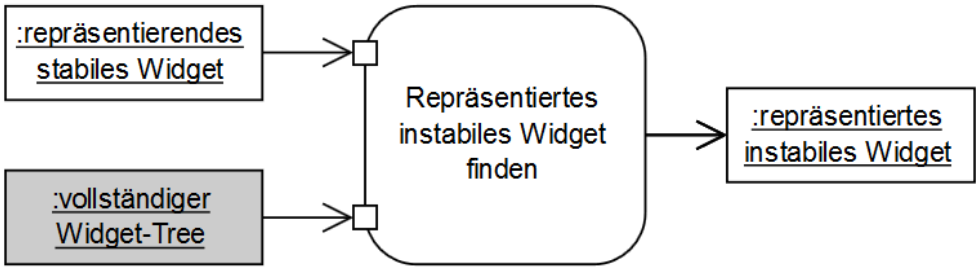


Abbildung 90: Phase 4 „Tests ausführen“ – Subaktivität „Repräsentiertes instabiles Widget finden“

Die Subaktivität „Repräsentiertes instabiles Widget finden“ (vgl. Abbildung 90) nimmt das repräsentierende stabile Widget und sucht das repräsentierte Widget im gegebenen vollständigen Widget-Tree. Das gefundene Widget ist das zu bedienende Widget des laufenden SUTs. Es wird für folgende Subaktivitäten bereitgestellt.

4.6.3.1 Beispiel

Abbildung 91 zeigt beispielhaft ein Widget, wie es durch Subaktivität „Repräsentiertes instabiles Widget finden“ ermittelt werden könnte.

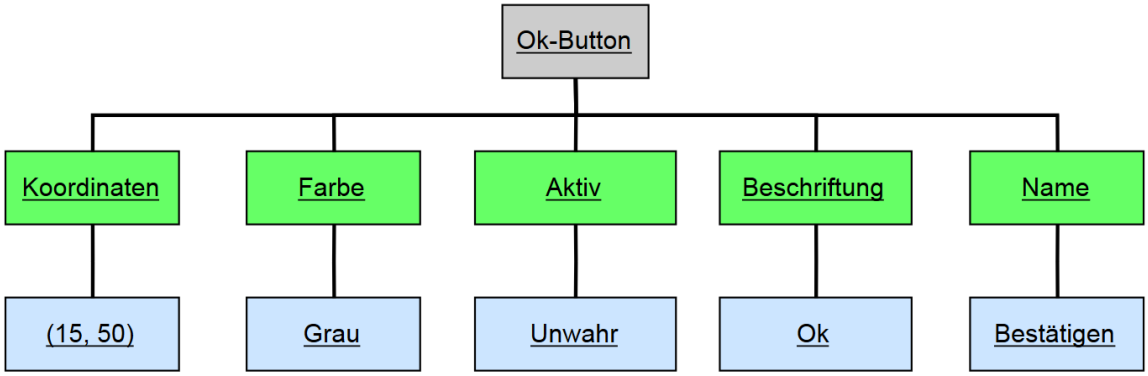


Abbildung 91: Beispiel eines zu bedienenden Widgets, wie aus dem SUT ausgelesen wird.

4.6.4 Subaktivität „Aktion durchführen“

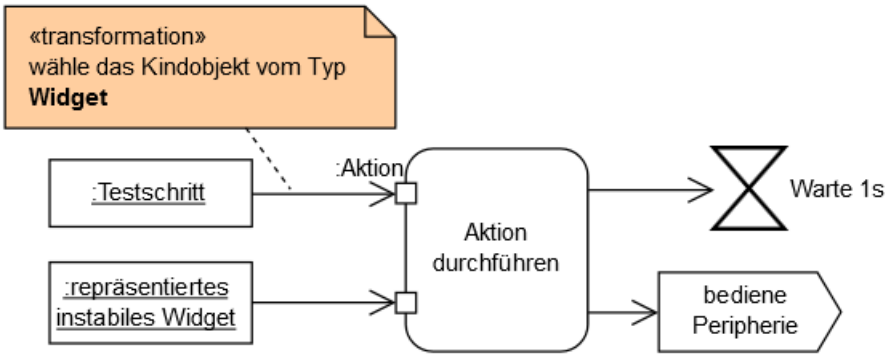


Abbildung 92: Phase 4 „Tests ausführen“ – Subaktivität „Aktion durchführen“

In Subaktivität „Aktion durchführen“ (vgl. Abbildung 92) wird die Aktion aus dem aktuell auszuführenden Testschritt auf dem gegebenen Widget ausgeführt, indem Eingaben der Peripherie simuliert werden.

Dazu wird die Aktion – nach Tabelle 2 (Seite 90) – in ihre entsprechenden Eingaben zerlegt. Diese Eingaben werden dann auf der Peripherie ausgeführt.

Nach dem Ausführen der Aktion wird kurz gewartet, um dem SUT Zeit zum Reagieren und Aufbauen der neuen GUI zu geben.

Diese Subaktivität ist konzeptionell relativ einfach. Die komplexere technische Umsetzung wird später in Abschnitt 5.3.6 beschrieben.

4.6.4.1 Beispiel

Ist beispielweise der Ok-Button aus Abbildung 91 zusammen mit der Aktion „Klick“ gegeben, dann wird die Aktion Klick in die zwei Eingaben „MouseDown“ und „MouseUp“ zergliedert. Es werden nun also die folgenden zwei Eingaben ausgeführt:

- MouseDown bei Koordinate (15, 50),
- MouseUp bei Koordinate (15, 50).

Es wird nun also tatsächlich ein Mausklick auf dem Ok-Button der GUI durchgeführt.

4.6.5 Subaktivität „SUT stoppen“

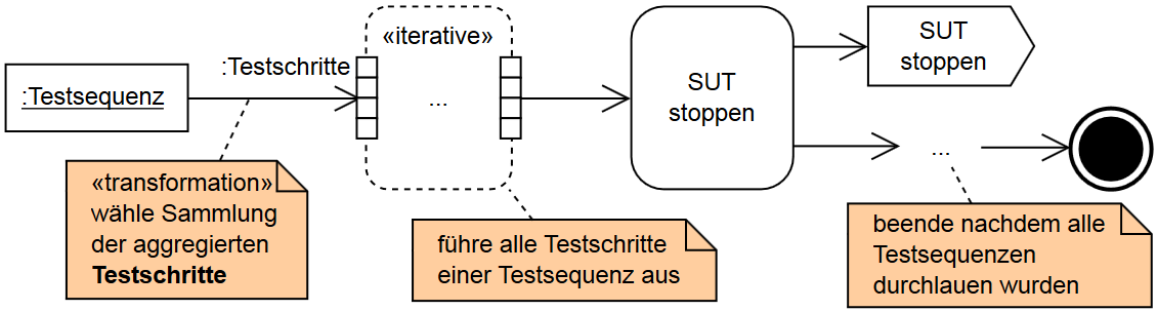


Abbildung 93: Phase 4 „Tests ausführen“ – Subaktivität „SUT stoppen“

Die Subaktivität „SUT stoppen“ (vgl. Abbildung 93) wird ausgelöst, sobald alle Testschritte einer Testsequenz ausgeführt wurden. Diese Subaktivität stoppt das SUT. Danach kann die nächste Testsequenz ausgeführt werden.

Die Subaktivität „SUT stoppen“ ist konzeptionell einfach. Die komplexere technische Umsetzung wird später in Abschnitt 5.3.3 beschrieben.

4.6.5.1 Beispiel

Der Windows-Taschenrechner wird geschlossen und ist nicht mehr auf dem Bildschirm sichtbar.

4.6.6 Zusammenfassung von Phase 4

Phase 4 „Tests ausführen“ nutzt die Testsequenz-Spezifikationen der dritten Phase und führt diese automatisch auf dem SUT aus. Dazu wird für jeden Testschritt die GUI gescannt und die spezifizierten Aktionen auf den spezifizierten Widgets ausgeführt. Vor einer Testsequenz wird das SUT jeweils gestartet und danach beendet. In dieser Phase wird auf Teile des Verhaltens von Phase 1 „Capture“ zurückgegriffen. Wie die Testsequenzen im Detail ausgeführt werden, wurde in diesem Abschnitt beschrieben.

Nach Beenden dieser Phase endet ein Testzyklus.

4.7 Zusammenfassung Gesamtkonzept

In diesem Abschnitt wurden die Klassifikationsbaum-Methode, Zustandsautomaten, Widget-Trees und die Capture/Replay-Methode zu einer neuen Methode zusammengeführt. Dabei wurden Modelle vorgestellt, welche das Zielmodell (vgl. Abschnitt 3) annähern und das Verhalten um diese Modelle automatisch aus einem GUI-System zu generieren. Außerdem wurde gezeigt, wie auf dem Modell automatisch Testsequenzen generiert werden können, welche danach automatisch auf der GUI ausgeführt werden können.

5. Umsetzung

In Abschnitt 4 wurde ein Konzept für ein GUI-Testverfahren vorgestellt, welches den Testprozess vom Aufnehmen von Testsequenzen, über die Modellerzeugung und Testfallgenerierung bis zur automatischen Testausführung beschreibt.

Dieses Verfahren wurde im Rahmen dieser Arbeit prototypisch implementiert. Mit dem entstandenen Werkzeug kann ein kompletter Zyklus vom Aufnehmen einer Testsequenz bis zum automatischen Abspielen generierter Testfälle durchlaufen werden. Im Rahmen dieser Arbeit wurden nicht alle Details und Sonderfälle des Konzeptes implementiert. Zum Beispiel wurde auf die Behandlung von Rechts-Klicks oder Checkboxen verzichtet. Trotz einiger fehlender Details genügt die Umsetzung, um die technische Umsetzbarkeit des Konzeptes zu zeigen.

5.1 Features

Auch Bauersfeld und Vos [BaVo12b] implementieren ein Werkzeug zum Testen von GUI-System. Es heißt GUITest. Ihr Werkzeug stellt folgende Feature bereit, welche auch unsere Implementation bietet:

- Funktioniert auf allen nativen GUIs, welche von der Windows API erkannt werden.
- SUTs müssen nicht instrumentiert werden.
- Erlaubt dem Benutzer, eigene Aktionen zu definieren.
- Erzeugte Testsequenzen können gespeichert und wieder abgespielt werden.

Anders als GUITest bietet unser Werkzeug folgende Eigenschaften:

- GUITest ist spezialisiert auf Robustheitstests. Das heißt es sucht selbstständig zufällige Testsequenzen durch die GUI, ohne dass diese unbedingt realistisches oder zielführendes Nutzerverhalten darstellen. Es geht darum Fehler zu finden. Unsere Implementation ist vor allem für funktionales Testen nützlich. Hier geht es darum zu

testen, ob konkrete Anforderungen erfüllt sind und das SUT seinen vorgesehenen Zweck erfüllt.

- Im Vergleich zu GUITest visualisiert unsere Implementation das Modell der GUI und Testsequenzen (vgl. Abschnitt 4.2).
- Anders als GUITest unterstützt unser Werkzeug bei der systematischen Auswahl sinnvoller Testsequenzen.

Auch Memon et al. [MeBN03a] bieten – ähnlich wie diese Arbeit – eine Implementation zum modellbasierten GUI-Testen, welche prototypisch Capture, die halb-automatische Modellierung und automatisierte Ausführung implementiert. Während Memon et al. die SUTs mit *GUI forests*, *event-flow graphs* und *integration trees* modellieren (vgl. Abschnitt 1.7), nutzt diese Arbeit Zustandsautomaten, Widget-Trees und Klassifikationsbäume. Damit implementiert diese Arbeit die zweite große Familie von modellbasierten GUI-Testmethoden.

5.2 Eingabedaten

5.2.1 Capture

Um Testsequenzen aufzunehmen werden einige Eingabedaten benötigt. Da in Phase 1 „Capture“ (vgl. 4.3) das SUT automatisch gestartet und gescannt werden muss, wird ein Dateipfad zum ausführbaren SUT benötigt. Diese Datei wird dann automatisch mithilfe des Betriebssystems gestartet und die zugehörige GUI wird gescannt.

Weiterhin werden die aufzunehmenden Nutzereingaben benötigt. Das in Abschnitt 4 beschriebene Gesamtkonzept ermöglicht die Verarbeitung verschiedener Eingaben, wie Mausklicks und Tastaturanschläge. Die Implementation, welche im Rahmen dieser Arbeit entstanden ist, beschränkt sich vorerst auf Links-Klicks mit der Maus.

5.2.2 Testausführung

Um Testsequenzen auf einem SUT automatisch ausführen zu können, müssen wie in [BaVo12b] der Pfad zur ausführbaren Datei (exe-Datei) und ggf. weitere Informationen

zum Zurücksetzen des SUTs angegeben werden. Mittels dieser Informationen kann dann, wie schon beim Capture-Prozess beschrieben (Abschnitt 5.2.1), das SUT gestartet und gescannt werden.

Zur automatischen Bedienung der GUI sind keine zusätzlichen Eingabedaten erforderlich, sondern lediglich die Daten, die schon aus den vorherigen Phasen bekannt sind.

5.3 Implementation

Die erarbeitete Lösung wurde als Plug-In für TESTONA implementiert und funktioniert für alle GUIs[40] in Windows-Betriebssystemen.[41] Zu diesem Zweck wurden verschiedene existierende Arbeiten – wie Frameworks und Bibliotheken – wiederverwendet. Die Eigenleistung der hier vorgestellten Implementation liegt also, neben der Umsetzung neuer Algorithmen und Technologien, darin, wie bestehende Technologien kombiniert werden, um die Methode aus Abschnitt 4 zu verwirklichen.

5.3.1 Überblick

Im Rahmen früherer Arbeiten wurde von Berner & Mattner Systemtechnik GmbH[42] eine allgemeine Struktur für GUI-Testprozesse definiert (vgl. [KrNF14, NaKr13]). Dieses Framework ist in Abbildung 94 zu sehen. Dieses Framework wird im Folgenden *Test-Framework* genannt.

[40] In Abschnitt 4.2.5 sind bereits konzeptuelle Grenzen aufgezeigt, welche dafür sorgen, dass bestimmte Arten von GUIs nur schlecht modelliert werden. Dennoch kann die entstandene Implementation auch auf diese GUIs angewandt werden.

[41] Die Implementation könnte später für zusätzliche Betriebssysteme erweitert werden. Auf diese Erweiterung wurde allerdings im Rahmen dieser Arbeit verzichtet, da während der Entstehung dieser Arbeit eine der benötigten Bibliotheken nur für Windows mit geeigneten Lizenzen verfügbar war. Im Laufe der Arbeit wurde auch die Bibliothek JNativeHook (https://code.google.com/p/jnativehook/) veröffentlicht, welche sich u.U. für die Erweiterung auf zusätzliche Betriebssysteme eignet.

[42] http://www.berner-mattner.com/

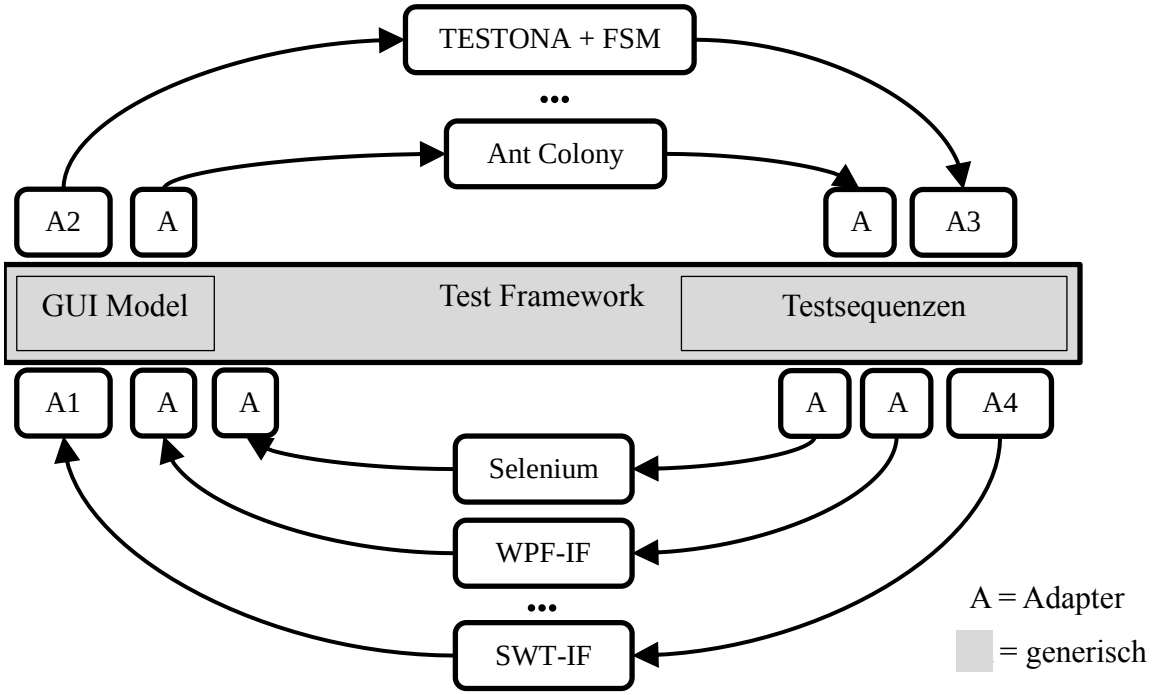


Abbildung 94: Test-Framework für allgemeine GUI-Test Prozesse, nach [KrNF14, NaKr13].

Die erarbeitete Methode wurde so implementiert, dass sie diesem Test-Framework genügt. In diesem Test-Framework wird zuerst das SUT (unten) von einem Adapter A1 gescannt, welcher für eine spezielle SUT-Technologie ausgelegt ist. Adapter A1 speichert die Informationen über die GUI in einem allgemeinen GUI-Modell (eine XML-Datei[43], links grau abgebildet). Dann liest ein Adapter A2 dieses GUI-Modell und transformiert die Informationen in eine Form, welche von einer konkreten Testgenerierung weiterverwendet werden kann (in unserem Fall vom TESTONA, oben abgebildet). Diese Software generiert Testsequenzen und Adapter A3 transformiert diese Sequenzen in eine allgemeine, technologieunabhängige Form (eine XML-Datei, rechts abgebildet). Zuletzt ist Adapter A4 dafür zuständig, diese Testsequenzen auf dem SUT auszuführen.

Der erhoffte Nutzen dieses Test-Frameworks ist es, die verschiedenen Technologien leicht austauschbar zu machen. So könnte man beispielsweise einfach die Windows-GUIs dieser Arbeit gegen Web-GUIs einer vorherigen Arbeit austauschen, indem lediglich Adapter A1 und A4 geändert werden.

[43] XML steht für *Extensible Markup Language* (engl. „erweiterbare Auszeichnungssprache“) und dient zur serialisierten Darstellung von hierarchisch strukturierten Daten.

Die in Abschnitt 4 vorgestellt Methode fügt sich gut in dieses Framework ein. Dabei implementiert *Adapter A1* die Schritte aus *Phase 1*, *Adapter A2* die Schritte aus *Phase 2*, usw. Das Verhalten und die Funktionalität dieser Adapter ist durch die Phasen und Modelle bereits beschrieben (vgl. Abschnitt 4). Dieser Abschnitt beschreibt nun die grobe Struktur dieser Adapter, die eingebundenen Bibliotheken und die genutzten und erzeugten Modellimplementationen.

Zur Einbettung des Verfahrens musste außerdem TESTONA erweitert werden. So mussten zum Beispiel neue Buttons, Menüs und Dialoge bereitgestellt werden, um neue GUI-Test-Projekte anzulegen, um den Capture-Prozess zu starten oder um die automatische Testausführung zu starten. Allgemeiner formuliert musste also der (nicht dargestellte) Steuerprozess zur Lenkung des Zyklus aus Abbildung 94 implementiert werden.

5.3.2 Die Modellimplementationen

Die Implementation verbindet überwiegend vorhandene Technologien, Bibliotheken und Frameworks, welche alle ihre eigenen, unabhängigen Modelle bereitstellen. Eine wichtige Aufgabe der Implementation ist es, diese Modelle ineinander zu überführen und kompatibel zu machen. Oft treten für viele der in Abschnitt 4.2 konzeptionell vorgestellten Modelle mehrere Implementationen auf, welche in diesem und den hierauf folgenden Unterabschnitten genauer erläutert werden.

5.3.2.1 GUI-Modell, Capture-Sequenzen und Testsequenzen

Das technologieunabhängige GUI-Modell und die allgemeine Testsequenz-Spezifikation des Test-Frameworks waren bereits als XML-Spezifikationen implementiert (vgl. [KrNF14, Nasa13]). Diese Implementationen wurden im Rahmen dieser Arbeit wiederverwendet und angepasst. So wurde im Rahmen dieser Arbeit das ehemals provisorische Zustandskonzept zu vollwertigen Klassen für Zustände und Transitionen aufgewertet, bisher generische Eigenschaften zu wiederverwendbaren GUI-Eigenschaften konkretisiert und die Modularität erhöht, um beispielsweise Widget-Trees und Zustandsautomaten in verschiedenen Dateien zu persistieren.

Die XML-Spezifikation des GUI-Modells beschreibt sowohl das konzeptuelle GUI-Modell (Abschnitt 4.2.1), inklusive Widget-Trees, als auch die Capture-Sequenzen (Abschnitt 4.2.2), obwohl es im Test-Framework nur „GUI-Modell“ heißt. Diese verwirrende Namensgebung ist auf die frühere Test-Framework-Implementation zurückzuführen, in denen Sequenzen nur eine untergeordnete Rolle spielten und daher einfach als Teil des GUI-Modells angesehen wurden.

Die Testsequenz-Spezifikation des Test-Frameworks entspricht im Wesentlichen dem erforderlichen Modell für Testsequenzen (vgl. Abschnitt 4.2.4). Für die Testsequenzen existieren damit zwei Modelle: die allgemeine Testmatrix der Klassifikationsbaum-Methode und die GUI-spezifische Testsequenz-Spezifikation im Test-Framework.

Die Spezifikationen liegen im XSD-Format (XML Schema Definition) vor. Konkrete Instanzen der Modelle, zum Beispiel ein Modell des Windows-Taschenrechners, haben also das XML Format. Diese konkreten Modelle können somit gegen das Schema validiert werden.

Um programmatisch auf den Modellen arbeiten zu können wurde *JAXB (Java Architecture for XML Binding)* verwendet. Mittels x*jc (Binding Compiler)* wurde so einmalig aus den Modellspezifikationen Java-Quelltext erzeugt. Die so entstandenen Java-Klassen werden benutzt, um die Modelle zu manipulieren, auszulesen, aus XML-Dateien zu laden oder sie in XML-Dateien zu speichern.

5.3.2.2 Abstrahiertes GUI-Modell

Das abstrahierte GUI-Modell (Abschnitt 4.2.3) verwendet verschiedene Teilmodelle, von denen einige schon vorhanden waren, während andere als Teil dieser Arbeit entwickelt wurden.

Klassifikationsbaum

Ein Modell des Klassifikationsbaums ist bereits als Teil von TESTONA vorhanden (vgl. [KrLu10]). Es liegt in Form von Java-Klassen vor. Die Klassifikationsbäume werden nur

innerhalb von TESTONA benötigt, daher gibt es genau eine Implementation der Klassifikationsbäume. Diese Implementation wird unverändert für diese Arbeit verwendet.

Zustandsautomat

Für den Zustandsautomaten existieren drei verschiedene Implementationen.

Das GUI-Modell aus dem Test-Framework (vgl. Abschnitt 5.3.2.1) kann nichtorthogonale, nichthierarchische Zustandsautomaten abbilden. Diese reichen aus, um das einfache GUI-Modell (vgl. Abschnitt 4.2.1) und die Capture-Sequenzen (vgl. Abschnitt 4.2.2) zu modellieren, jedoch nicht für das abstrahierte GUI-Modell (vgl. Abschnitt 4.2.3).

TESTONA enthält ein weiteres Modell für einen Zustandsautomaten (vgl. Abbildung 95) [KrWe12]. Dieses Modell kann zwar Orthogonalität und Hierarchie modellieren, hat jedoch einige Nachteile. So haben Zustände und Regionen des Modells keine eigenen Java-Klassen, sondern greifen nur auf die Modellbestandteile des Klassifikationsbaumes zurück. Zustände werden als (Klassifikationsbaum-)Klassen und Regionen werden als Klassifikationen modelliert. Auch für Transitionen gibt es keine eigenen Java-Klassen. Stattdessen werden alle Transitionen in einem String kodiert, welcher, mittels String-Operationen, interpretiert und manipuliert wird. Die modellierten Zustandsautomaten können bereits grafisch repräsentiert werden, jedoch immer nur eine Region und hierarchische Ebene gleichzeitig. Da die Zustandsautomaten bereits umfassend in TESTONA integriert sind und die vorhandenen Implementationen der Testsequenzgenerierung (Phase 3, vgl. Abschnitt 4.5) auf diesem Modell arbeiten, wird dieses Modell weiterverwendet.

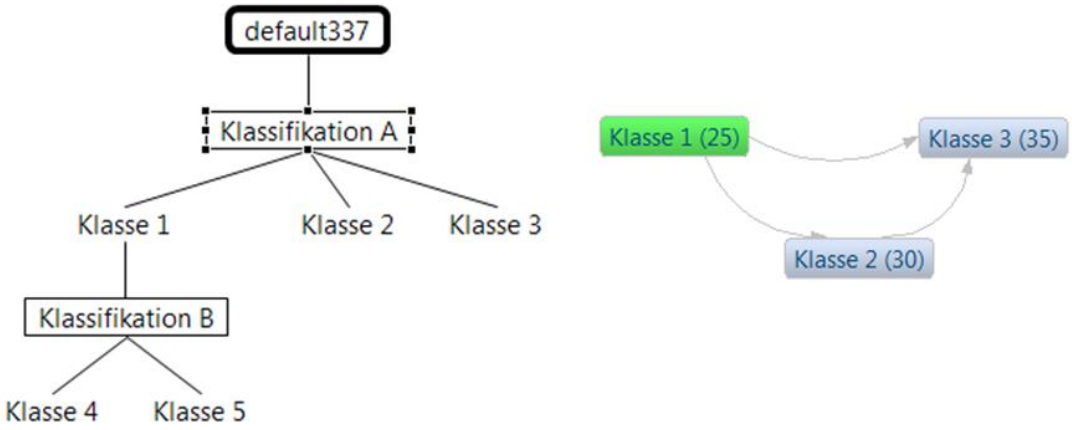


Abbildung 95: TESTONAs Modell für Zustandsautomaten ist eng mit dem Klassifikationsbaum gekoppelt. Zustände sind Klassen, Regionen sind Klassifikationen. Alle Transitionen werden in einem String gespeichert. Dargestellt wird stets nur eine Region („Klassifikation A“ im Bild) und eine Hierarchieebene (*direkte* Kindelemente von „Klassifikation A“).

Auch das dritte Modell repräsentiert einen orthogonalen, hierarchischen Zustandsautomaten. Es wurde im Rahmen dieser Arbeit entwickelt und soll die Schwächen von TESTONAs Modell überwinden. Dazu wurde ein einfaches Metamodell für den Zustandsautomaten entworfen (vgl. Abbildung 96). Dieses Modell wurde in *EMF (Eclipse Modelling Framework)* spezifiziert. Daraus wurde Java-Quelltext generiert. Die so entstandenen Java-Klassen werden benutzt, um die Modelle zu manipulieren, auszulesen. Die Java-Klassen können auch genutzt werden, um Modelle aus XML-Dateien zu laden oder sie in XML-Dateien zu speichern.

Die grafische Darstellung des dritten Modells für den Benutzer wurde basierend auf Graphiti[44] implementiert. So kann der komplette Zustandsautomat, inklusive Orthogonalität und Hierarchie, dem Benutzer dargestellt und von diesem manipuliert werden (vgl. Abbildung 97). Zur Unterscheidung von den beiden anderen Modellen wird das dritte Modell im Folgenden auch *Graphiti-Zustandsautomat* genannt.

[44] http://www.eclipse.org/graphiti/

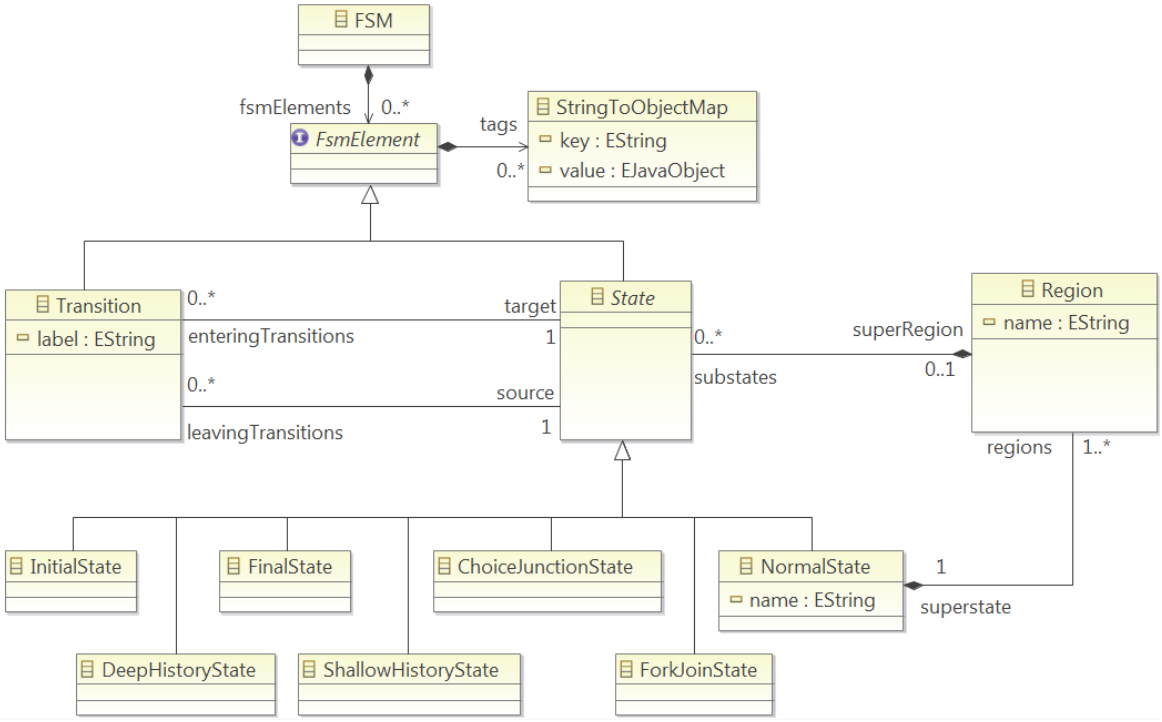


Abbildung 96: Verwendetes Metamodell für die orthogonalen, hierarchischen Zustandsautomaten. Spezifiziert in EMF.

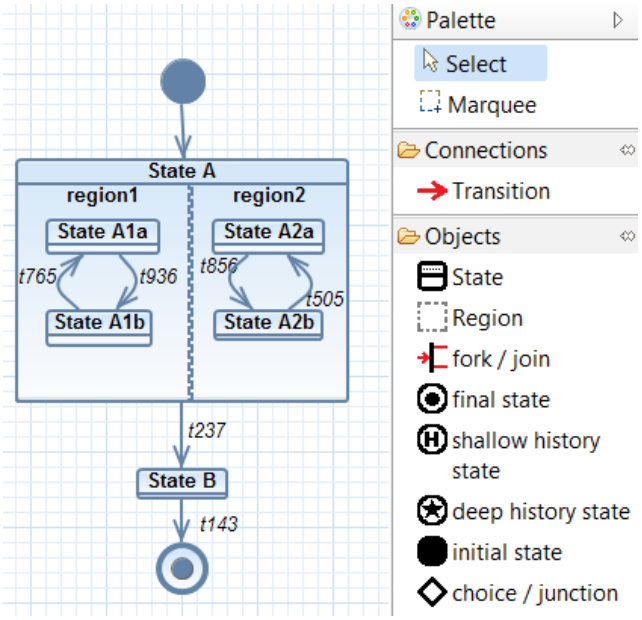


Abbildung 97: Der Graphische Editor des orthogonalen, hierarchischen Zustandsautomaten wurde basierend auf Graphiti implementiert.

Orthogonale Widget-Trees

Für das abstrahierte GUI-Modell wird eine Implementation der orthogonalen Widget-Trees (vgl. Abschnitt 4.2.3.1) benötigt.

Klassische (nichtorthogonale) Widget-Trees waren bereits im GUI-Modell des Test-Frameworks implementiert. Als Teil dieser Arbeit wurde diese Implementation erweitert, um auch Widget-Tree-Faktoren modellieren zu können. Dazu musste ein weiterer Knotentyp hinzugefügt werden: die Verbindungspunkte.

Einen kompletten orthogonalen Widget-Tree, welcher aus mehreren Widget-Tree-Faktoren besteht, als eigene Klasse oder XML Schema Definition zu modellieren, war für diese Arbeit nicht nötig. Stattdessen werden die Faktoren Zuständen zugeordnet oder in geeigneten Java `Collection`s gehalten.

5.3.3 Adapter A1

Der Adapter A1 – welcher dafür verantwortlich ist den Capture-Prozess zu ermöglichen und dabei ein beliebiges GUI-System zu scannen und in ein technologieunabhängiges Modell zu speichern – war bisher nicht implementiert und wurde im Rahmen dieser Arbeit erstmalig entwickelt.

Ein Überblick der von Adapter A1 verwendeten Bibliotheken und Modelle ist in Abbildung 98 zu sehen. Detaillierte Beschreibungen folgen in den nächsten Abschnitten.

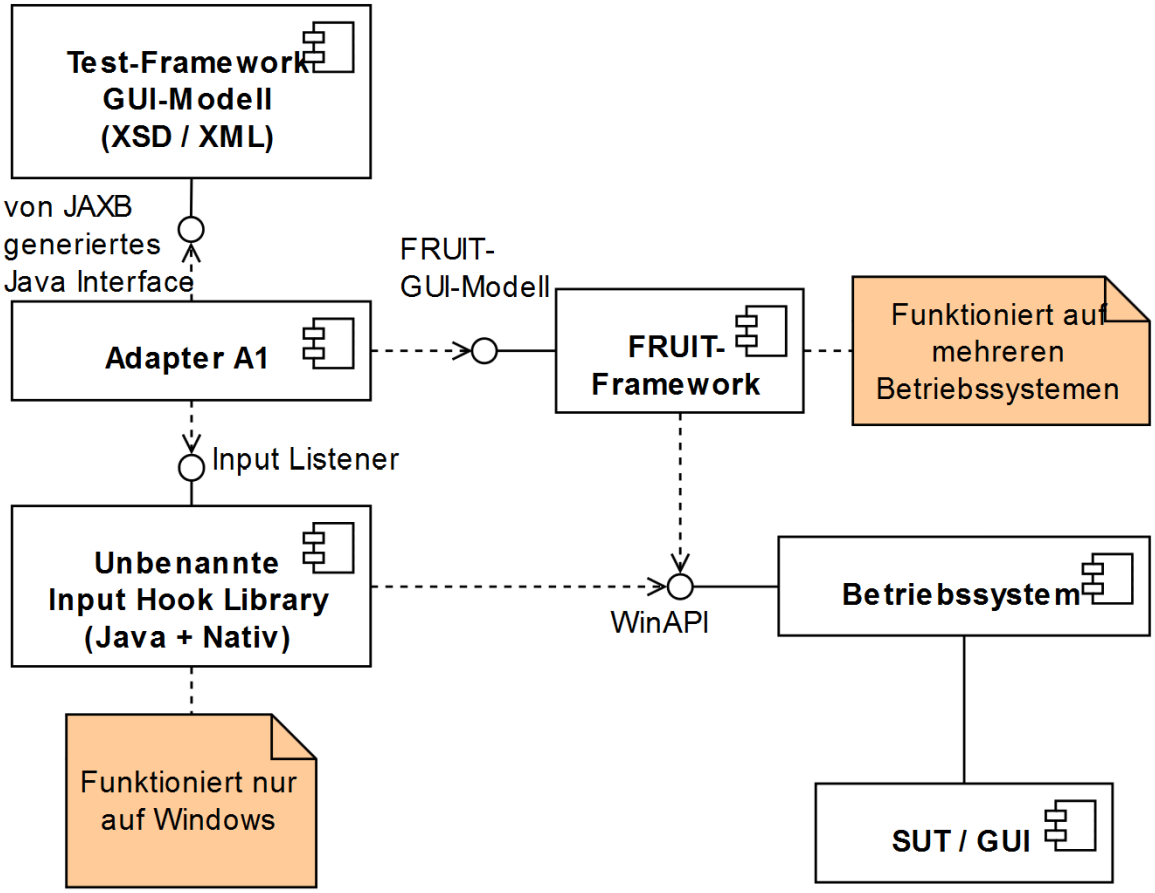


Abbildung 98: Überblick der von Adapter A1 verwendeten Bibliotheken und Modelle

5.3.3.1 GUIs scannen

Um sicherzustellen, dass Adapter A1 jede beliebige GUI scannen kann, wurde das FRUIT-Framework[45] benutzt. Es stellt ein einheitliches Auslesen verschiedener GUIs mittels Java-Methoden bereit. Das FRUIT-Framework ist noch in Entwicklung und wird zukünftig voraussichtlich noch weiter verbessert. Derzeit stellt das FRUIT-Framework bereits Zugriff auf viele Aspekte verschiedener GUIs bereit.

[45] Das FRUIT-Framework wird derzeit an der Universität Politecnica de Valencia entwickelt und im Rahmen des FITTEST Projekts (http://crest.cs.ucl.ac.uk/fittest/) verwendet. Das FRUIT-Framework ist bisher nicht veröffentlicht, jedoch basiert es auf dem GUITest-Framework von Bauersfeld und Vos [BaVo12b].

FRUIT stellt die ausgelesene GUI in einem eigenen Widget-Tree-Modell bereit.[46] Dieses wird von Adapter A1 in das Widget-Tree-Modell des Frameworks transformiert (vgl. Abbildung 98).

Das FRUIT-Framework hat noch kleinere Defizite. Innerhalb dieser Arbeit wurden diese Defizite hingenommen, da es den Rahmen der Arbeit gesprengt hätte, einen zusätzlichen eigenen Zugang zur GUI bereitzustellen.

Beispielsweise fehlen einzelne Widgets im von FRUIT bereitgestellten Widget-Tree. So fehlt beim Windows-Taschenrechner das Widget mit den zuletzt eingegebenen Termen (vgl. Abbildung 99). Ob dieses Problem seine Ursache im FRUIT-Framework, in der Windows API oder in der Implementation des Taschenrechners hat, wurde nicht untersucht. Durch den Austausch von FRUIT gegen Bibliotheken mit bildbasierten Verfahren zum Auslesen der GUI könnte dieses Problem vermieden werden.

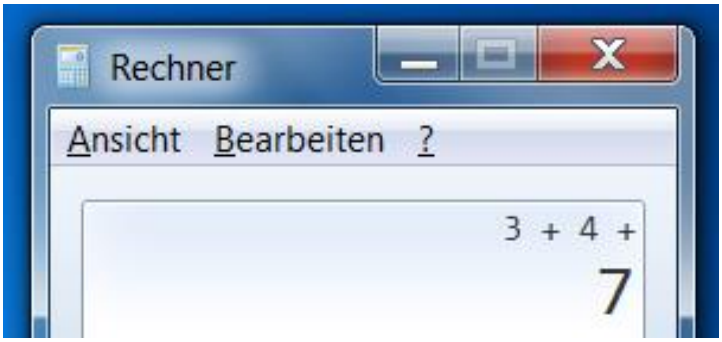


Abbildung 99: Die vom FRUIT-Framework bereitgestellten Widget-Trees sind teils unvollständig. So fehlt beim Windows-Taschenrechner das Widget mit den zuletzt eingegebenen Termen (im Bild „3 + 4 +“).

5.3.3.2 Capture

Die Implementierung der Capture-Funktionalität für beliebige GUIs ist nicht trivial. Da das entwickelte TESTONA Plug-In in Java implementiert ist, kann es nicht mit einfachen Sprachmitteln die Nutzereingaben beobachten, welche außerhalb von TESTONA stattfinden.

[46] In Abbildung 98 *FRUIT-GUI-Modell* genannt.

Deshalb wird eine Bibliothek[47] verwendet, welche betriebssystemweit alle Nutzereingaben registriert. Diese Bibliothek nutzt einige native DLL-Dateien und das *Java Native Interface* (JNI), um native Nutzereingaben in Windows[48] für Java-Programme beobachtbar zu machen. Diese Bibliothek wird benutzt um Benutzereingaben zu erkennen. Für die Benutzereingaben stellt die Bibliothek ein eigenes Modell in Form von Java-Klassen bereit. Adapter A2 transformiert daher die Nutzereingaben-Modelle der Bibliothek in das Modell des Test-Frameworks (vgl. Abbildung 98).

Obwohl diese Bibliothek auf Windows beschränkt ist, erweitert diese Arbeit den Funktionsumfang nicht über die verwendeten Bibliotheken hinaus. Das bedeutet speziell, dass das entwickelte Werkzeug nicht wirklich in der Lage ist beliebige GUIs zu testen, sondern vorerst auf GUIs in Windows-Betriebssystemen beschränkt ist.

In Verbindung mit dem FRUIT-Framework kann aus den Nutzereingaben hergeleitet werden, welches Widget auf welche Art im SUT benutzt wurde. Der Capture-Mechanismus baut also auf beiden Bibliotheken auf (vgl. Abbildung 98). Sowohl Capture als auch Replay wurden im Rahmen dieser Arbeit entwickelt.

Die Zuordnung einer Nutzereingabe – im Prototyp eines Mausklicks – zu einem bestimmten Widget (vgl. Abschnitt 4.3.7) erfolgt über die Bildschirmkoordinaten des Klicks und des Widgets. Koordinaten basierte Verfahren zum GUI-Testen gelten als besonders instabil. Da diese Koordinaten in unserer Implementation jedoch nur im Moment der Nutzereingabe relevant sind, und nach Zuordnung des Widgets sofort verworfen werden, resultieren daraus keine Instabilitäten.

[47] Die Bibliothek hat keinen Namen. Sie kann gefunden werden unter http://kra.lc/blog/2011/07/java-global-system-hook/.

[48] Diese Bibliothek ist der einzige Grund, weshalb das entwickelte Plug-In nur für das Betriebssystem Windows funktioniert. Alle anderen Komponenten sind Betriebssystem unabhängig modelliert. Im Laufe der Fertigstellung dieser Arbeit wurde JNativeHook (https://code.google.com/p/jnativehook/) veröffentlicht, jedoch nicht mehr eingebunden. JNativeHook stellt eine vergleichbare Funktionalität bereit, jedoch auch für andere Betriebssysteme, wie Mac OS X und Linux.

5.3.4 Adapter A2

Ein Adapter A2 – welcher verantwortlich ist für das Bereitstellen der Daten für TESTONA – wurde bereits im Rahmen einer früheren Arbeit von Nasarek [Nasa13] entwickelt, welche sich mit dem Testen von Webseiten beschäftigt.

Im Rahmen dieser Arbeit wurde ein Adapter A2 implementiert, welcher zusätzlich zum bisherigen Adapter auch Capture-Sequenzen auswertet und daraus zusätzlich orthogonale, hierarchische Zustandsautomaten erzeugt.

Ein Überblick der von Adapter A2 verwendeten Bibliotheken und Modelle ist in Abbildung 100 zu sehen.

Abgesehen vom Verhalten (vgl. Abschnitt 4.4) implementiert Adapter A2 auch die Modelltransformationen zwischen den verschiedenen Implementationen des Zustandsautomaten (vgl. Abschnitt 5.3.2.2). So wird der nichthierarchische, nichtorthogonale Zustandsautomat des Test-Frameworks eingelesen und in den Graphiti-Zustandsautomaten transformiert. Dieser wird – wie in Phase 2 beschrieben – abstrahiert. Der so entstandene hierarchische, orthogonale Zustandsautomat liegt weiterhin als Graphiti-Zustandsautomat vor. Diesen transformiert Adapter A2 dann in die TESTONA Modelle für Klassifikationsbaum und Zustandsautomat.

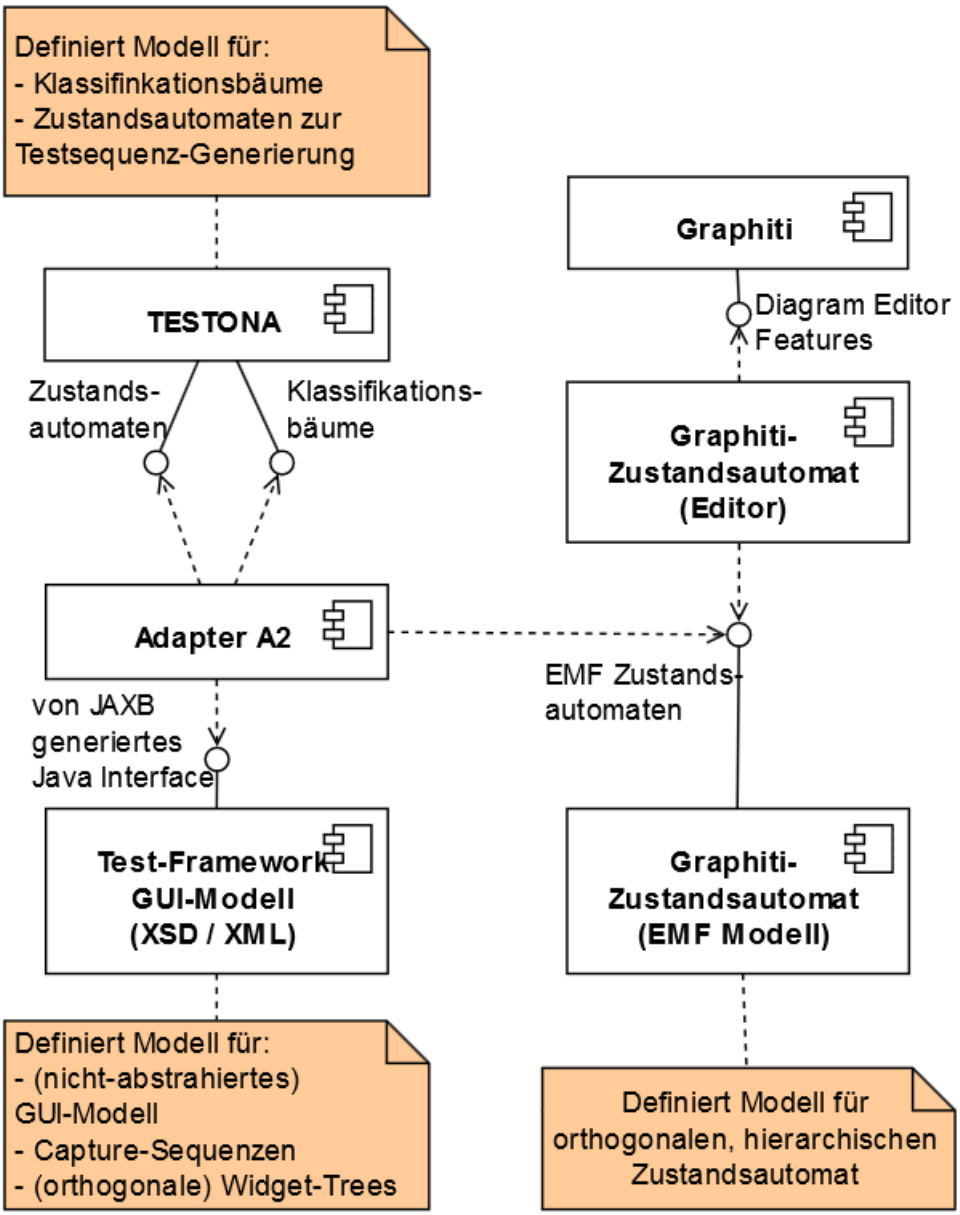


Abbildung 100: Überblick der von Adapter A2 verwendeten Bibliotheken und Modelle

5.3.5 Adapter A3

Adapter A3 ist dafür verantwortlich, die von TESTONA bereitgestellten Testsequenzen in ein technologieunabhängiges Modell zu schreiben.

Ein anderer Adapter A3 wurde bereits im Rahmen einer früheren Arbeit entwickelt, welche sich mit dem Testen von Webseiten beschäftigt (vgl. [Nasa13]). Eine Wiederverwendung und Anpassung dieses Adapters für beliebige GUI-Systeme fand nicht statt, da eine Neuimplementierung effizienter schien.

Ein Überblick der von Adapter A3 verwendeten Bibliotheken und Modelle ist in Abbildung 101 zu sehen.

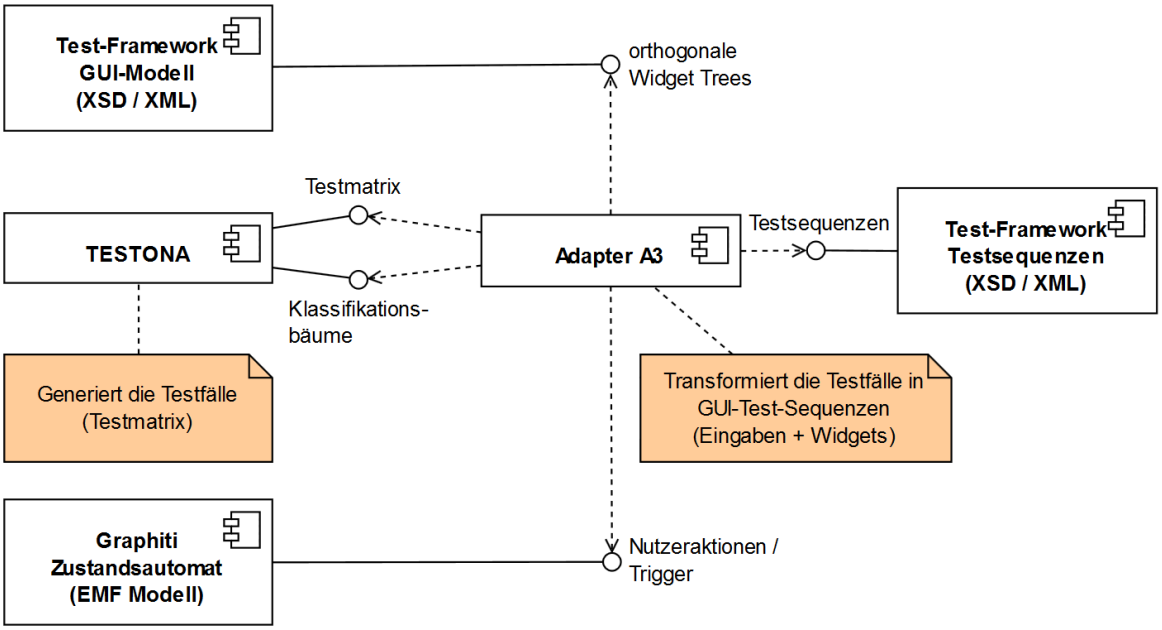


Abbildung 101: Überblick der von Adapter A3 verwendeten Bibliotheken und Modelle

Anders als die anderen Adapter implementiert A3 nicht die komplette Phase 3, sondern nur die zweite Hälfte dieser Phase. Während die erste Hälfte der Phase 3, die Generierung der Testsequenzen beschreibt, werden diese in der Implementation bereits von TESTONA selbst generiert. Diese Testsequenzen liegen dann in Form einer als Testmatrix vor (vgl. Abschnitt 4.5.2).

Adapter A3 transformiert diese Testmatrix dann nur noch in das Modell für Testsequenzen (vgl. Abschnitte 4.5.3 und 5.3.2.1) aus dem Test-Framework. Für diese Transformation sind auch die Nutzeraktionen aus dem EMF Zustandsautomaten und die Widgets aus den orthogonalen Widget-Trees nötig.

5.3.6 Adapter A4

Der Adapter A4 – welcher verantwortlich ist für die automatische Ausführung von Testsequenzen auf einem beliebigen GUI-System – war bisher nicht implementiert und wurde im Rahmen dieser Arbeit erstmalig entwickelt.

Ein Überblick der von Adapter A4 verwendeten Bibliotheken und Modelle ist in Abbildung 102 zu sehen.

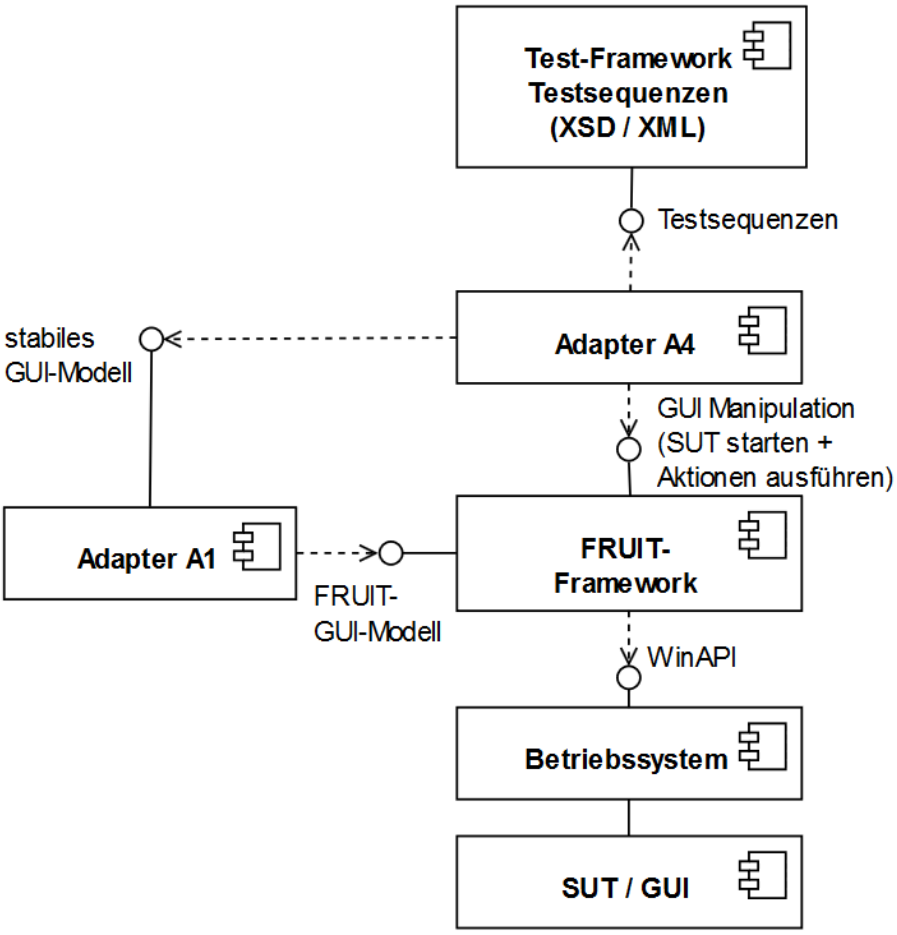


Abbildung 102: Überblick der von Adapter A4 verwendeten Bibliotheken und Modelle

Bevor das SUT automatisch bedient wird, muss die GUI gescannt und stabilisiert werden. Dazu werden Teile von Adapter A1 wiederverwendet. Die Widgets aus den Testsequenzen werden dann im gerade ermittelten stabilen Widget-Tree gesucht, um zu bedienende Widgets zu identifizieren.

Um sicherzustellen, dass Adapter A4 jede beliebige GUI bedienen kann, wurde auch hier das FRUIT-Framework benutzt (vgl. Abschnitt 5.3.3). Adapter A4 kann das Framework nutzen, um darüber das SUT zu starten. Nachdem damit dann eine GUI gescannt wurde, stellt FRUIT einheitliche Java-Methoden zum Bedienen der gescannten GUI bereit.

So wird der Replay-Mechanismus realisiert und die spezifizierten Testsequenzen ausgeführt, indem immer abwechselnd die GUI gescannt wird und danach automatisch bedient wird.

Im Vergleich zum Capture-Mechanismus (vgl. 5.3.3.2) benötigt der Replay-Mechanismus keine Bibliothek zum Registrieren der Benutzereingaben.

5.3.7 TESTONA

Das Vorhandensein der Adapter und Modelle allein reicht nicht aus, um ein GUI-System zu testen. Zusätzlich wird noch *„Glue Code“* benötigt, welcher die Adapter ins Werkzeug TESTONA integriert, sie dem Tester zugänglich macht und in der richtigen Reihenfolge ausführt.

Ein Überblick der Integration der verschiedenen Adapter ist in Abbildung 103 zu sehen.

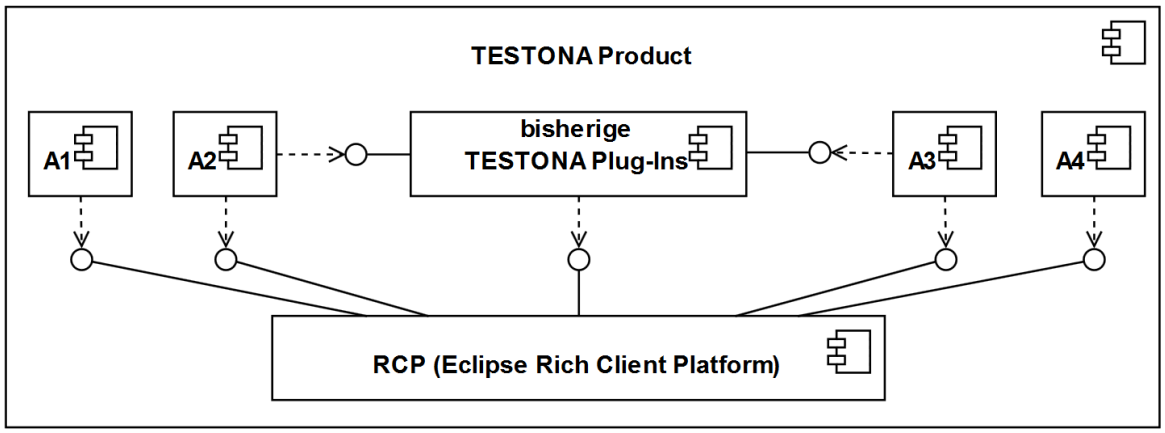


Abbildung 103: Überblick der Integration der vier Adapter in TESTONA

Das bisherige TESTONA selbst basiert auf Eclipse RCP, dem gleichen Framework welches der Eclipse IDE zugrunde liegt. Das heißt, das bisherige TESTONA Werkzeug besteht aus mehreren Plug-Ins, welche mittels RCP integriert sind und zu einer Software vereint sind.

Gleichermaßen wurden die implementierten Adapter und Modelle in Plug-Ins verpackt und dem TESTONA Produkt hinzugefügt.

Zusätzlich wurde – als Teil der Plug-Ins – Funktionalität implementiert, welche die Prozesse der Adapter anstößt. Dieses Anstoßen wird dem Tester über Buttons und Menüs

zugänglich gemacht, welche beim RCP Framework registriert werden müssen und von diesem dadurch automatisch entsprechend dargestellt und angeordnet werden. Außerdem wurde Funktionalität implementiert und zugänglich gemacht, welche die Erzeugung und das Speichern von GUI-Test-Projekten ermöglicht. Einige der so entstandenen Buttons und Menüs sind in Abbildung 104 bis Abbildung 107 zu sehen.

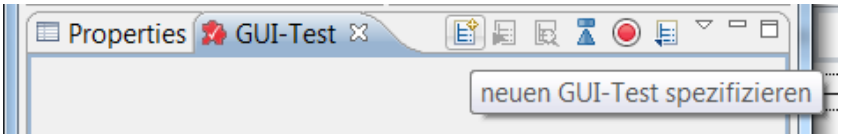


Abbildung 104: Eine View (so heißen Registerkarten inklusive Inhalt in RCP) stellt Buttons bereit, um neue GUI-Tests zu spezifizieren und die Prozesse der verschiedenen Adapter anzustoßen.

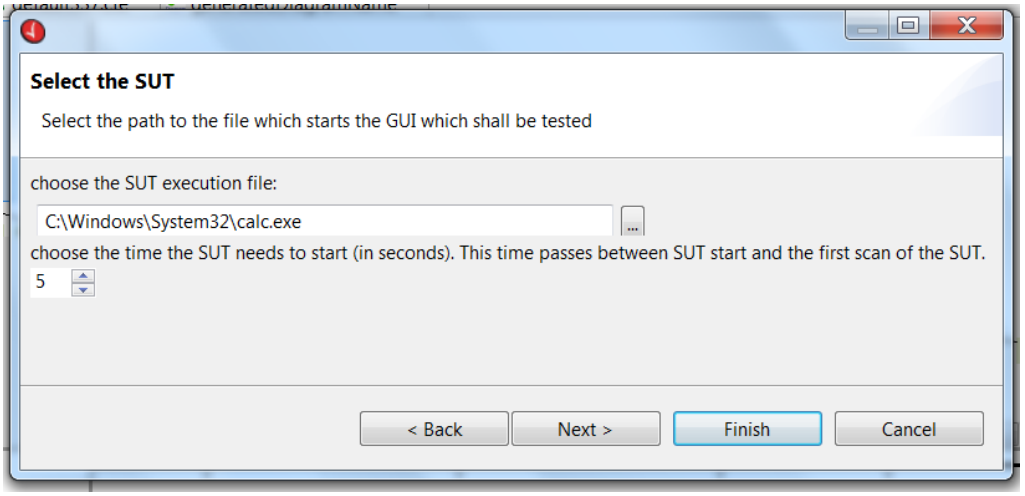


Abbildung 105: Im Wizard für neue GUI-Test-Projekte lässt sich das SUT, mittels eines Dateipfades spezifizieren.

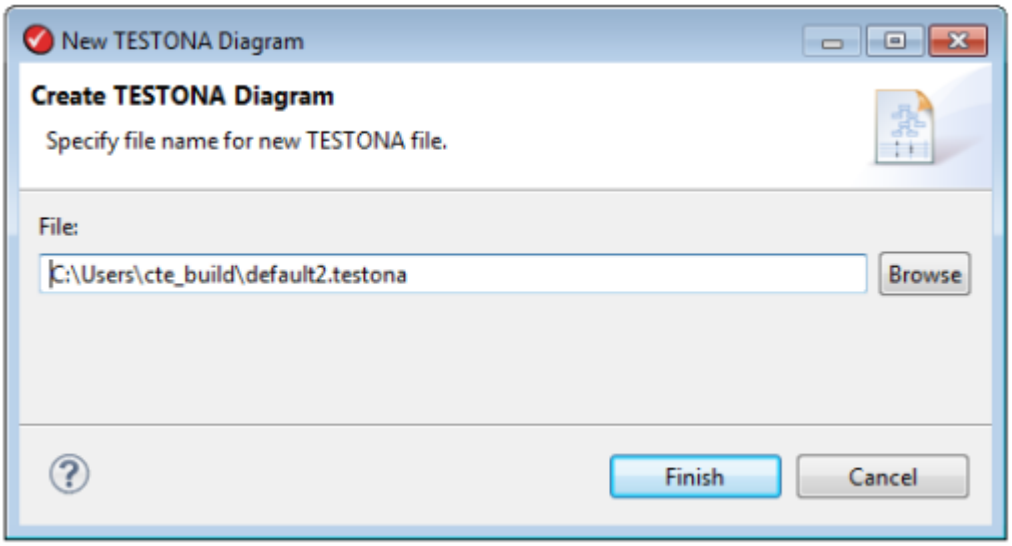


Abbildung 106: Das GUI-Test-Projekt wird – wie auch andere TESTONA-Projekte – in der gewünschten `cte`-Datei gespeichert.

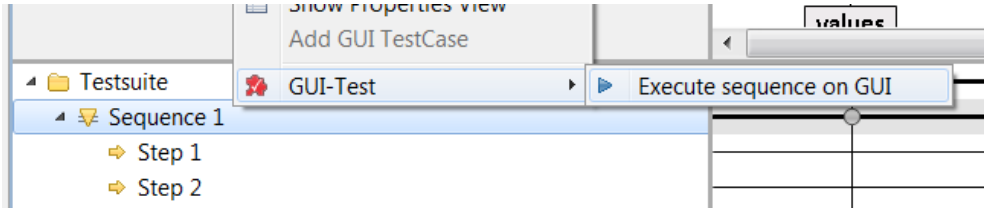


Abbildung 107: Einzelne Testsequenzen der Testmatrix können über ein entsprechendes Kontextmenü automatisch ausgeführt werden.

6. Auswertung

Die Implementation des entwickelten Verfahrens wurde quantitativ und qualitativ am GUI-System Windows-Taschenrechner evaluiert. Die Evaluation soll erste Hinweise über die Praxistauglichkeit des entwickelten Testentwurfsprozess und über die Qualität des Prototyps liefern.

Dabei sind vor allem drei Modelle auszuwerten: der orthogonale, hierarchische Zustandsautomat, die Widget-Trees und der Klassifikationsbaum. Ein Schwerpunkt wurde bei der Auswertung stark auf den Zustandsautomaten gelegt. Da die Elemente des Klassifikationsbaums weitestgehend mit denen des Zustandsautomaten übereinstimmen, wären die gewonnen Erkenntnisse weitestgehend analog. Zum Klassifikationsbaum werden daher nur die Besonderheiten behandelt. Die Widget-Trees sind – bis auf Orthogonalität – keine Eigenleistung dieser Arbeit, weshalb nur ihre Neuerungen evaluiert werden.

Neben der automatischen Erzeugung der einzelnen Modelle ist die Hauptleistung dieser Arbeit in der Kombination der Modelle zu einem Gesamtkonzept zu sehen. Ein weiterer Schwerpunkt der Auswertung ist deshalb die Bewertung, wie gut diese Kombination gelungen ist, mit Rückblick auf die anfängliche Problemstellung (vgl. Abschnitt 1.4).

6.1 Vorgehensweise

Um die Evaluierung objektiv und messbar zu machen, wurde nicht einfach willkürlich in einer beliebigen GUI „umhergeklickt“. Stattdessen wurde ein Tutorial [Gont13] des Windows-Taschenrechners als Spezifikation der aufzunehmenden und zu modellierenden Aktionen und Reihenfolge festgelegt. Details darüber sind in Abschnitt 3 beschrieben.

6.2 Mehrstufige Soll- und Ist-Werte

In Abschnitt 3 ist ein Idealmodell des SUTs beschrieben. Dieses Idealmodell wurde frei von techischen Zwängen und mithilfe von Kontextwissen des Testers ermittelt. Es diente als Zielstellung bzw. durch die Methode anzustrebendes Ergebnis.

Bei strikter Ausführung der beschriebenen Methode (vgl. Abschnitt 4) wird das Idealmodell nicht vollkommen ideal rekonstruiert. Stattdessen wurde eine automatisierbare Methode ohne Vorraussetzung von Kontextwissen und unter den Einschränkungen der Mathematisierung entworfen.

Vor der Implementierung wurde die Methode manuell auf das SUT angewendet[49] und so manuell ein Modell dieses SUTs konstruiert. Dieses Modell selbst dient als Erwartungswert für die Evaluierung der Implementation. Das entstandene Modell ist in Abbildung 108 zu sehen.

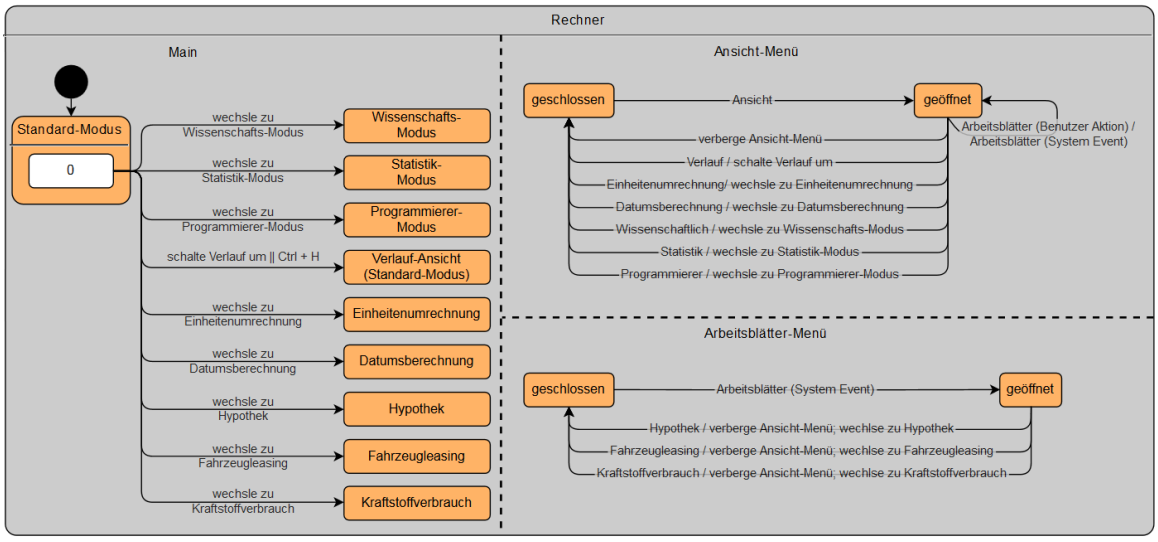


Abbildung 108: Ausschnitt des manuell konstruierten Modells, nach stoischer Abarbeitung des Taschenrechner-Tutorials und manueller Anwendung der entwickelten Methode. Dieses Modell dient als Erwartungswert für die Evaluierung der Implementation. Das vollständige – und besser lesbare – Modell ist im Anhang B zu finden.

Es erfolgt also eine Evaluierung in zwei Schritten. Zum einen wird das manuell erzeugte Modell (Abbildung 108) gegen das Idealmodell (Abschnitt 3) evaluiert. Zum anderen wird

[49] Das heißt die Tutorials wurden darauf ausgeführt.

das Ergebnis der Implementation gegen das manuell erzeugte Modell evaluiert. Nachfolgend werden die drei verglichenen Modelle kurz *Idealwert*, *Erwartungswert* und *Realwert* bzw. *Idealmodell*, *Erwartungsmodell* und *Realmodell* genannt. Die vollständigen Modelle sind in den Anhängen A, B und C zu finden.

In den nächsten Abschnitten wird zuerst das Erwartungsmodell gegen das Idealmodell evaluiert und dann das Realmodell gegen das Erwartungsmodell.

6.3 Vergleich Erwartungsmodell gegen Idealmodell

Die Unterschiede zwischen Erwartungsmodell und Idealmodell werden in den folgenden Unterabschnitten herausgestellt. Die Modelle sind, abgesehen von den hier erwähnten Unterschieden, im Wesentlichen gleich, weshalb die Gemeinsamkeiten nicht extra aufgeschlüsselt werden.

6.3.1 Keine Wächterausdrücke

Im Idealmodell werden teilweise Wächterausdrücke (vgl. 2.2.3) genutzt, welche einen Zustandswechsel abhängig von intern gespeicherten Werten ausführen. Die Variablen und Wächterausdrücke wurden nicht in die Methode aufgenommen, da dies den Rahmen dieser Arbeit gesprengt hätte. Welche Auswirkung das Weglassen der Wächterausdrücke auf den Erwartungswert hatte, soll am nachfolgenden Beispiel gezeigt werden.

Schaltet man im Taschenrechner den Verlauf ein, tippt dann eine Zahl und dann die „=“-Taste, so wird die getippte Zahl im Verlauf angezeigt (Abbildung 109). Tippt man die gleiche Zahl und „=“ ohne den aktivierten Verlauf, so verhält sich der Taschenrechner anders. Um dieses Verhalten im Idealmodell zu modellieren, wurde der Zustandsübergang beim Drücken von „=“ mit einem Wächterausdruck versehen, welcher prüft, ob der Verlauf aktiviert ist (Abbildung 110, links). Im Erwartungsmodell existieren keine Variablen und Wächterausdrücke, wodurch zusätzliche Zustände entstehen (Abbildung 110, rechts).

Im Rahmen dieser Arbeit wurden verschiedene einfache Ansätze ausprobiert, um Variablen und Wächter in die Methode zu integrieren. Jedoch stellte sich bei allen bislang versuchten Ansätzen heraus, dass eine sinnvolle Erzeugung von Wächtern und Variablen aus einer Capture-Sequenz komplexer ist als anfangs angenommen. Ob das Problem ohne Kontextwissen überhaupt lösbar ist, ist unklar.

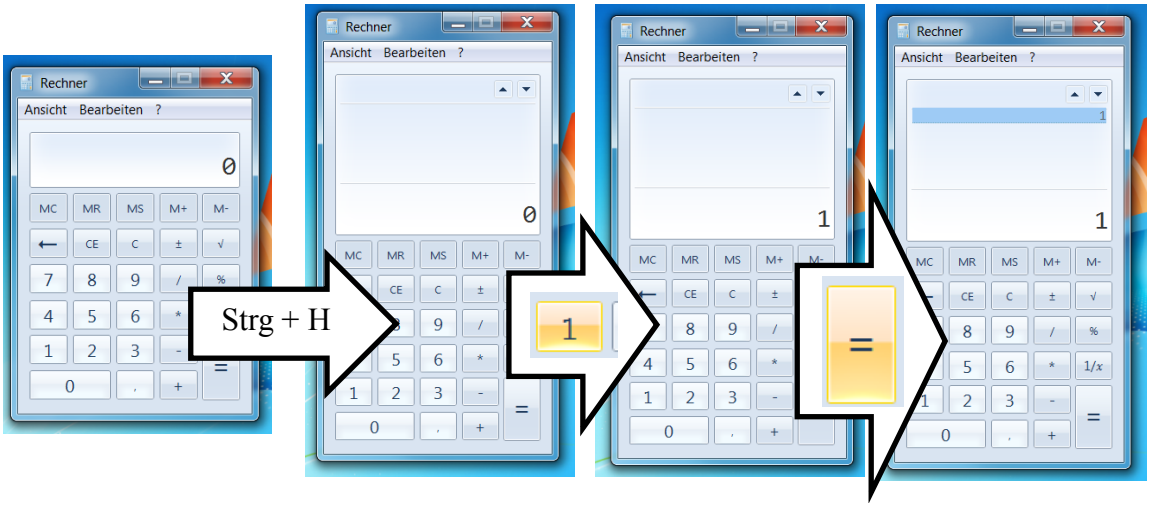


Abbildung 109: Eine Eingabesequenz aus dem Tutorial aktiviert den Verlauf, tippt dann eine Zahl und darauf die „=“-Taste. Dadurch wird die getippte Zahl zusätzlich oben im Verlauf angezeigt.

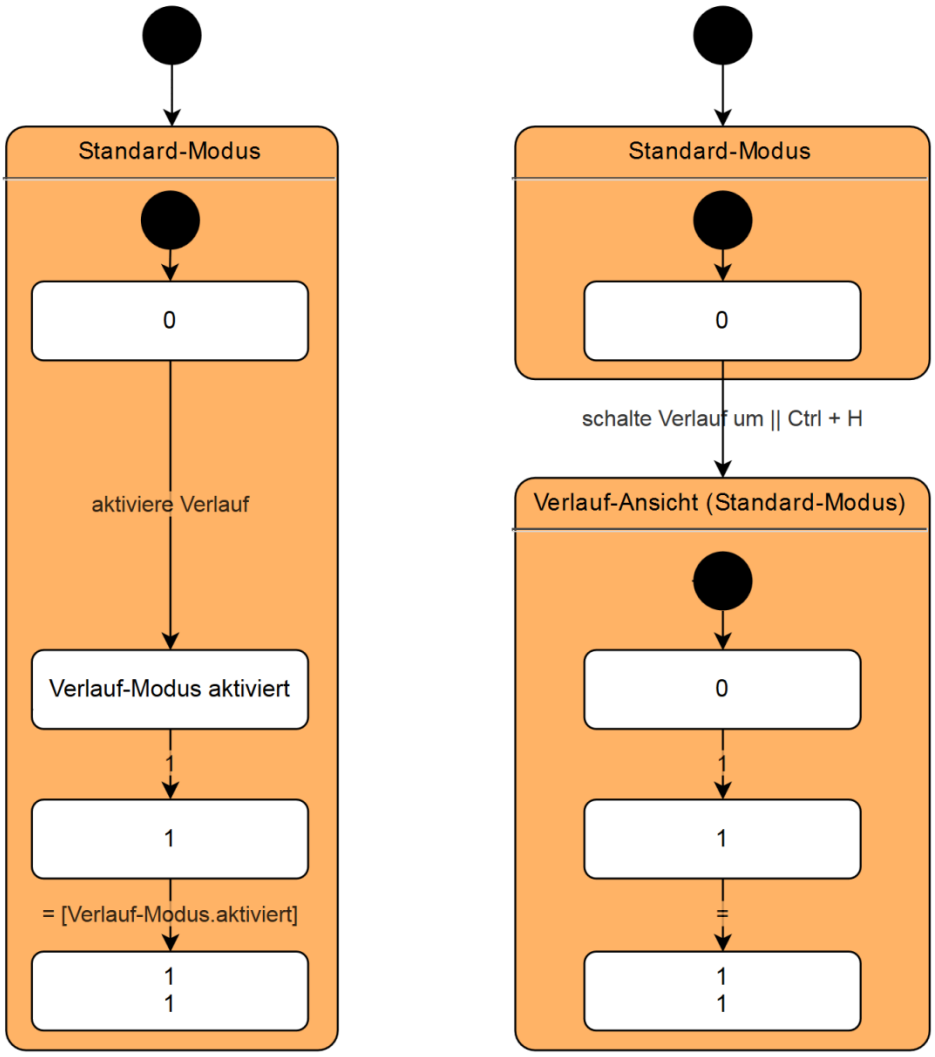


Abbildung 110: Im Idealmodell (links) gibt es Variablen und Wächterzustände. Im Erwartungsmodell (rechts) entstehen stattdessen neue Zustände – im Bild ein Struktur-Zustand – für jede mögliche Variablenbelegung.

6.3.2 Keine Erkennung interner Zustandswechsel

Da die entwickelte Methode ein SUT mittels seiner GUI testet, werden auch nur sichtbare Zustandswechsel als solche wahrgenommen.

Tippt man beispielweise im Taschenrechner eine Zahl und dann „M+“, so wird die Zahl zum internen Speicher addiert. Der interne Zustand ändert sich also. Im Idealmodell werden diese zwei Zustände separat modelliert (Abbildung 111, links). Da sich bei diesem Zustandswechsel nichts in der angezeigten GUI ändert, existiert auch nur ein Zustand im Erwartungsmodell (Abbildung 111, rechts).

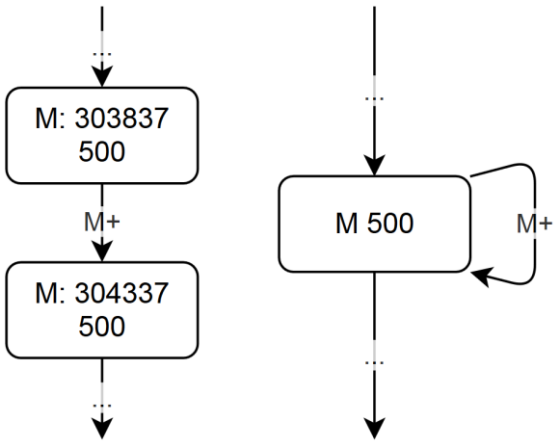


Abbildung 111: Interne Zustandswechsel werden mittels Kontextwissen im Idealmodell durch mehrere Zustände modelliert (links). Im Erwartungsmodell ist alles der gleiche Zustands, was gleich aussieht (rechts).

6.3.3 Keine Transitionen von Überzuständen

Harels *State Charts* erlauben Transitionen von Überzuständen zu ihren Unterzuständen (vgl. Abschnitt 2.2.3). Beispielsweise ist es im Taschenrechner egal, in welchem Unterzustand man sich befindet, man kann jederzeit ins Menü gehen und von dort aus in die verschiedenen Modi (Standard, Wissenschaftlich, etc.) wechseln. Im Idealmodell wird dies modelliert, indem vom Überzustand eine Transition in jeden Modus-Zustand verläuft (Abbildung 112, links). Im Erwartungsmodell starten die Transitionen nur in den tatsächlich verlassenen Unterzuständen (Abbildung 112, rechts). Das Erwartungsmodell bietet also weniger kombinatorische Möglichkeiten als das Idealmodell.

Um die Transitionen in den Überzuständen starten zu lassen ist Kontextwissen nötig. Man muss wissen, welche Transitionen jederzeit durchlaufen werden können und welche Transitionen nur aus bestimmten Zuständen heraus möglich sind.

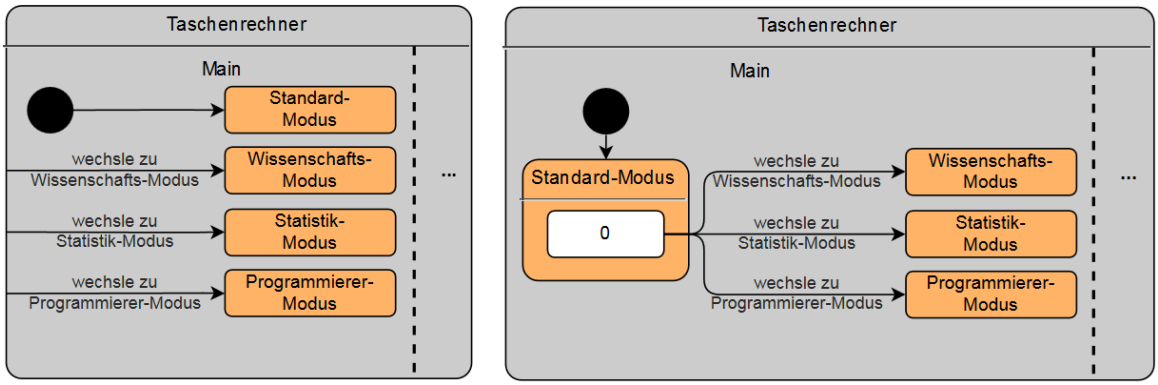


Abbildung 112: Im Idealmodell verlaufen Transitionen auch vom Überzustand zu seinen Unterzuständen (links), im Erwartungsmodell beginnen Transitionen stets im tatsächlich verlassenen Werte-Zustand (rechts).

6.3.4 Orthogonale Bereiche

Im Windows-Taschenrechner gibt es Formulare (wie zur Umrechnung von Einheiten), welche man ein- und ausblenden kann (Abbildung 113). Diese sind völlig entkoppelt vom eigentlichen Taschenrechner, sie beeinflussen den Taschenrechner nicht und werden auch nicht von ihm beeinflusst.

Diese Formulare wurden im Idealmodell daher als orthogonaler Bereich modelliert (Abbildung 114). Da Kontextwissen nötig ist um diese Orthogonalität zu erkennen, gibt es diesen zusätzlichen Bereich im Erwartungsmodell nicht (Abbildung 115).

Wie sinnvoll eine Anwendung ist, welche zwei völlig unabhängige Teilfunktionalitäten hat ist fragwürdig. Weil es sich dabei um einen Spezialfall handelt und weil Kontextwissen nötig ist, wurde im Rahmen dieser Arbeit nicht versucht, diese Orthogonalität in die Methode zu integrieren.

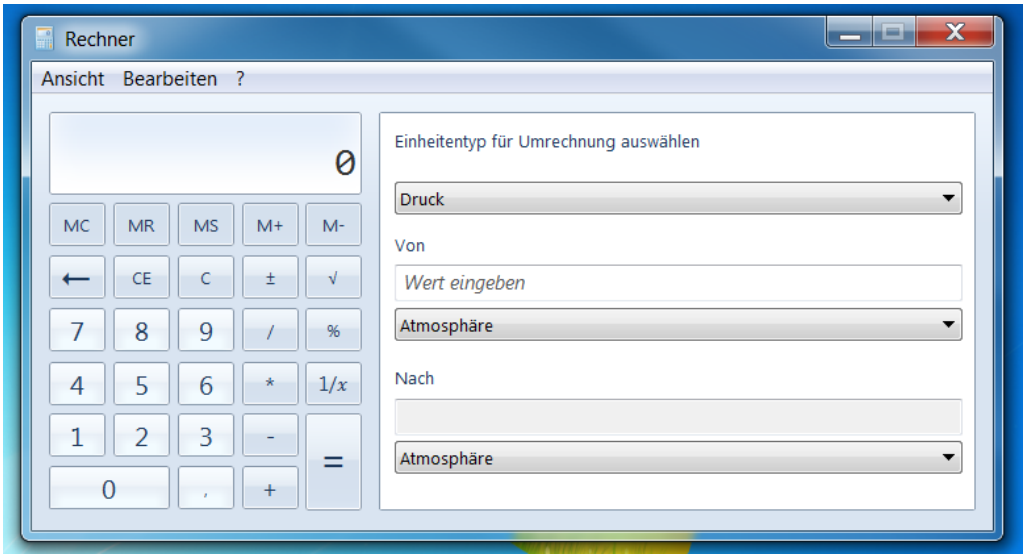


Abbildung 113: Der Windows-Taschenrechner bietet zwei völlig unabhängige Teilfunktionalitäten. Den normalen Taschenrechner (links) und verschiedene Berechnungs-Formulare (rechts).

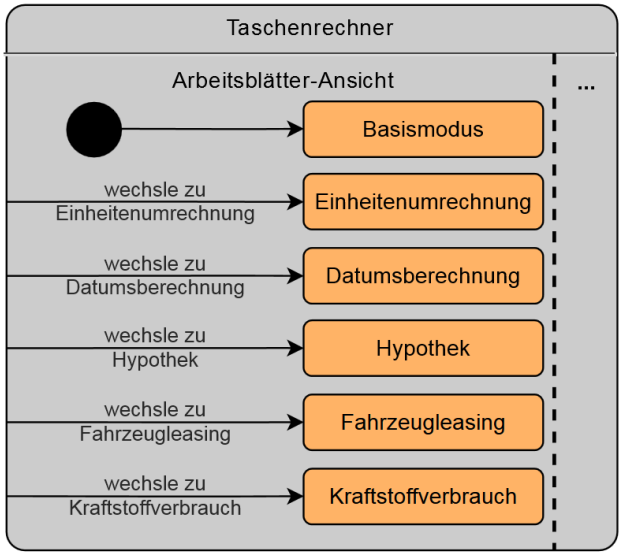


Abbildung 114: Im Idealmodell wird die unabhängige Teilfunktionalität des Taschenrechners in einem eigenen orthogonalen Bereich modelliert.

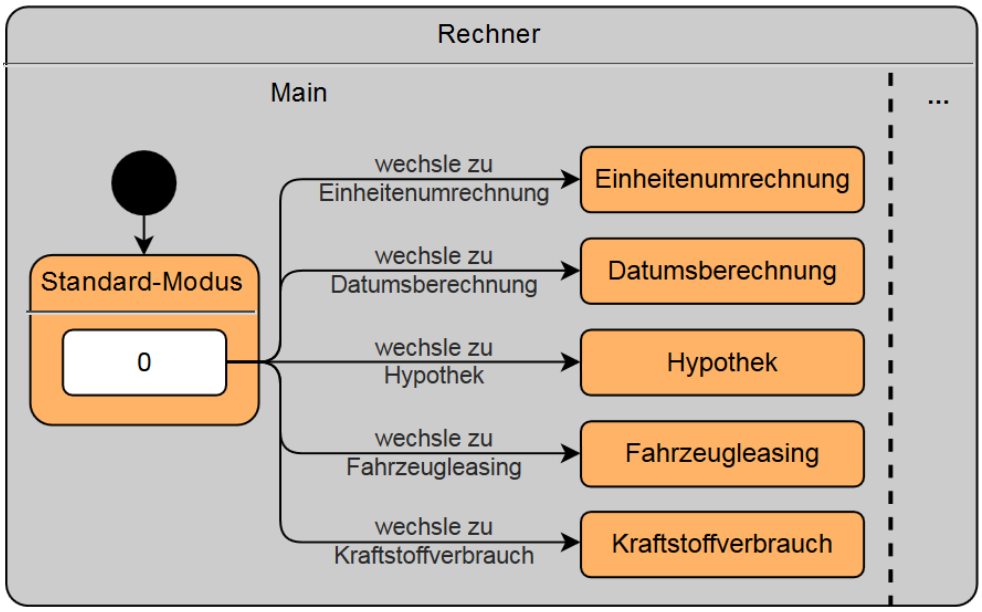


Abbildung 115: Im Erwartungsmodell wird die unabhängige Teilfunktionalität nicht als unabhängig erkannt, da Kontextwissen erforderlich wäre. Daher wird kein eigener orthogonaler Bereich modelliert.

6.4 Vergleich Realmodell gegen Erwartungsmodell

Die Unterschiede zwischen Realmodell und Erwartungsmodell werden in den folgenden Unterabschnitten herausgestellt. Die Modelle sind, abgesehen von den hier erwähnten Unterschieden, gleich, weshalb die Gemeinsamkeiten nicht extra aufgeschlüsselt werden.

6.4.1 Evaluiertes Realmodell

Aufgrund eines Problems in der entwickelten Methode, welches nicht mehr korrigiert wurde, wurde das Realmodell nur mit einer Untermenge des Erwartungsmodells verglichen. In den folgenden Unterabschnitten wird zuerst das Problem beschrieben und dann gezeigt, welches Realmodell gegen welches Erwartungsmodell evaluiert wurde.

6.4.1.1 Problem: verdoppelte Regionen

Während der Evaluierung stellte sich heraus, dass in der entwickelten Methode die orthogonalen Regionen teils nicht korrekt zusammengefasst werden. Dieses Problem wur-

de beim Entwickeln des Erwartungsmodells übersehen, da vor der Implementierung noch keine konkreten Widget-Trees vorlagen. Das nachfolgende Beispiel soll dieses Problem genauer erläutern.

Der Taschenrechner hat verschiedene Modi, in denen verschiedene Widgets dargestellt werden: zum Beispiel Standard und Wissenschaftlich (vgl. Abbildung 116). Man wechselt zwischen diesen Modi jedoch immer mittels des gleichen Ansichts-Menüs, unabhängig vom Ausgangs- und Ziel-Modus. Das Menü sollte also in einem orthogonalen Bereich modelliert werden (vgl. Abbildung 117). Tatsächlich wird es jedoch in mehreren Bereichen geöffnet, wenn es in verschiedenen Modi aufgenommen wurde (Abbildung 118).[50]

Zur Behebung dieses Fehlers wäre eine Anpassung in der Methode (Abschnitt 4) und der Implementation (Abschnitt 0) notwendig. Unter Umständen reicht es hier schon aus, eine zusätzliche Subaktivität beim Erzeugen der Orthogonalität (vgl. Abschnitt 4.4.3) hinzufügen. Es kann jedoch auch sein, dass die Definition des orthogonalen Widget-Trees (vgl. Abschnitt 4.2.3.1, insbesondere Abbildung 31) problematisch ist. Laut dieser Definition fasst ein orthogonaler Widget-Tree mehrere Faktoren unter sich zusammen. Zur korrekten Modellierung ist es jedoch nötig, dass Faktoren von mehreren dieser orthogonalen Widget-Trees „wiederverwendet“ werden. In anderen Worten: gleiche Faktoren müssen identisch sein, unabhängig von ihrem orthogonalen Widget-Tree. Wie genau dieses Problem letztlich behoben wird, muss in zukünftigen Arbeiten geklärt werden. Abbildung 119 bietet einige Lösungsansätze.

[50] Dieses falsche Modell mit duplizierten Regionen ermöglicht beispielsweise den inkonsistenten Zustand, in dem das Menü gleichzeitig geöffnet und geschlossen ist.

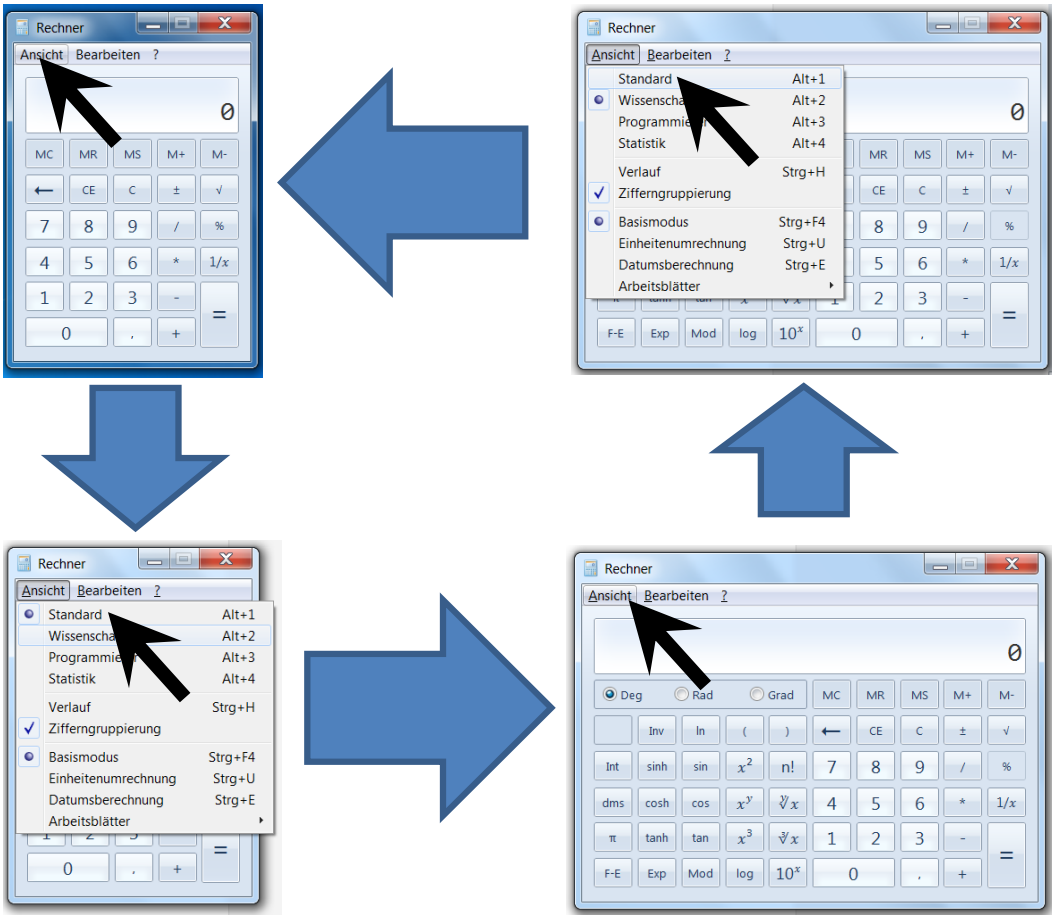


Abbildung 116: Der Taschenrechner hat verschiedene Modi, in denen verschiedene Widgets dargestellt werden: zum Beispiel Standard und Wissenschaftlich. Man wechselt zwischen diesen Modi jedoch immer mittels des gleichen Menüs, unabhängig vom Ausgangs- und Ziel-Modus.

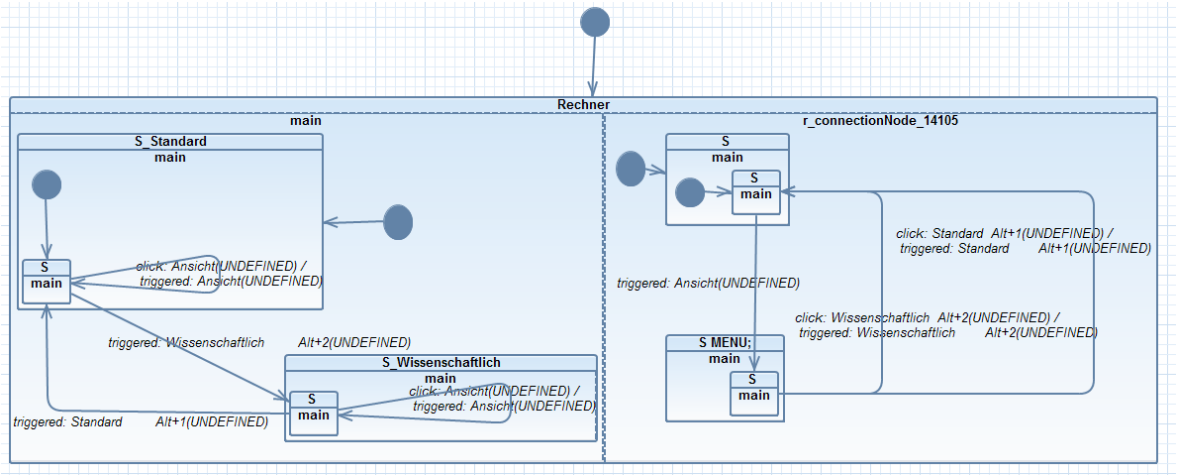


Abbildung 117: Das Ansichts-Menü des Taschenrechners sollte einen orthogonalen Bereich haben, unabhängig vom Modus in dem es geöffnet ist.

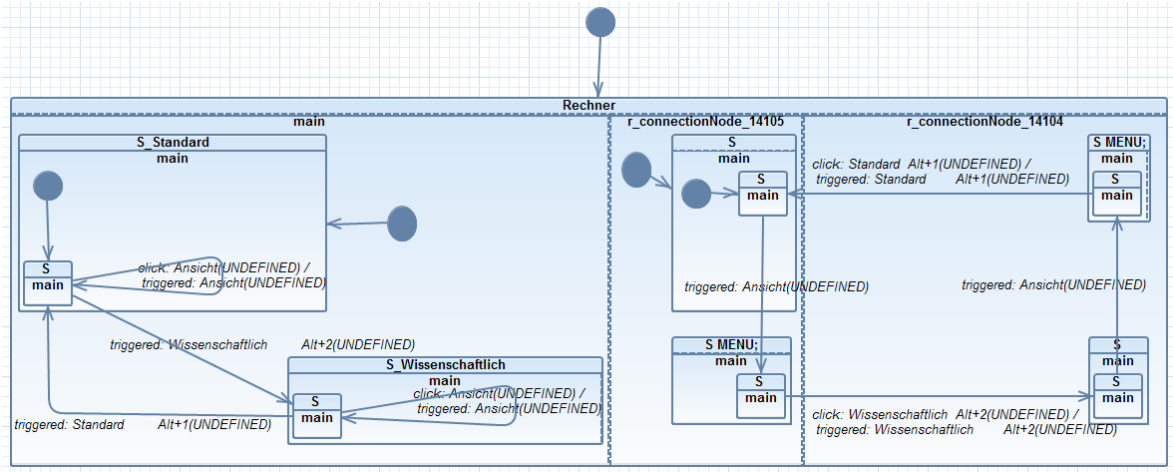


Abbildung 118: Durch einen Fehler in der Methode erhält das Ansichts-Menü des Taschenrechners einen orthogonalen Bereich je Modus in dem es aufgenommen wurde. Für die Übersichtlichkeit wurden einige der generierten Zustände im Bild umbenannt.

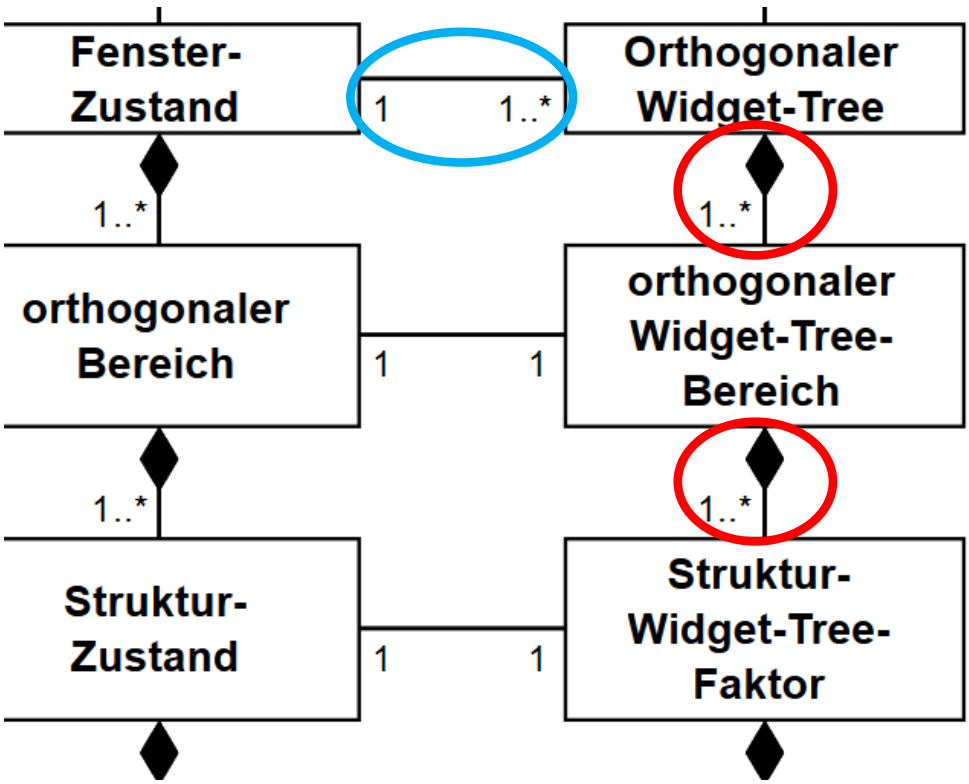


Abbildung 119: Eine der rot markierten Beziehungen im abstrahierten GUI-Modell könnte problematisch sein, da Widget-Tree-Faktoren vielleicht mehreren orthogonalen Widget-Trees zugeordnet werden müssen, um das Problem der verdoppelten Regionen zu vermeiden. Im Vergleich zu allen anderen Elementen des Zustandsautomaten haben Fenster-Zustände keine eindeutige Entsprechung im Widget-Tree (blauer Kreis). Dies ist eine Inkonsistenz im Design des Modells. Das Problem der verdoppelten Regionen könnte sich durch beheben dieser Design-Schwäche auflösen.

6.4.1.2 Tatsächlich evaluiertes Teilmodell

Im vorherigen Abschnitt wurde erklärt, dass für gleiche Menüs mehrere orthogonale Bereiche entstehen können. Deshalb wurde eine Untermenge der Testschritte des Tutorials gewählt – und damit eine Untermenge des Erwartungsmodells – für die dieses Problem nicht auftritt.

Anstatt gegen das vollständige Erwartungsmodell (Abbildung 108) zu evaluieren, wird gegen ein reduziertes Erwartungsmodell (siehe Abbildung 120, oben) evaluiert, in welchem keine Regionen verdoppelt werden. Das generierte und evaluierte Realmodell ist in Abbildung 120 unten dargestellt. Die Zustände wurden so angeordnet, dass die Modelle gut vergleichbar sind.

Die Gemeinsamkeiten und Unterschiede der Modelle und deren Ursachen werden in den nachfolgenden Abschnitten genauer untersucht.

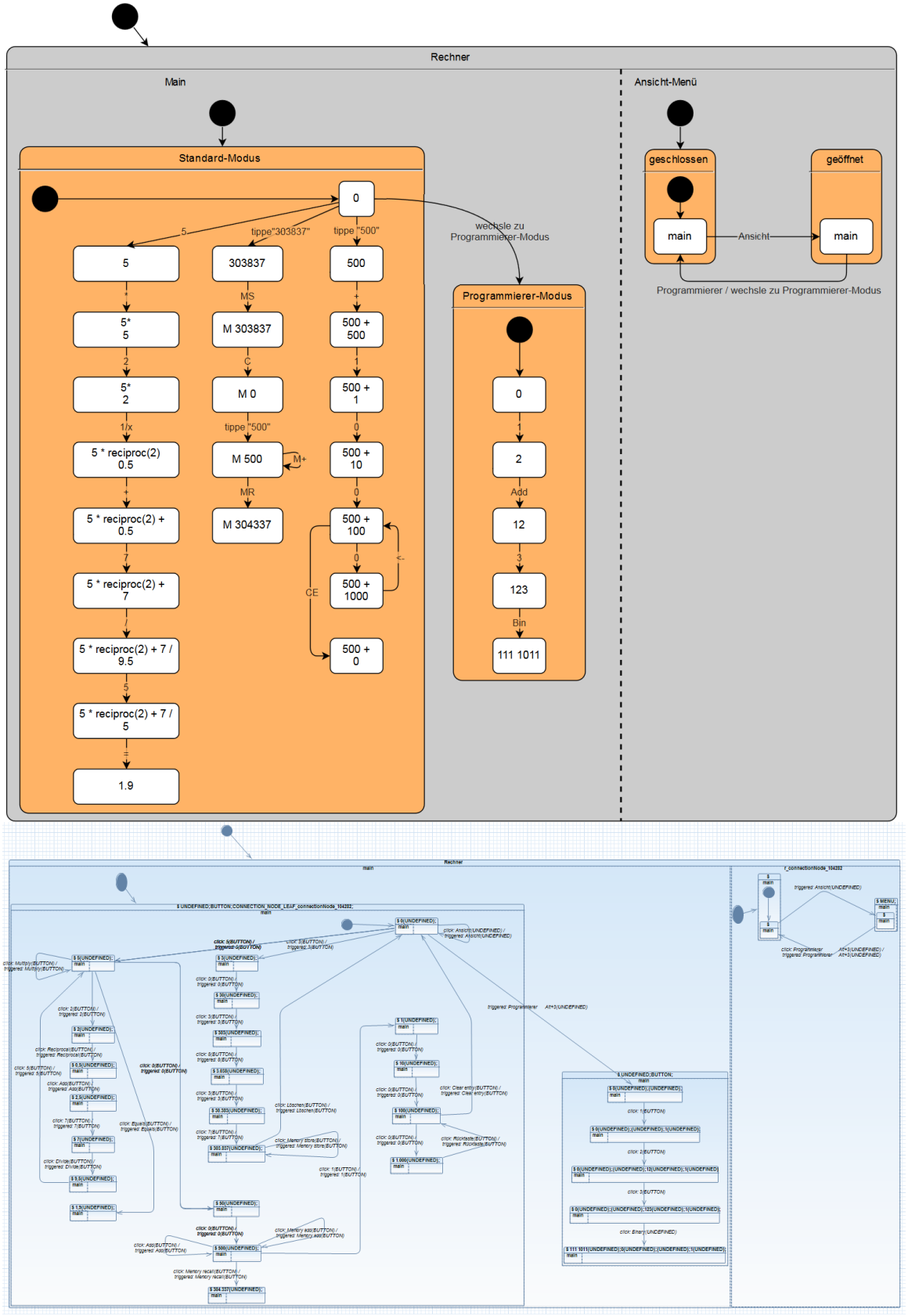


Abbildung 120: Oben: Untermenge des Erwartungsmodells, für welches keine Regionen verdoppelt werden. Unten: Realmodell der evaluierten Eingabesequenzen. Hier wurde auf die gewohnte Farbwahl verzichtet. Für eine größere Darstellung siehe Anhänge B und C.

6.4.2 Quantitative Auswertung

Die in Abschnitt 6.4.1.2 vorgestellten Real- und Erwartungsmodelle wurden mittels einiger Vergleichs-Metriken gegeneinander evaluiert. Einen Überblick über die gemessenen Gemeinsamkeiten und Unterschiede gibt Tabelle 6. Die Bedeutung der einzelnen Werte wird nachfolgend in diesem Abschnitt erläutert. Auf die Ursachen für die gemessenen Unterschiede wird im nächsten Abschnitt eingegangen.

Metrik	**Soll-Wert**	**Ist-Wert**	**Relative Abweichung / Anzahl**	**Anmerkungen**
Anzahl Zustände:	34	33	-2,9%	ohne Pseudo-Zustände
Fenster-Zustände	1	1	0	
Strukturzustände	4	4	0	
Wertezustände	29	28	-3,4%	
Unveränderte Zustände	-	27	79,4%	
Hinzugekommene Zustände	-	6	+17,6%	
Weggelassene Zustände	-	7	-20,5%	
Anzahl Transitionen	30	36	+20%	Transitionen von Pseudo-Zuständen (z.B. Startzustand) werden nicht mitgezählt.
Anzahl verwerteter automatischer Aktionen	1	2	+100%	
Anzahl orthogonaler Regionen	2	2	0	

Tabelle 6: Vergleichs-Metriken zwischen Real- und Erwartungsmodell

Zählt man nur die Gesamtzahl der Zustände beider Modelle aus, so scheint im Realmodell lediglich ein Zustand zu fehlen, was einer Abweichung von 2,9 Prozent entspricht. Wendet man etwas Kontextwissen an, um die Zustände mit einer Bedeutung zu versehen, dann kann man im Real- und Erwartungsmodell die Zustände mit gleicher Bedeutung einander zuordnen. Diese Zuordnung zeigt, dass 27 Zustände (79,4%) aus dem Erwartungsmodell mit gleicher Bedeutung im Realmodell auftauchen. Im Realmodell gibt es sechs Zustände (17,6%) mit einer Bedeutung, welche im Erwartungsmodell nicht vertreten waren. Umgekehrt kommen sieben Zustände (20,5%) aus dem Erwartungsmodell nicht mit gleicher Bedeutung im Realmodell vor. Diese zwei Unterschiede neutralisieren sich gegenseitig, wodurch der Gesamtunterschied – welcher ohne Kontextwissen ermittelt wurde – besser aussieht, als er ist. Die Unterschiede treten ausschließlich in den Wertzuständen auf. Fenster- und Struktur-Zustände im Real- und Erwartungsmodell sind gleich.

Das Realmodell enthält 36 Transitionen, also 20% mehr als das Erwartungsmodell. Da die bereits die Zustände teils unterschiedliche Bedeutungen haben, ist nur schwer zu ermitteln, welche Transitionen eine gleiche oder unterschiedliche Bedeutung haben.

Die Anzahl der automatisch ausgelösten Aktionen, welche auch tatsächlich als Event empfangen werden, ist doppelt so hoch wie erwartet.

Die Anzahl der orthogonalen Regionen ist so hoch wie erwartet. Allerdings wurden die Evaluierungs-Modelle auch bewusst so reduziert, dass hier ein Fehler umgangen wurde (vgl. Abschnitt 6.4.1).

6.4.3 Qualitative Auswertung

In den folgenden Unterabschnitten werden die Details und Ursachen der Unterschiede zwischen Real- und Erwartungsmodell beschrieben. Die Ursache für die Gemeinsamkeiten ist einfach eine korrekte Implementierung und die Details der Gemeinsamkeiten sind alles, bis auf die beschriebenen Unterschiede. Daher wird auf die Gemeinsamkeiten im Folgenden nicht weiter eingegangen.

6.4.3.1 Benennung

Im Realmodell sind die Zustände und Transitionen oft anders benannt als im Erwartungsmodell (vgl. Abbildung 121). Nach welchen Regeln die Benennung in den verschiedenen Modellen erfolgt, wurde bisher nicht beschrieben und soll auch hier nur oberflächlich erwähnt werden. Grob gesagt wurden im Erwartungsmodell Kontextwissen und optimistische Annahmen über die Widget-Trees genutzt, um möglichst nachvollziehbare Benennungen vorzunehmen (vgl. Beispiel Abbildung 121 oben). Im Realmodell wurden dagegen für Transitionen die realen Werte aus den Widget-Trees benutzt. So heißt zum Beispiel der Menüeintrag, welcher mit „Programmierer“ beschriftet ist, real „Programmierer Alt+3“ und ist von einem bisher nicht implementieren Widget-Typ. Für Zustände werden die Bezeichnungen der Widgets aufgelistet, welche nicht in den Geschwister-Zuständen vorkommen. Zum Beispiel unterscheidet sich der Zustand in Abbildung 121 unten dadurch von seinen Geschwister-Zuständen, dass er ein unbenanntes Widget eines nicht implementierten Typs und einen unbenannten Button hat. Die vom FRUIT-Framework bereitgestellten Widgets (vgl. Abschnitt 5.3.3.1) sind oft unbenannt, was eine Benennung zusätzlich erschwert.

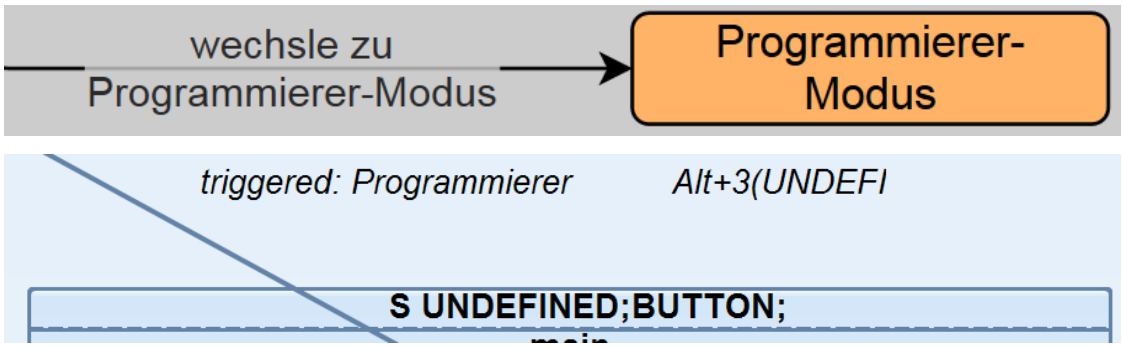


Abbildung 121: Zustände und Transitionen wurden im Erwartungsmodell (oben) mittels Kontextwissen und optimistischen Annahmen über die Widget-Trees sinnvoll benannt. Im Realmodell (unten) heißen die Elemente daher oft anders.

Da die Benennung nicht im Mittelpunkt dieser Arbeit stand und bereits in dieser primitiven Form für eine akzeptable Benutzbarkeit sorgte, wurde sie weder in der Methode beschrieben, noch weiter verbessert. Die Bedeutung der Zustände ist trotz der anderen Benennung deutlich geworden, so dass alle Zustände evaluiert werden konnten.

6.4.3.2 Fehlende Widgets

In Abschnitt 5.3.3.1 wurde bereits kurz erwähnt, dass in den vom FRUIT-Framework bereitgestellten Widget-Trees einige Widgets fehlen.

Zwei Beispiele, welche Widgets nicht im Widget-Tree vertreten sind, sind in Abbildung 122 beschrieben. Diese Widgets sind zwar deutlich sichtbar, werden jedoch nicht als technisch eigenständige Einheiten erkannt, wodurch die Widget-Trees von zwei verschieden aussehenden GUI-Zuständen gleich sind.

Beim Erstellen des Erwartungsmodells wurden optimistisch vollständige Widget-Trees angenommen, wodurch den zwei Ansichten aus Beispiel 1 (Abbildung 122) auch zwei verschiedene Widget-Trees und somit zwei verschiedene Zustände zugeordnet wurden (Abbildung 123, links). Im Realmodell fallen – durch gleiche Widget-Trees – diese Zustände oft zu einem einzigen zusammen. Auch Transitionen verlaufen hierdurch anders.

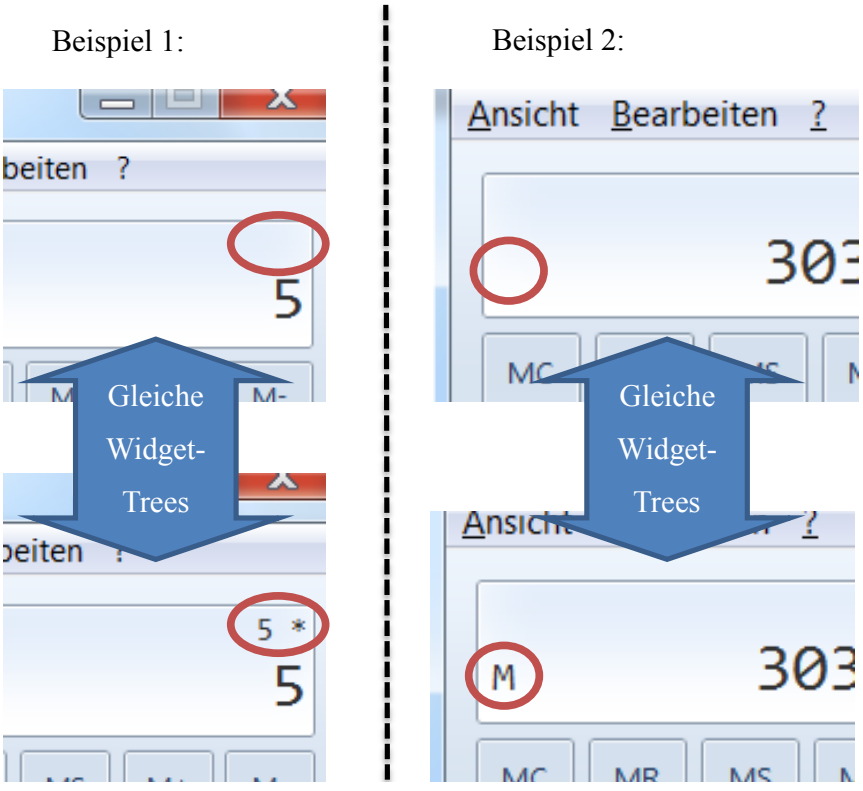


Abbildung 122: Das FRUIT-Framework liefert teils unvollständige Widget-Trees. Im linken Beispiel fehlt im Baum das Widget mit den zuletzt getätigten Eingaben. Im rechten Beispiel fehlt das Widget mit dem „M“, welches anzeigt, dass etwas im internen Speicher des Taschenrechners ist.

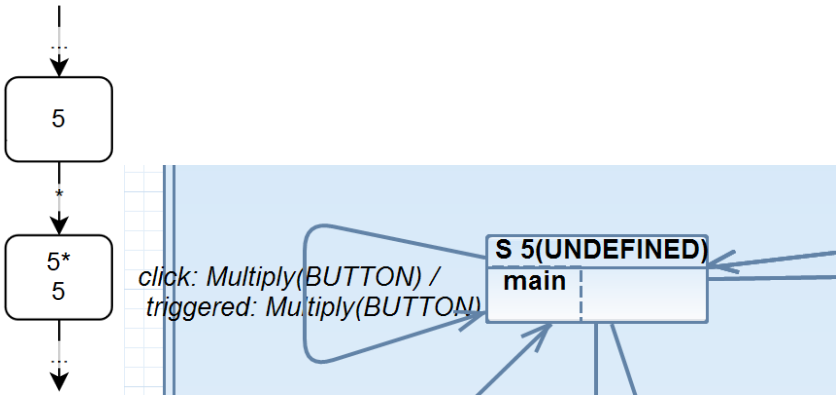


Abbildung 123: Im Erwartungsmodell (links) sind die Zustände korrekt aufgeteilt, weil ein vollständiger Widget-Tree angenommen wurde. Im Realmodell (rechts) werden unterschiedliche Ansichten der GUI teilweise als gleicher Zustand erkannt. Dadurch entfallen Zustände (links unten) und Transitionen verlaufen anders (Im Beispiel wird Multiplikation wird zu einer Selbsttransition).

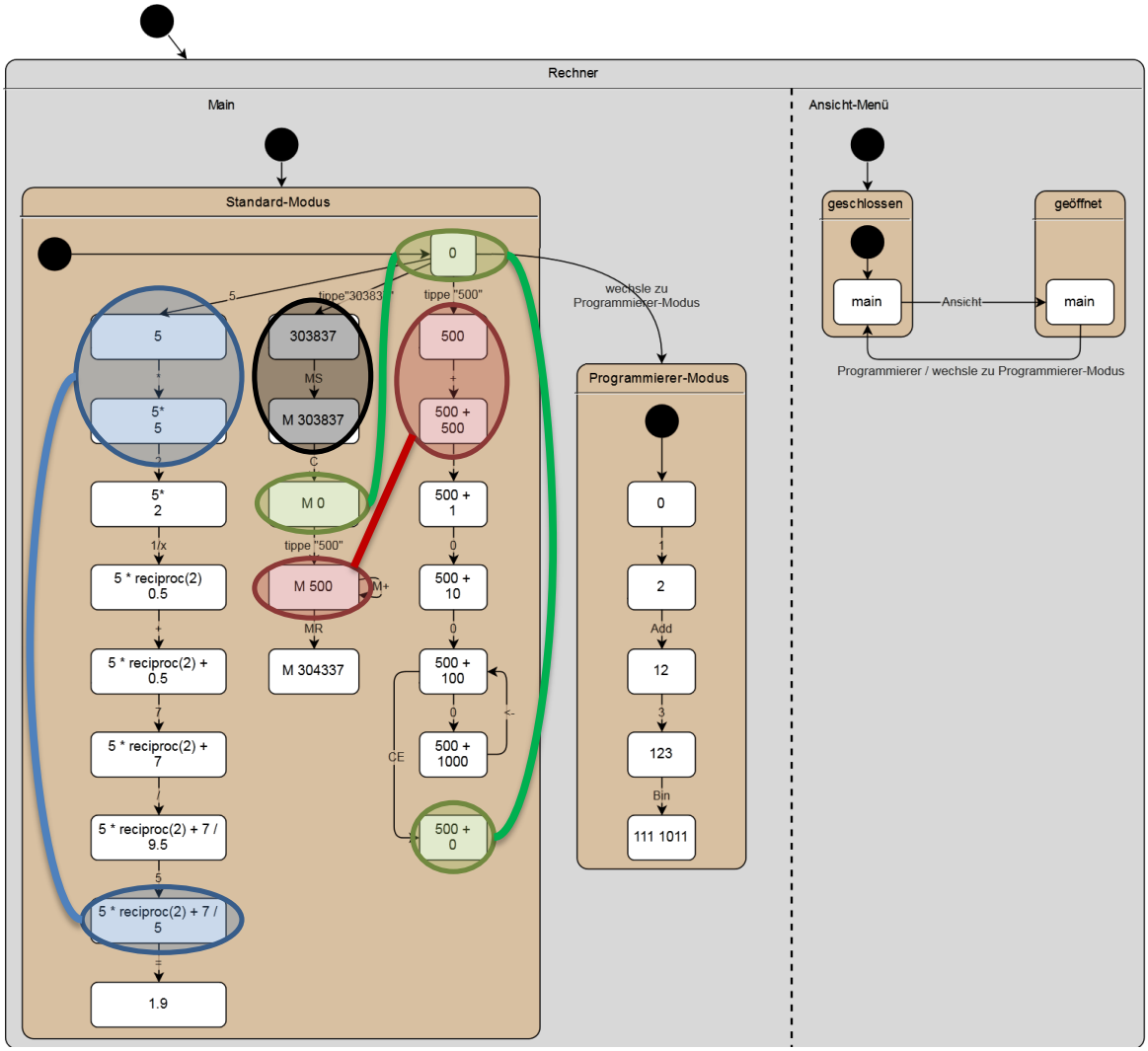


Abbildung 124: Durch die fehlenden Widgets werden im Realmodell mehrere Zustände des Erwartungsmodells vereinigt. Im Bild wurden jeweils die Zustände in einer Farbe umrahmt, welche durch fehlende Widgets vereinigt werden.

Abbildung 124 zeigt, welche Zustände durch die fehlenden Widgets vereinigt werden. Durch diese Vereinigung entfallen genau die sieben Zustände, welche in der quantitativen Auswertung (Abschnitt 6.4.2) gezählt wurden. Die fehlenden Widgets sind damit die einzige Ursache für das Wegfallen von Zuständen.

6.4.3.3 Keine Tastatur-Implementation

Im implementierten Prototyp werden vorerst noch keine Tastatureingaben registriert, sondern nur Mauseingaben. Wenn also im Tutorial beschrieben war, das man etwas tippen soll, wurden stattdessen mit der Maus die Buttons bedient, welche den äquivalenten Effekt haben.

Stand beispielweise im Tutorial „tippe 303837“, so wurde fürs Realmodell stattdessen nacheinander mit der Maus auf die Buttons „3“, „0“, „3“, „8“, „3“ und „7“ geklickt. Durch die Subaktivität „Eingaben zusammenfassen“ (Abschnitt 4.3.8) würden die sechs Tastaturanschläge zu einer einzigen Transition zusammengefasst werden (Abbildung 125, links), wie es auch im Erwartungsmodell geschehen ist. Die Mauseingaben hingegen werden zu einer Transition pro Klick zusammengefasst und dazwischen entsprechende Zustände aufgenommen (Abbildung 125, rechts). So entsteht pro Tastenanschlag zusätzlich zum ersten eine zusätzliche Transition und ein zusätzlicher Zustand im Realmodell.

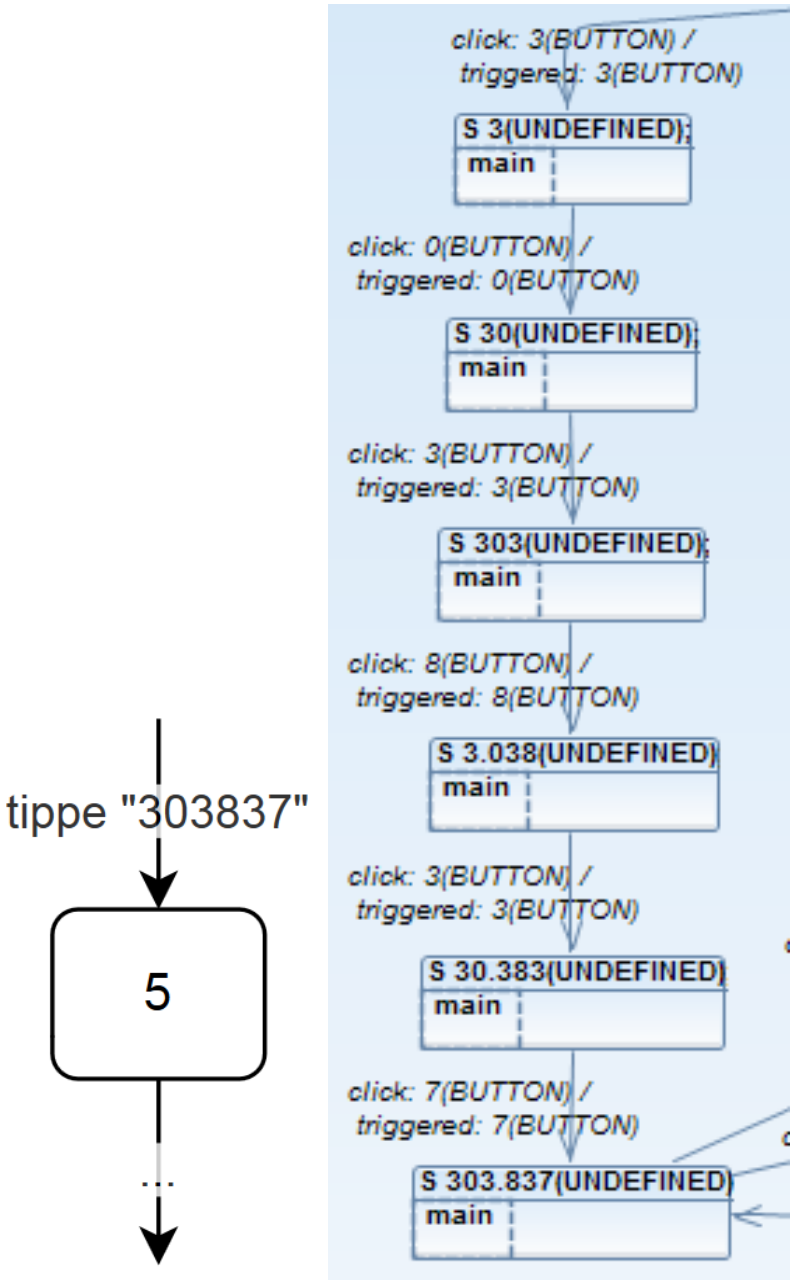


Abbildung 125: Im Erwartungsmodell (links) werden Tastatureingaben aufgenommen und korrekt zusammengefasst. Zur Aufnahme des Realmodells (rechts) wurden dagegen Tastatureingaben durch Mauseingaben ersetzt, welche weniger kompakt zusammengefasst werden.

Abbildung 126 zeigt alle Transitionen des Erwartungsmodells, die Tastatureingaben beinhalten. Die Aufspaltung dieser Transitionen ist die alleinige Ursache für die sechs zum Erwartungsmodell hinzugekommenen Zustände (vgl. Quantitative Auswertung, Abschnitt 6.4.2). Auch fünf der sechs zusätzlichen Transitionen haben ihre Ursache im Fehlen der Tastatureingaben-Implementation.

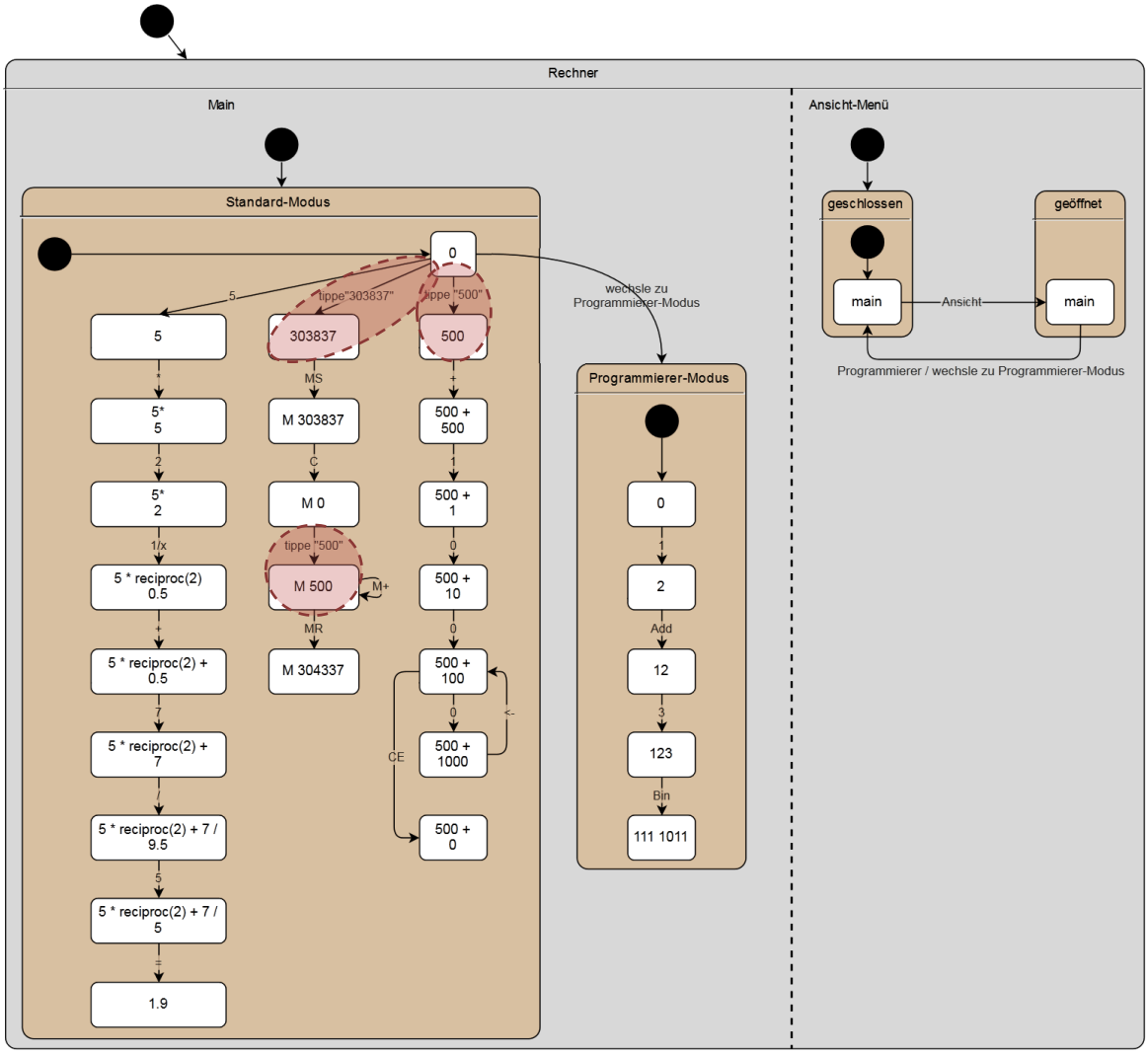


Abbildung 126: Die Transitionen im Erwartungsmodell welche Tastatureingaben erfordern und deshalb im Realmodell aufgeteilt wurden sind im Bild umkreist.

6.4.3.4 Auslöser für Kontextmenü

Das Erwartungsmodell wurde so modelliert, das man direkt innerhalb eines orthogonalen Bereichs eines Menüs dieses Menü öffnen kann. Diese Modellierung war zwar näher am Idealmodell als das Realmodell, jedoch folgte sie nicht stoisch genug der entwickelten Methode. So wurde Subaktivität „Sequenzen einbeziehen“ (Abschnitt 4.4.4) beim manuellen Aufstellen des Erwartungsmodells nicht konsequent genug durchgeführt, bzw. nicht gründlich genug über die verfügbaren Widget-Trees nachgedacht. Da das Erwartungsmodell aufgestellt wurde, indem ein Mensch einen komplexen Algorithmus manuell angewandt hat, ist ein solcher Fehler nicht ungewöhnlich (vgl. [Nati14]). Der Prototyp wurde implementiert, um derartige Fehler in der theoretischen Methode zu finden.

Im Beispiel (Abbildung 127) sieht man, wie das Ansicht-Menü geöffnet wird. Im Erwartungsmodell (1) verläuft eine Transition direkt vom „Ansicht-Menü geschlossen“-Zustand in den „Ansicht-Menü geöffnet“-Zustand. Im Realmodell dagegen ist im Haupt-Bereich eine Selbsttransition, welche die Ansicht-Aktion auslöst (2). Diese Aktion wird als Ereignis von der Transition im orthogonalen Bereich des Ansichts-Menüs wahrgenommen und löst so den Zustandsübergang im orthogonalen Bereich aus (3).

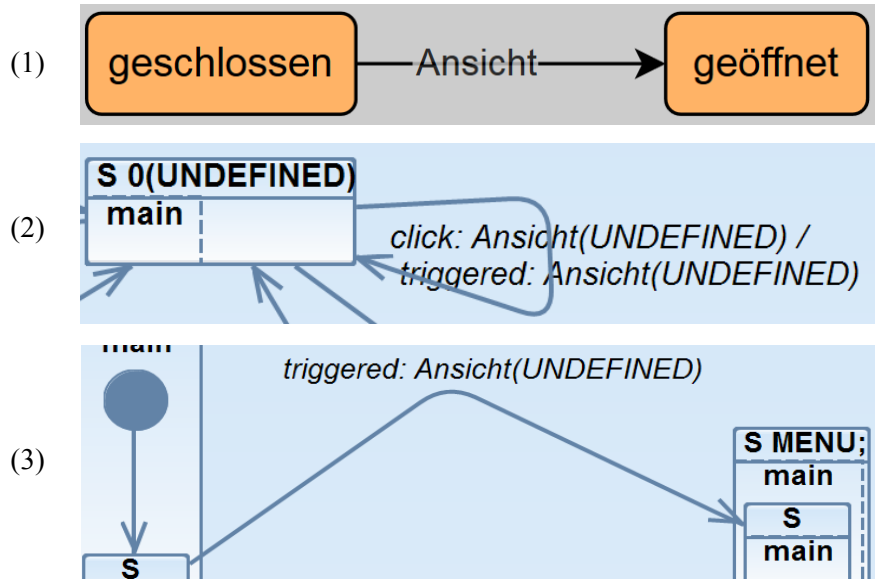


Abbildung 127: Im Erwartungsmodell (1) werden Menüs direkt durch Transitionen im orthogonalen Bereich geöffnet. Im Realmodell dagegen wird durch eine Selbsttransition im Haupt-Bereich eine Aktion ausgelöst (2), welche indirekt den Zustandswechsel im orthogonalen Bereich auslöst (3).

Dieser indirekte Zustandsübergang im Realmodell sorgt für insgesamt eine (von sechs) zusätzliche Transition und für eine zusätzliche Aktion im Vergleich zum Erwartungsmodell.

6.4.3.5 Klassifikationsbaum und Widget-Trees

Die Elemente des Klassifikationsbaums entsprechen vollständig den Elementen des bereits evaluierten orthogonalen, hierarchischen Zustandsautomaten. Die Gemeinsamkeiten und Unterschiede sind Äquivalent zu denen des Zustandsautomaten, weswegen der Baum hier nicht noch einmal separat evaluiert wird.

Die integrierte vorhandene Implementation der Sequenzgenerierung (vgl. Abschnitt 4.5) funktioniert nur teilweise. So werden die automatischen Aktionen noch nicht in der Sequenzgenerierung ausgelöst und ausgewertet. Die restliche Generierung funktioniert jedoch wie erwartet. Aufgrund der vereinigten Zustände (vgl. Abschnitt 6.4.3.2) kommt es dennoch dazu, dass unausführbare Sequenzen generiert werden.

Die Widget-Trees, welche das FRUIT-Framework liefert, sind oft Lückenhaft und anders als erwartet. Dennoch geht das Zusammenspiel zwischen Widget-Tree, Zustandsautomat und Klassifikationsbaum gut auf. Auch die im Rahmen dieser Arbeit definierten orthogonalen Widget-Trees erfüllen ihren Zweck gut. Lediglich das Zusammenfassen der Faktoren unter einem orthogonalen Widget-Tree (Abschnitt 4.2.3.1) bereitet kleinere Probleme (Abschnitt 6.4.1.1).

6.5 Rückblick: Problemstellung

In der Problemstellung wurden Eingangs einige Fragen aufgeworfen, welche hier jetzt nochmals zusammenfassend beantwortet werden sollen. Details zu den Antworten befinden sich in den jeweiligen Abschnitten. Die Fragen sind im Folgenden kursiv formatiert, die Antworten folgen direkt dahinter.

Kann die Klassifikationsbaum-Methode auf GUI-Tests angewendet werden? Welchen Beitrag leistet die Klassifikationsbaum-Methode beim Testentwurfsprozess?

Ja, die Klassifikationsbaum-Methode kann auf GUI-Tests angewandt werden. Ihren Hauptbeitrag leistet die Klassifikationsbaum-Methode beim Generieren der Testsequenzen aus einem Modell, nachdem dieses bereits erstellt wurde. Mittels der Klassifikationsbaum-Methode können kombinatorische Überdeckungskriterien, wie zum Beispiel paarweise Zustandsüberdeckung generiert werden.

Können die Klassifikationsbaum-Methode, das Widget-Tree-Modell, der Zustandsautomat und die Capture/Replay-Methode zu einer systematischen und komfortablen Methode für einen allgemeinen GUI-Testprozess kombiniert werden?

Ja, die vier Methoden wurden im Rahmen dieser Arbeit erfolgreich zu einer systematischen, komfortablen Methode kombiniert.

Welche Vor- und Nachteile trägt jedes Teilstück zum kombinierten Verfahren bei? Wie können die Nachteile gehandhabt werden?

In Abschnitt 2.5 wurde bereits ein Überblick über die Vor- und Nachteile der vier einzelnen Methoden gegeben. Insgesamt vereint das kombinierte Verfahren alle Vorteile und umgeht alle Nachteile, mit folgenden Ausnahmen:

- Eine höhere Stabilität des Verfahrens gegenüber Änderungen der SUT als beim einfachen Capture/Replay ist anzunehmen, da keine Stabilität der Testsequenzen mehr nötig ist, sondern ein Modell der GUI vorliegt, aus welchem man bei Änderungen einfach neue Sequenzen generieren kann. Auch die verwendete Strategie zum Wiederfinden von Widgets (vgl. Abschnitt 4.6.2) beeinflusst die Stabilität. Wie stabil das Verfahren wirklich gegenüber Änderungen des SUTs ist, wurde von vorn herein aus der Untersuchung ausgeschlossen (vgl. Abschnitt 1.5.2) und kann daher nicht mit Sicherheit beantwortet werden.
- Die Größe der Klassifikationsbäume wurde durch das Verfahren ein wenig verbessert. Durch das Aufspannen orthogonaler Bereiche wurde die Anzahl der Klassen reduziert.[51] Für komplexe GUIs oder Testreihen kann der Baum immer noch unübersichtlich groß werden und benötigt zusätzliche Strategien und Testplanung.

Welche Synergieeffekte treten durch die Kombination der Methoden auf? Welche Vorteile der Einzelmethoden werden durch die Kombination mit anderen Methoden unterdrückt?

Der wichtigste Synergieeffekt ist die Vereinigung aller Vorteile und das gleichzeitige Vermeiden aller Nachteile der Einzelmethoden (siehe vorherige Frage).

In der hier vorgestellten Methode spielt der Tester fast keine Rolle. Die Modelle werden automatisch konstruiert und es gibt kaum Möglichkeiten diese nachträglich manuell anzupassen.

[51] gegenüber einem äquivalenten nichtorthogonalen Modell

Durch eine feste Vorgabe der Modellstruktur gehen die Freiheiten beim Modellieren des SUTs verloren. Der dadurch erkaufte Vorteil ist die automatische Konstruktion des Modells.

Die direkte Verbindung zwischen aufgenommenen Eingabesequenzen (Capture) und abgespielten Testsequenzen (Replay) wird aufgelöst. Stattdessen wird mittels Capture ein Modell konstruiert und gegebenenfalls abweichende Testsequenzen daraus hergeleitet. Der Vorteil ist eine höhere Stabilität gegenüber Änderungen des SUTs, für welche intuitive Erlernbarkeit eingebüßt werden musste. Für zukünftige Arbeiten wäre es denkbar, die aufgenommenen Sequenzen als Testsequenzen auf dem Modell darzustellen, um diesen Nachteil wieder zu eliminieren.

Welche Informationen einer konkreten GUI-Technologie gehen verloren, wenn zu einem technologieunabhängigen GUI-Modell abstrahiert wird und wie wirkt sich dieser Informationsverlust auf den Testprozess aus?

Im Rahmen dieser Arbeit konnte kein derartiger Verlust beobachtet werden. Ganz im Gegenteil erwartet das Modell sogar mehr Informationen, als die konkreten Technologien tatsächlich liefern konnten (vgl. Abschnitt 6.4.3.2).

Das technologieunabhängige *GUI-Modell* wurde noch weiter zu einem *abstrahierten GUI-Modell* verallgemeinert. Bei diesem Abstraktionsprozess wird bewusst akzeptiert einige Informationen zu verlieren, wodurch die Wartbarkeit des Modells erhöht wird und zusätzliche – wahrscheinlich zutreffende – Eigenschaften des GUI-Systems modelliert werden, welche zuvor nicht mittels Capture aufgenommen wurden.

Wie wird der GUI-Testprozess durch die entwickelte Methode eingeschränkt?

Dies wurde bereits unter einer vorherigen Frage beantwortet. Kurz gesagt: Der Tester hat weniger Einfluss und das Modell schränkt die freie Modellierung ein.

Wie kann die erarbeitete Methode als Werkzeug implementiert werden? Welche existierenden Werkzeuge, Frameworks und Bibliotheken werden verwendet und wie werden diese angepasst?

Die Methode wurde im Rahmen dieser Arbeit prototypisch implementiert. Dazu wurde eine Vielzahl verschiedener Werkzeuge, Frameworks und Bibliotheken integriert. Eine detailliertere Beschreibung befindet sich in Abschnitt 0. Die technischen Unsicherheiten einer Implementierung wurden durch den Prototyp beseitigt. Eine vollständige Implementierung der Methode scheint ohne technische Risiken machbar zu sein.

7. Fazit und Ausblick

Die folgenden Unterabschnitte bieten ein Fazit dieser Arbeit, Ideen und Lösungsansätze für zukünftige Arbeiten.

7.1 Fazit

Durch die steigende Komplexität und Verbreitung von GUI-Systemen ist automatisiertes GUI-Testen zunehmend wichtig geworden. Obwohl es in der Wissenschaft hoch gelobt wird, ist modellbasiertes GUI-Testen in der Wirtschaft kaum verbreitet.

In dieser Arbeit wurde eine Methode vorgestellt, welche helfen soll diese Kluft zu schließen, indem verbreitete Ansätze, wie Capture/Replay, die Klassifikationsbaum-Methode und Zustandsautomaten zu einer neuen, modellbasierten GUI-Testmethode zusammengeführt werden. Die Methode leitet mittels Capture halbautomatisch ein Modell der GUI her. Dieses Modell wird genutzt, um automatisch ausführbare Testsequenzen zu generieren. Ein Werkzeug zur Durchführung der Methode wurde prototypisch implementiert und die Machbarkeit einer vollständigen Implementierung evaluiert.

Die GUI wird mittels eines Modells aus einem orthogonalen, hierarchischen Zustandsautomaten, einem Klassifikationsbaum und mehreren, in dieser Arbeit entwickelten, orthogonalen Widget-Trees beschrieben.

Es wurde sowohl die Methode selbst, als auch das implementierte Werkzeug evaluiert. Das durch die Methode erzeugte Modell stimmte dabei weitestgehend mit einem Idealmodell überein. Interne, nicht sichtbare Zustände und Kontextwissen werden durch die Methode jedoch nicht ermittelt und modelliert. Der Prototyp zeigte, dass die Methode noch um einen kleinen Teilschritt ergänzt werden muss. Der Prototyp erzielt bereits vielversprechende erste Ergebnisse, weist jedoch noch zwei größere Probleme auf. Zum einen erkennt eine verwendete Bibliothek nicht alle Widgets, so wie der Mensch sie wahrnimmt. Zum anderen ist der Prototyp noch nicht vollständig implementiert und unterstützt beispielsweise noch kein Capture von Tastatureingaben.

Trotz der herausgearbeiteten Probleme scheint die Kombination der vier Methoden vielversprechend und für das Testen von GUI-Systemen geeignet zu sein. Schwächen einzelner Methoden konnten weitestgehend überwunden werden, ohne viele Einbußen bei ihren Stärken hinzunehmen. Viele der festgestellten Probleme scheinen durch zukünftige Arbeiten überwindbar.

7.2 Vertiefende Themen

Für die entwickelte Methode und den Prototyp wurden noch einige Probleme identifiziert. Dieser Abschnitt soll Lösungsansätze für einige dieser Probleme aufzeigen.

Ein Problem war, dass orthogonale Regionen unter Umständen mehrfach im Modell vorkommen. Ansätze um dieses Problem zu handhaben wurden in Abschnitt 6.4.1.1 beschrieben.

Abschnitt 4.3.8 beschreibt eher oberflächlich und informell, wie mehrere Nutzereingaben zu Aktionen zusammengefasst werden. Hier wären eine formellere Definition und ein umfassenderes Regelwerk wünschenswert. Formale Sprachen und reguläre Ausdrücke könnten zum spezifizieren des Verhaltens und der Regeln verwendet werden.

Die durch fehlende Widgets vereinten Zustände (vgl. Abschnitt 6.4.3.2) könnten auf mehrere Arten behoben werden. Zum einen könnte ein Austauschen der GUI-scannenden Bibliothek dazu führen das alle Widgets erkannt werden. Eine bildbasierte Erkennung der Widgets könnte helfen. Auch eine Integration mehrerer Bibliotheken gleichzeitig ist denkbar. Zum anderen könnte der Tester die Modelle manuell nachbearbeiten und somit die technisch schwer feststellbaren Fehler wieder beseitigen. Die Methode müsste um eine Bearbeitung der Modelle erweitert werden.

Die fehlende Implementierung der Tastatur (vgl. Abschnitt 6.4.3.3) ist machbar. Jedoch sei erwähnt, dass die Entwicklung und speziell das Debuggen der Implementation zusätzliche Probleme bereiten kann. So löst jeder Tastenanschlag, egal ob im SUT oder im Debugger einen Prozess aus und ruft somit unerwünschtes Verhalten hervor. Eine Funktio-

nalität zum Pausieren und Fortsetzen des Capture-Prozesses ist nötig, welche auch aus dem Debugger heraus aktiviert werden kann.

In Abschnitt 5.3.2 wurde beschrieben, dass oft mehrere Implementationen für ein einziges Modell des Konzeptes existieren. In zukünftigen Arbeiten sollte versucht werden, diese Modelle so weit wie möglich zu fusionieren, um den Wartungsaufwand zu reduzieren. Viele der Modelle sind jedoch durch die Verwendung verschiedener Bibliotheken unvermeidlich.

Die fehlende Modellierung von Kontextwissen und internen Zuständen ist problematisch (vgl. Abschnitt 6.3). Entweder müsste man dem Tester die Möglichkeit geben, diese Modelle manuell nachzubearbeiten, oder man müsste durch größere Datenmengen automatisierbare Schlussfolgerungen ziehen. Für den zweiten Ansatz sei auf Givens et al. [GCSY13] verwiesen.

7.3 Weiterführende Themen

Dieser Abschnitt beschreibt einige Ansätze, wie man die Methode durch zusätzliche Funktionalität weiter verbessern könnte.

In Abschnitt 1.5 distanziert sich diese Arbeit sowohl von Regressionstests als auch von Testorakeln. Da der Hauptvorteil von automatisch ausführbaren Tests die Ausführung von Regressionstests ist, sollte genauer evaluiert werden wie stabil die Modelle und Testsequenzen gegenüber Änderungen des SUTs sind. Auch eine automatische Testauswertung und damit ein Testorakel sind dazu nötig. Ein einfacher, erster Ansatz wäre hier, zu prüfen, ob die bei der Ausführung tatsächlich durchlaufenen Zustände denen aus der Testsequenz-Spezifikation entsprechen. Xie stellt einen Test Orakel Generator vor, welcher halbautomatisch erwartete Ergebnis-Zustände des GUI herleitet [Xie06].

Auch Testgetriebene Entwicklung wurde in dieser Arbeit nicht behandelt (vgl. Abschnitt 1.5.1), da der aktuelle Ansatz das Modell aus einer bestehenden GUI generiert. Hier ist eine Generierung aus einem Mock-up denkbar.

Zur weiteren Reduzierung der Modellgröße könnten Variablen und Wächterausdrücke an den Transitionen des Zustandsautomaten eingeführt werden. Einige einfachere Ansätze diese Variablen und Wächterausdrücke automatisiert zu modellieren wurden im Zusammenhang mit dieser Arbeit ausprobiert (jedoch nicht beschrieben). Keiner der Ansätze resultierte in vielversprechenden Ergebnissen. Die zu ermittelnden Modellierungsvorschriften könnten also recht komplex oder schwer zu ermitteln sein. Shehady und Siewiorek stellen ein GUI-Modell vor, welches solche *Variable Finite State Machines (VFSM)* nutzt [ShSi97]. Samek gibt Richtlinien, welche Systemeigenschaften als Zustände und welche als Variablen modelliert werden sollten [Same08].

Ähnlich zu den Variablen könnte man auch versuchen, Zustände mit Historie zu modellieren. Dabei treten vermutlich ähnliche Schwierigkeiten auf diese automatisiert zu ermitteln.

Nguyen et al. [NgMT12] kombinieren Zustandsautomaten mit der Klassifikationsbaum-Methode anders als in dieser Arbeit beschrieben (vgl. Abschnitt 1.7). Dieses Verfahren ließe sich vielleicht mit dem hier entwickelten Verfahren kombinieren, indem die Klassifikationsbäume von Nguyen als Unterbäume der Transitions-Klassen dieser Arbeit verwendet werden. So könnten Pfade durch den Zustandsautomaten zusätzlich parametrisiert werden.

Ricca und Tonella [RiTo01] reduzieren ihre GUI-Modelle, indem alternativlose Sequenzen zusammengefasst werden. Dies kann man auch auf Zustandsautomaten anwenden. Hat ein Zustand *B* genau einen Vorgänger *A* und genau einen Nachfolger *C*, so kann man *A* und *C* symbolisch direkt verbinden und *B* ausblenden. So kann man lineare Ketten von Zuständen vereinfachen. Ähnlich könnte man auch versuchen Transitionen durch die Verwendung von Kreuzungs- und Entscheidungs-Pseudozuständen zusammenzufassen.

Memon [53] stellt eine modellbasierte Methode vor, welche es dem Tester komfortabler gestalten soll Tests zu spezifizieren. Der Tester spezifiziert dabei auf einem (dem Zustandsautomaten ähnlichen) Modell nur noch Start- und Zielzustände und ein Planungs-Algorithmus sucht selbstständig verschiedene Pfade vom Start- zum Zielzustand. Die so entstehenden Pfade können als Testsequenzen verwendet werden. Diese Methode hat

weiterhin den Vorteil, dass sie relativ stabil gegenüber Änderungen des SUTs und damit geeignet für Regressionstests ist. Sofern bei Änderungen des SUTs Start- und Zielzustand weiterhin existieren, müssen lediglich neue Pfade dazwischen generiert werden. Memons Methode könnte man in ähnlicher Weise auf die Methode dieser Arbeit anwenden, um funktionale Tests leicht zu spezifizieren oder um Vor- oder Nachbedingungen leichte herzustellen.

Anhang A: Idealmodell Taschenrechner

Im Folgenden wird das Idealmodell des Windows-Taschenrechners im Detail gezeigt. Der Übersichtlichkeit halber werden jeweils Ausschnitte der Modelle auf mehrere Diagramme verteilt dargestellt. Dabei wird top-down vorgegangen, also von Überblicks-Diagrammen hin zu Detail-Diagrammen.

Zustandsautomat

In Abbildung 128 bis Abbildung 139 wird der Zustandsautomat des Idealmodells top-down dargestellt. Dabei wird an einigen Stellen auf die Darstellung von Unterzuständen verzichtet, weil diese entweder im Tutorial detailliert genug behandelt werden, oder es nur einen einzigen Unterzustand gibt.

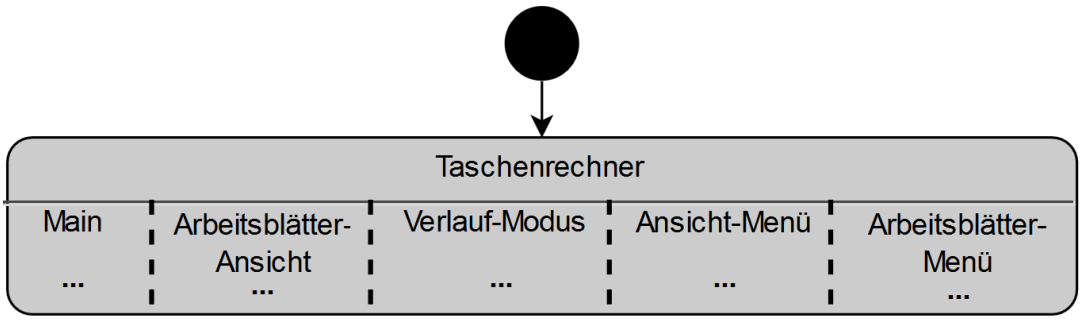


Abbildung 128: Idealmodell – Überblick

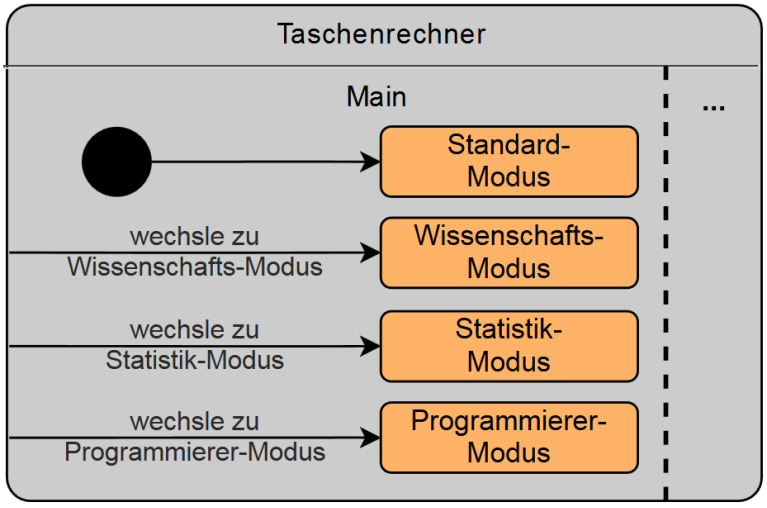


Abbildung 129: Idealmodell – Main-Region

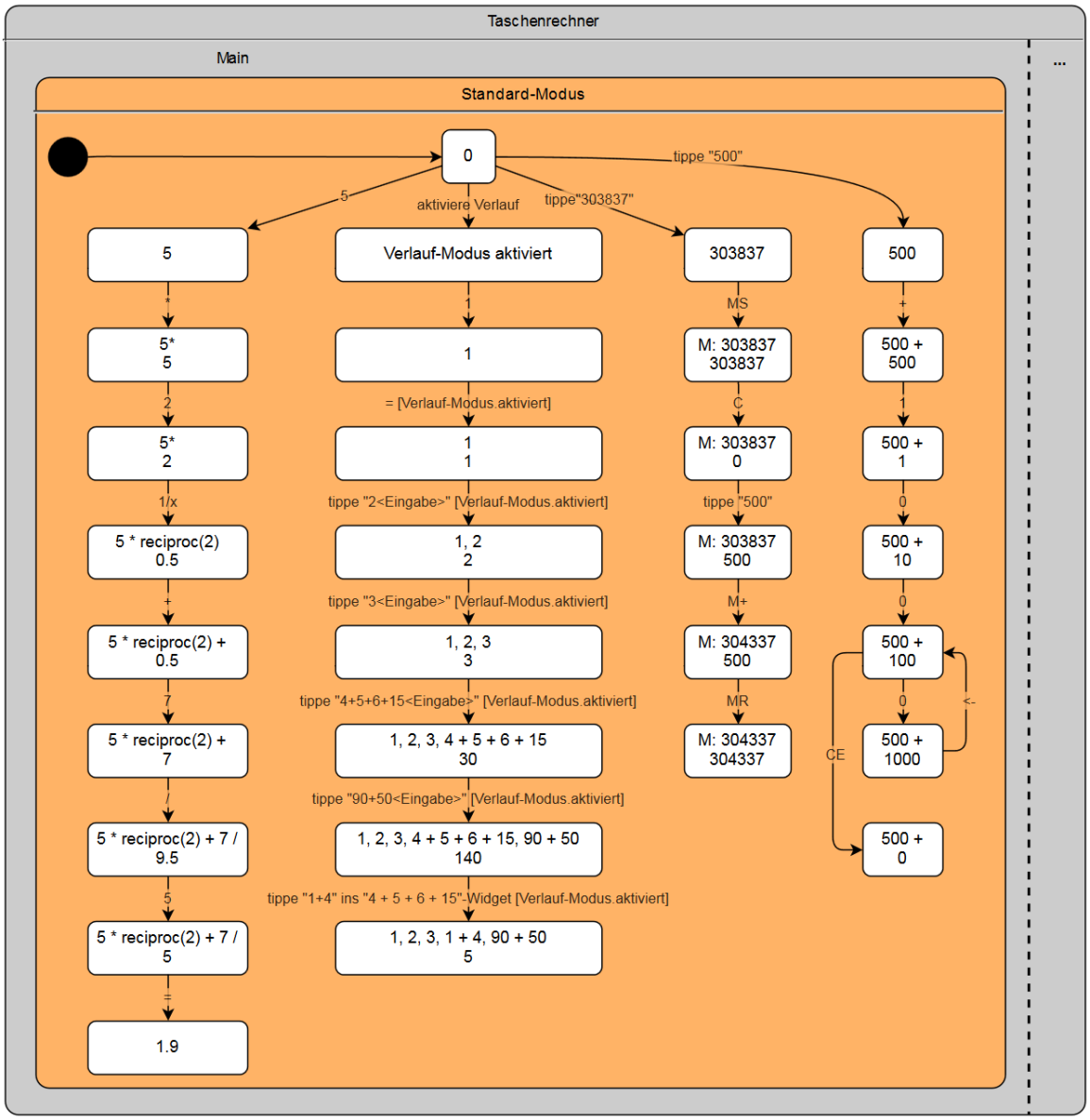


Abbildung 130: Idealmodell – Main-Region – Standard-Modus

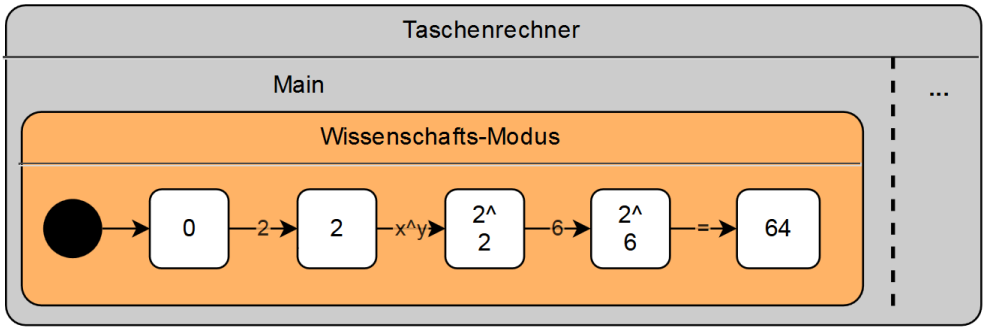


Abbildung 131: Idealmodell – Main-Region – Wissenschafts-Modus

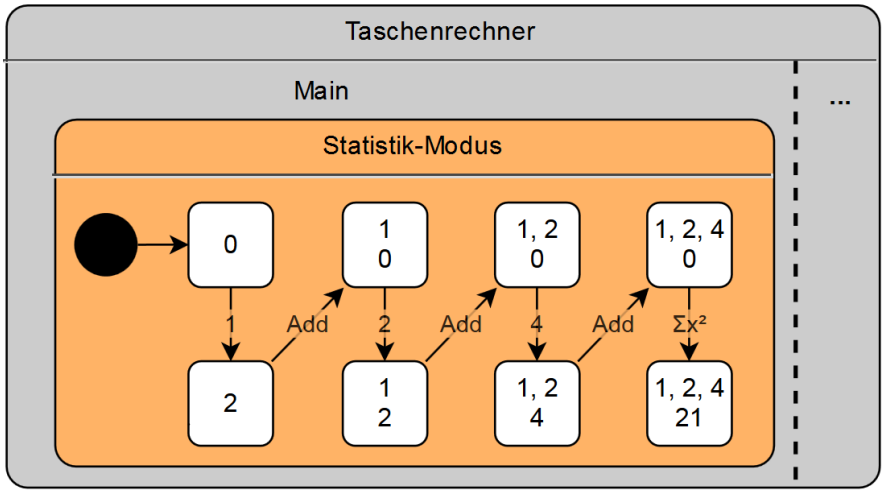


Abbildung 132: Idealmodell – Main-Region – Statistik-Modus

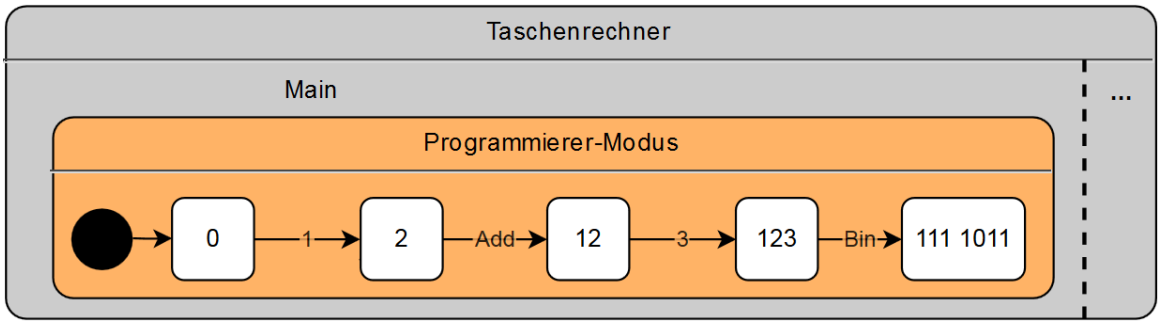


Abbildung 133: Idealmodell – Main-Region – Programmierer-Modus

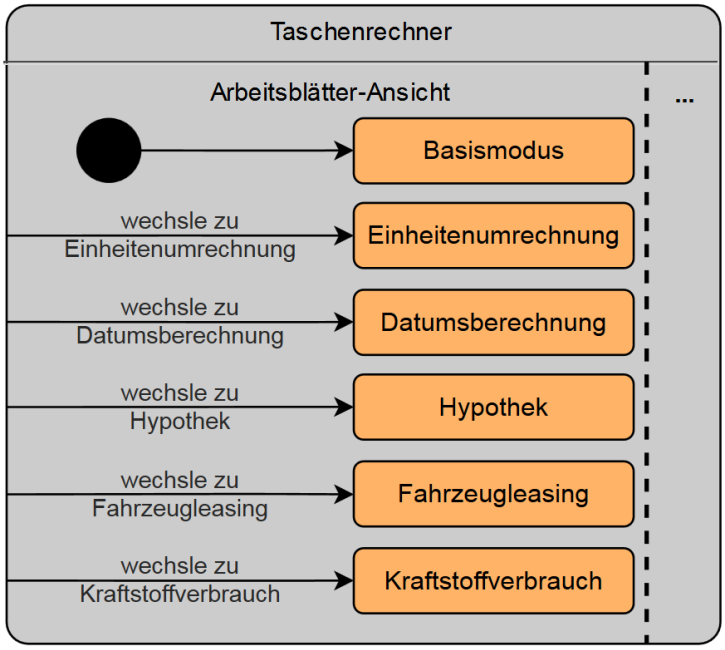


Abbildung 134: Idealmodell – Arbeitsblätter-Ansicht-Region

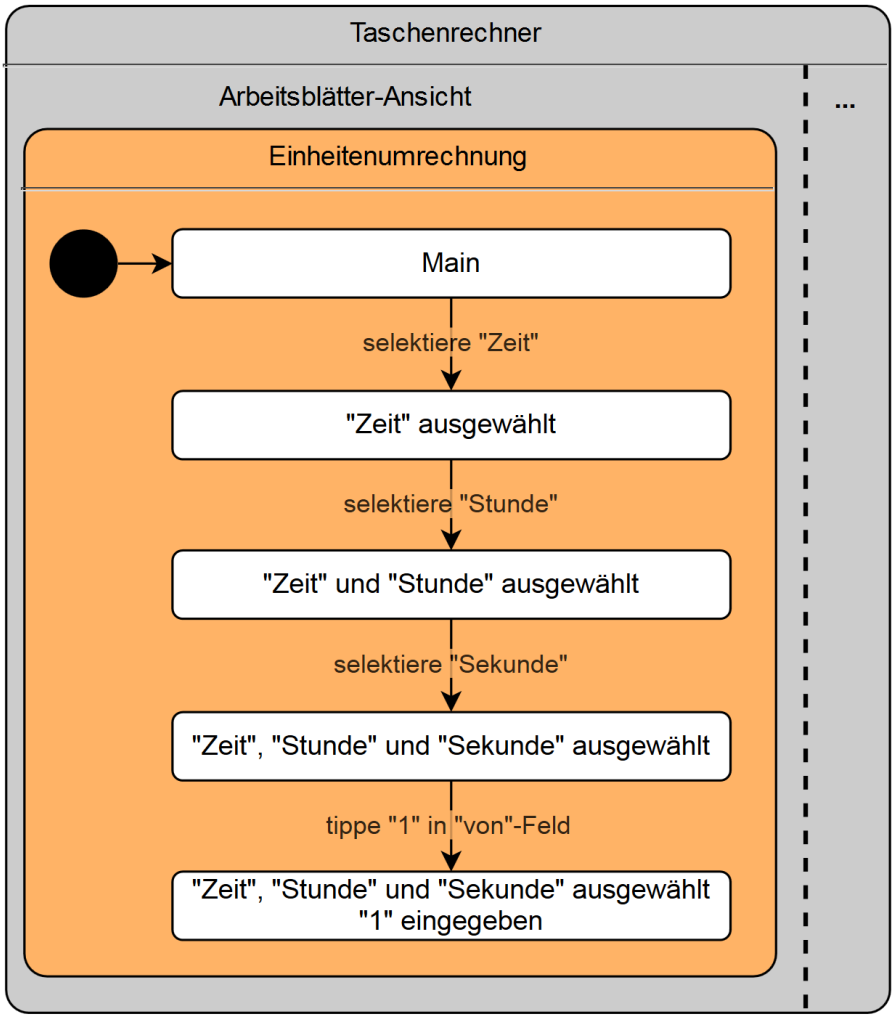


Abbildung 135: Idealmodell – Arbeitsblätter-Ansicht-Region – Einheitenumrechnung

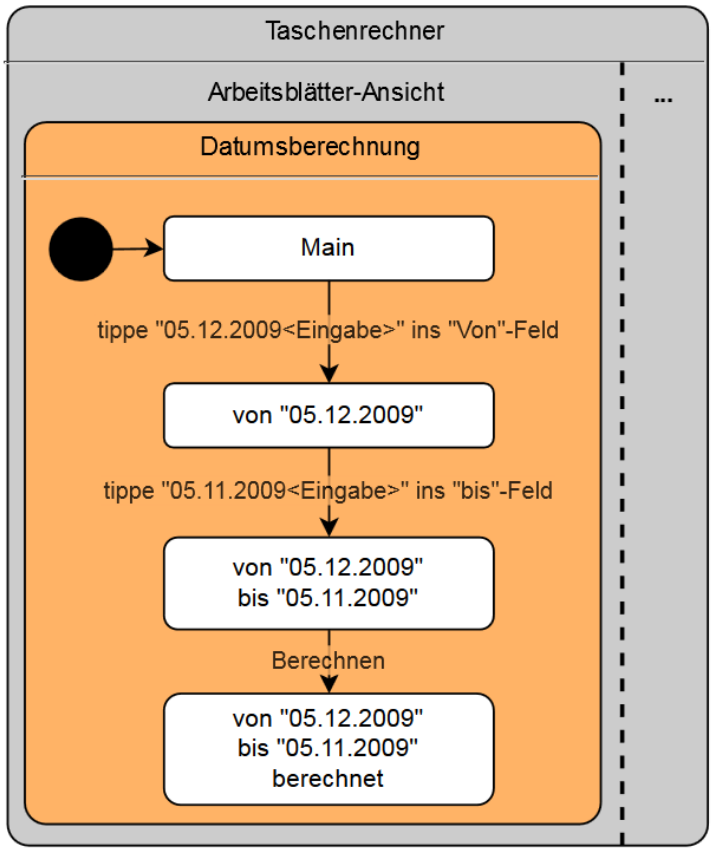


Abbildung 136: Idealmodell – Arbeitsblätter-Ansicht-Region – Datumsberechnung

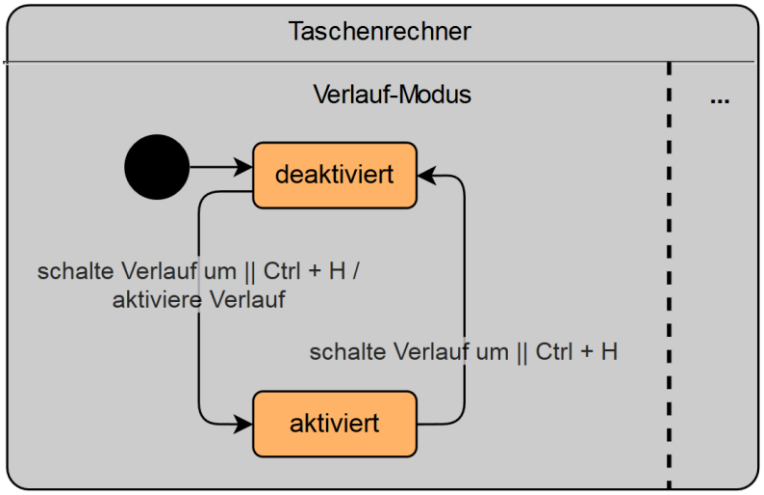


Abbildung 137: Idealmodell – Verlaufs-Modus-Region

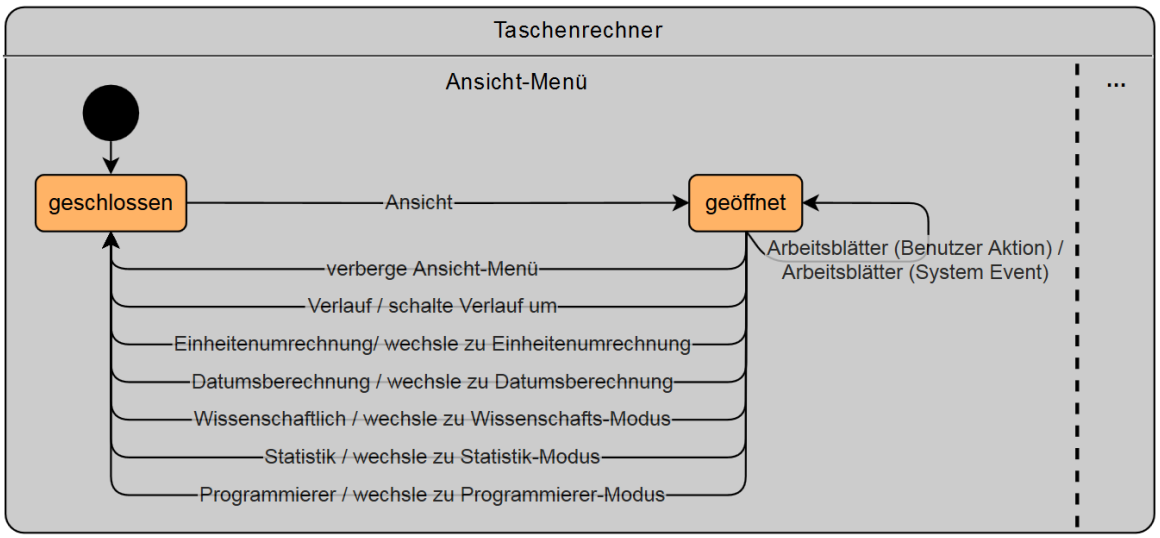


Abbildung 138: Idealmodell –Ansicht-Menü-Region

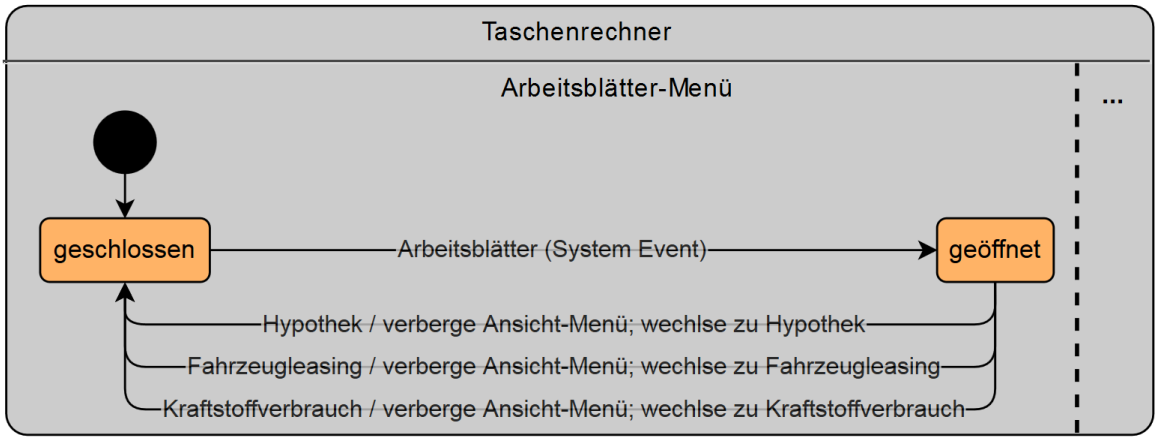


Abbildung 139: Idealmodell – Arbeitsblätter-Menü-Region

Klassifikationsbaum und Testsequenzen

Der Klassifikationsbaum ist analog zum Zustandsautomaten aufgebaut. Zustände werden zu Klassen und Regionen zu Klassifikationen. Abbildung 140 (oben) zeigt beispielhaft die obersten Ebenen des Klassifikationsbaums vom Idealmodell.

Mittels der Klassifikationsbaum-Methode wurde eine Testsequenzen je Abschnitt des Taschenrechner-Tutorials erstellt. Die so entstanden Testsequenzen geben jeweils den beschriebenen Ablauf des Tutorials wieder. Abbildung 140 gibt einen Überblick über die so entstandenen Testsequenzen.

Abbildung 141 zeigt exemplarisch Details einer der Testsequenz und einige Details des Klassifikationsbaumes.

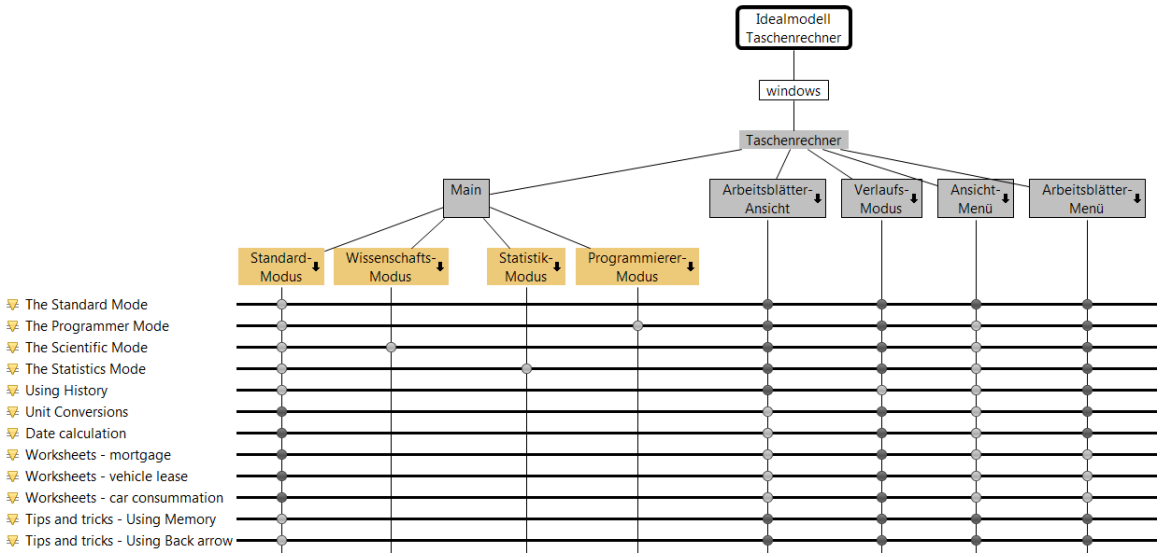


Abbildung 140: Testsequenzen Überblick – Eine Testsequenz je Abschnitt im Tutorial wurde erstellt. Dunkle Punkte in der Testmatrix bedeuten, dass es in der Zugehörigen Testsequenz keine Änderungen innerhalb der Klasse oder Klassifikation gab, helle Punkte bedeuten Veränderung. Beispielsweise ändert die Testsequenz *Using History* etwas in der Region *Verlaufs-Modus*, jedoch nicht in der Region *Arbeitsblätter-Menü*.

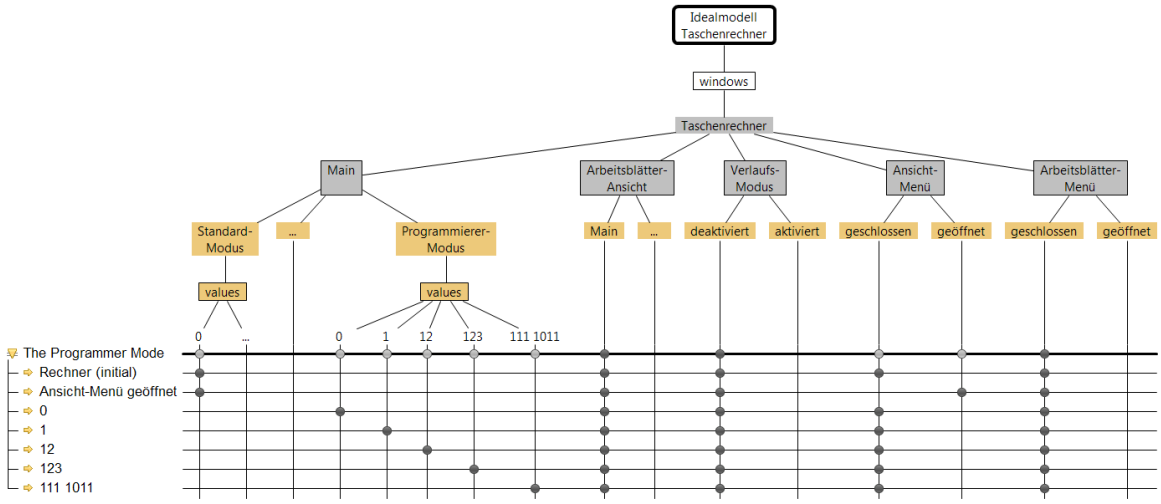


Abbildung 141: Testsequenz Beispiel – Die gezeigte Testsequenz aktiviert, wie im Tutorial beschrieben, den *Programmierer-Modus* des Taschenrechners (Test-Zustände 1 bis 3) und beginnt dann in diesem Modus zu rechnen (Test-Zustände 4 bis 7). Andere Testsequenzen sind analog aufgebaut.

Anhang B: Erwartungsmodell

Im Folgenden wird das Erwartungsmodell des Windows-Taschenrechners im Detail gezeigt. Der Übersichtlichkeit halber werden jeweils Ausschnitte der Modelle auf mehrere Diagramme verteilt dargestellt. Dabei wird top-down vorgegangen, also von Überblicks-Diagrammen hin zu Detail-Diagrammen. Elemente die sich gegenüber dem Idealmodell nicht geändert haben, werden nicht gesondert gezeigt.

In Abbildung 142 bis Abbildung 145 wird der Zustandsautomat des Idealmodells top-down dargestellt. Abbildung 146 zeigt die tatsächlich evaluierte Untermenge des Erwartungsmodells.

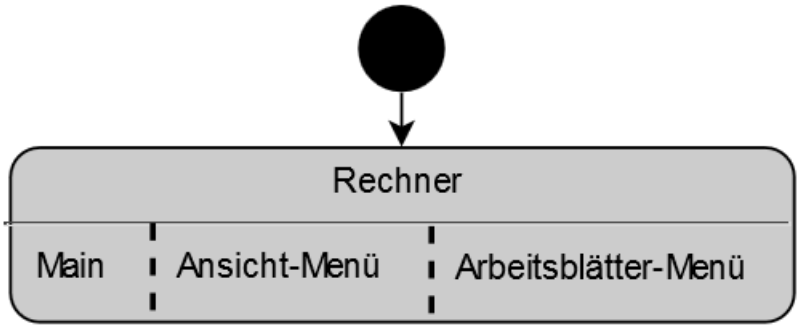


Abbildung 142: Erwartungsmodell – Überblick. Die Regionen *Ansicht-Menü* und *Arbeitsblätter-Menü* sind gleich wie im Idealmodell.

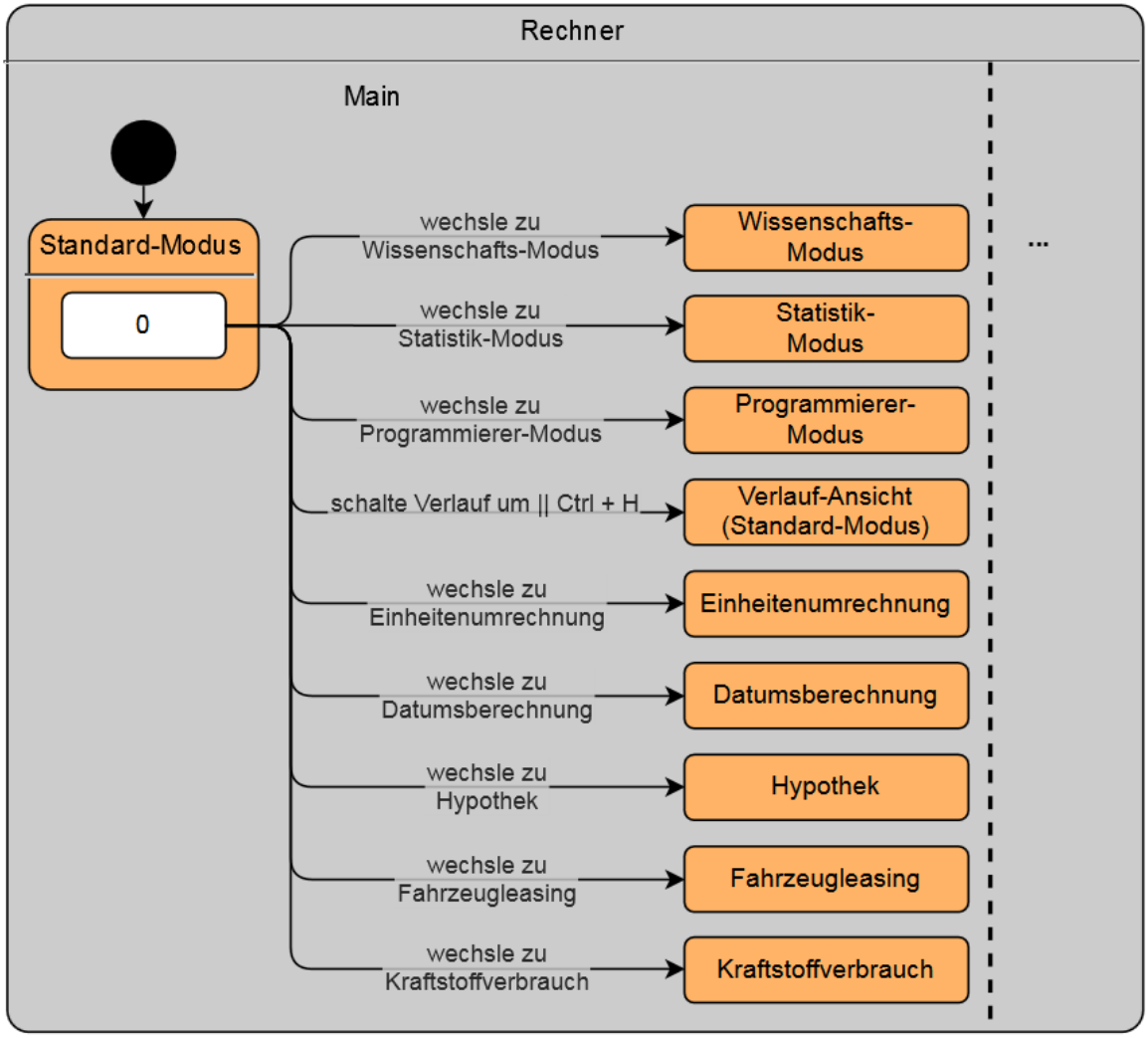


Abbildung 143: Erwartungsmodell – Main-Region. Die Struktur-Zustände *Wissenschafts-Modus*, *Statistik-Modus* und *Programmierer-Modus* sind gleich wie im Erwartungsmodell. Die Strukturzustände *Einheitenumrechnung*, *Datumsberechnung*, *Hypothek*, *Fahrzeugleasing* und *Kraftstoffverbrauch* sind auch gleich mit denen im Idealmodell, jedoch befinden sie sich in einer anderen Region.

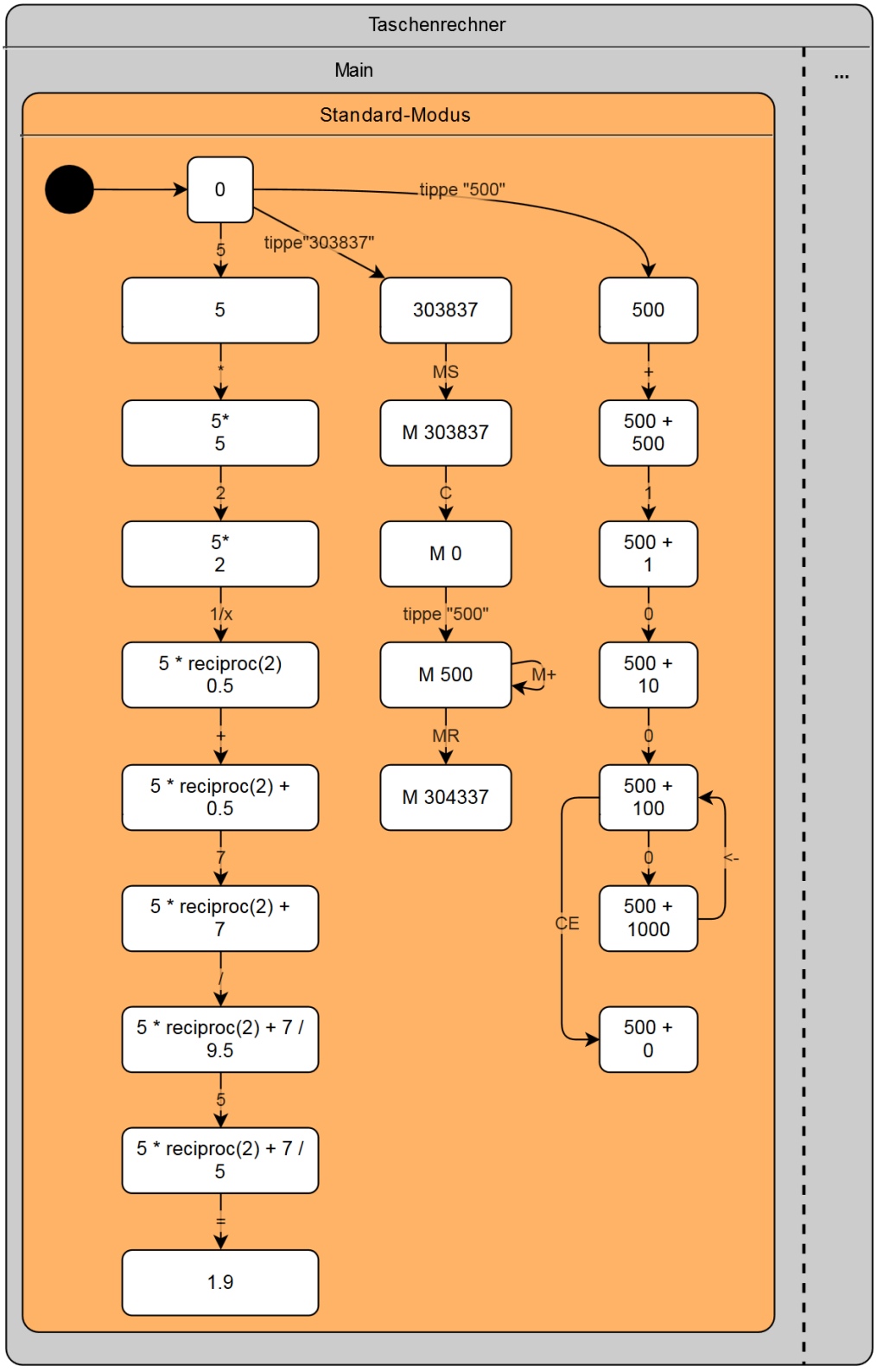


Abbildung 144: Erwartungsmodell – Main-Region – Standard-Modus

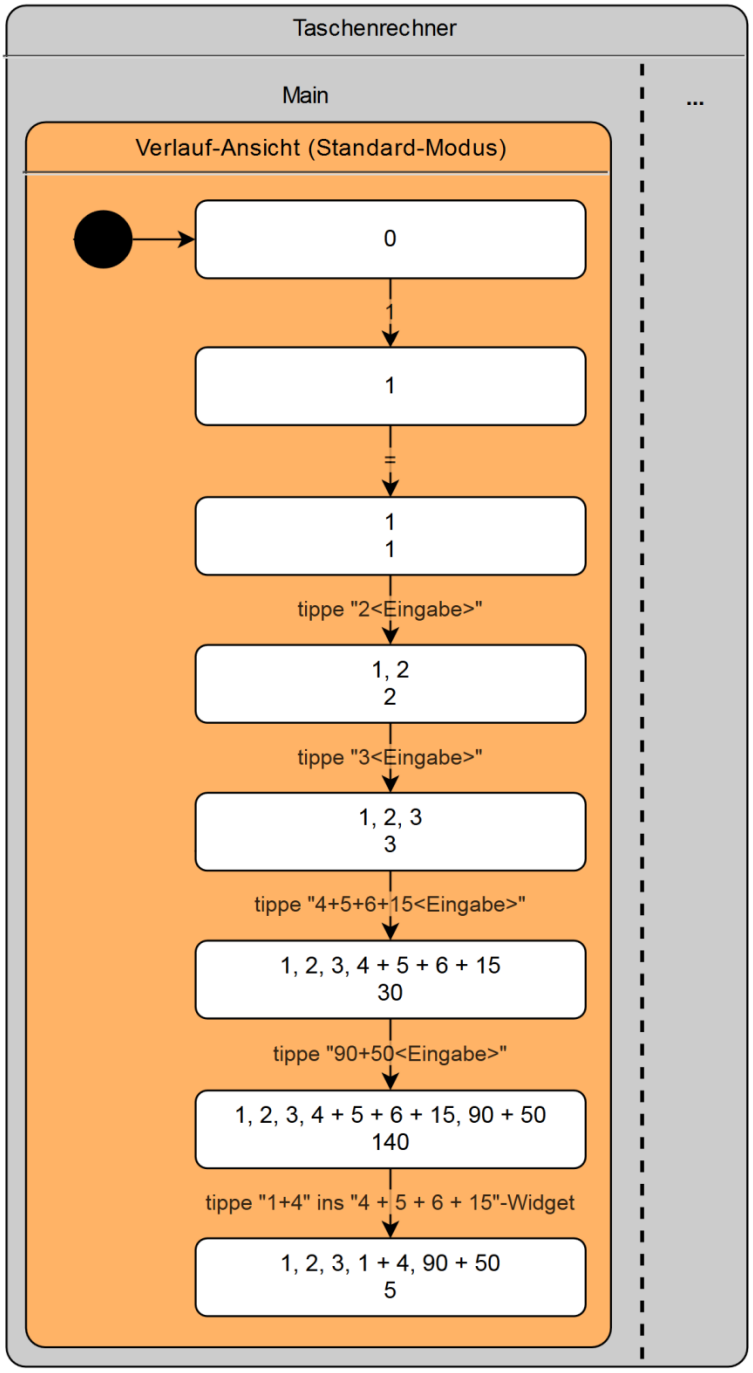


Abbildung 145: Erwartungsmodell – Main-Region – Verlauf-Ansicht (Standard-Modus)

Der Klassifikationsbaum ist analog zum Zustandsautomaten aufgebaut und wird hier nicht zusätzlich dargestellt. Zustände werden zu Klassen und Regionen zu Klassifikationen. Wenn mehrere Transitionen von einem Zustand in einen zweiten führen, werden entsprechende zusätzliche Klassen modelliert (vgl. Abschnitt 4.2.3.5).

Zu erwartende Testsequenzen hängen von den gewählten Überdeckungskriterien und Details der Generierung ab die nicht im Rahmen dieser Arbeit entwickelt wurden. Daher stellt das Erwartungsmodell keine Testsequenzen bereit.

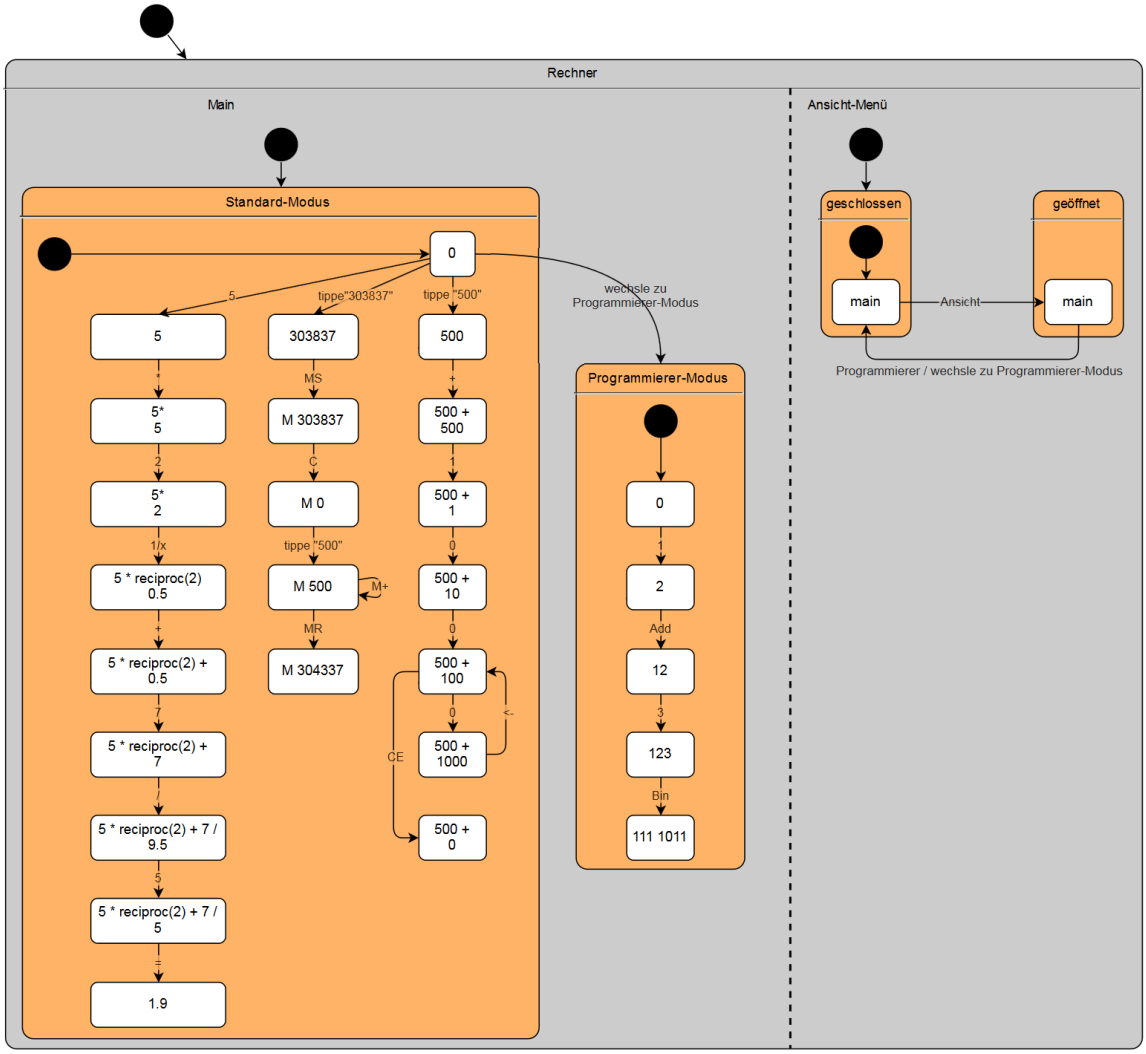


Abbildung 146: Tatsächlich evaluierte Untermenge des Erwartungsmodells.

Anhang C: Realmodell

Im Folgenden wird das Realmodell des Windows-Taschenrechners – also das Modell, welches tatsächlich von der entwickelten Software erzeugt wird – im Detail gezeigt. Der Übersichtlichkeit halber werden jeweils Ausschnitte der Modelle auf mehrere Diagramme verteilt dargestellt. Dabei wird top-down vorgegangen, also von einem Überblicks-Diagramm hin zu Detail-Diagrammen. Auf die gewohnte Farbwahl (grau, orange, weiß), wird hier verzichtet, da es sich um Bildschirmfotos der Software handelt.

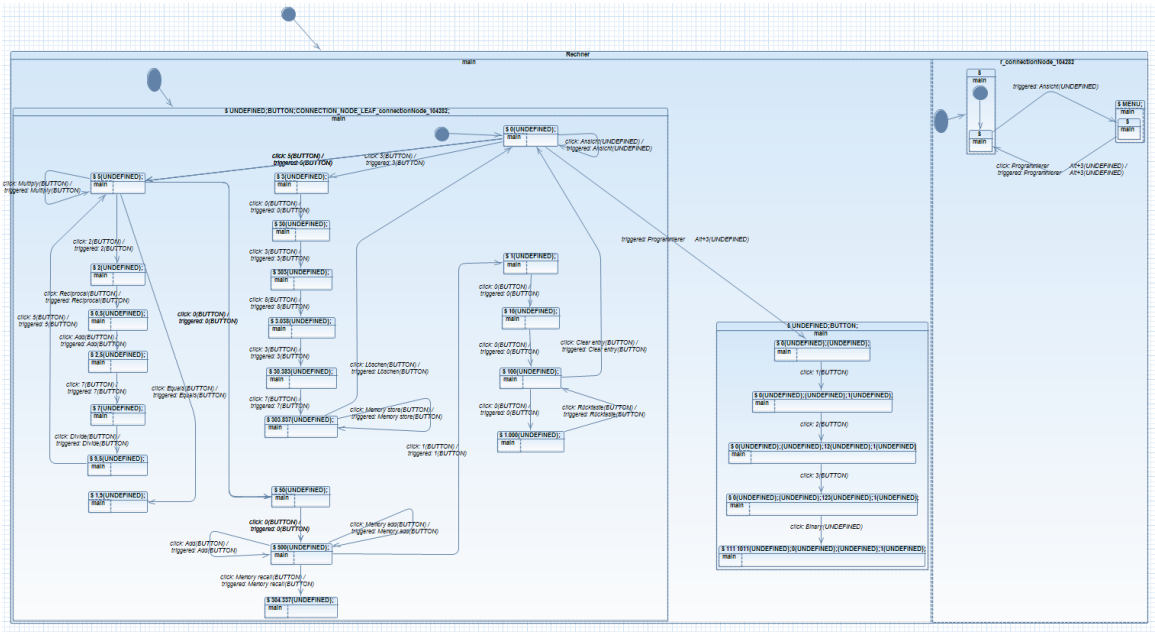


Abbildung 147: Überblick Realmodell – Das Bild zeigt das tatsächlich generierte Modell in seiner Gesamtheit.

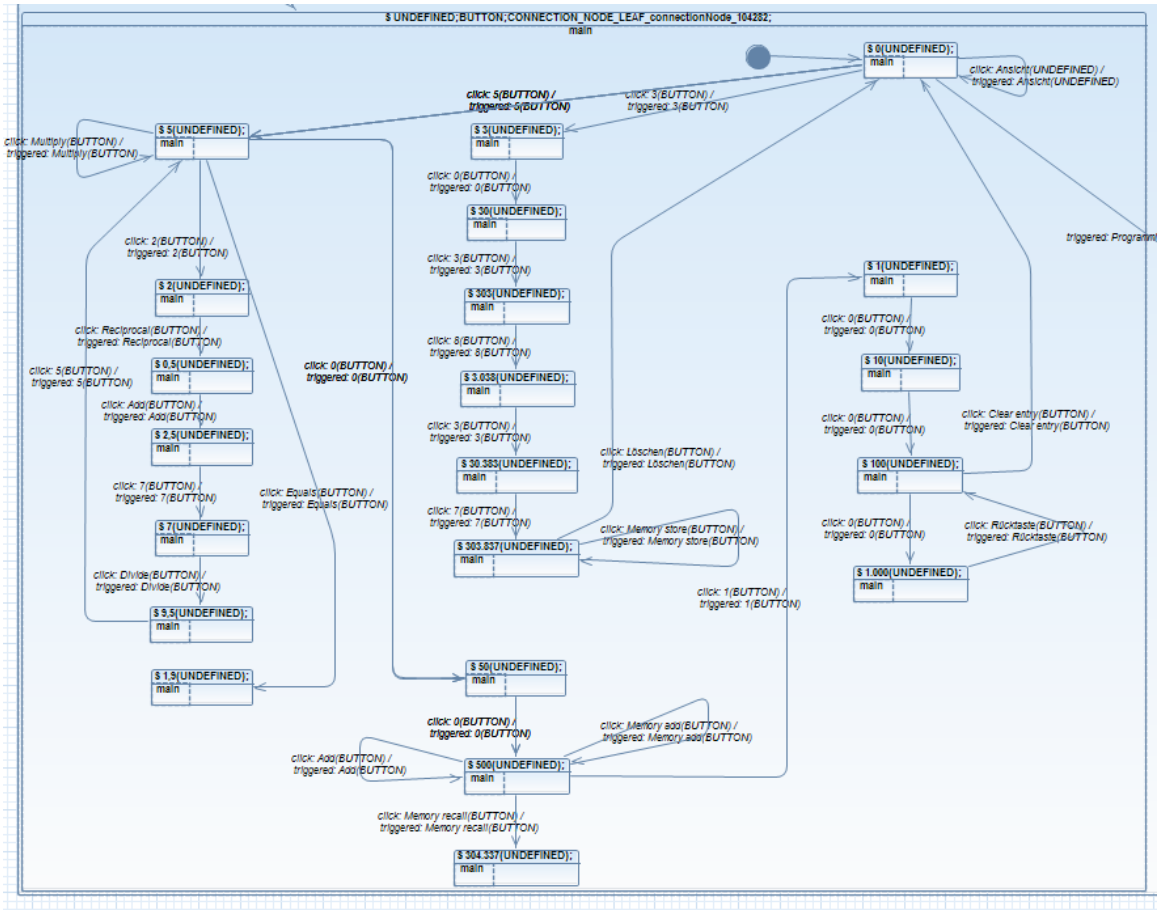


Abbildung 148: Realmodell Ausschnitt – Region: *Main* – Struktur-Zustand: *Standard-Modus*.

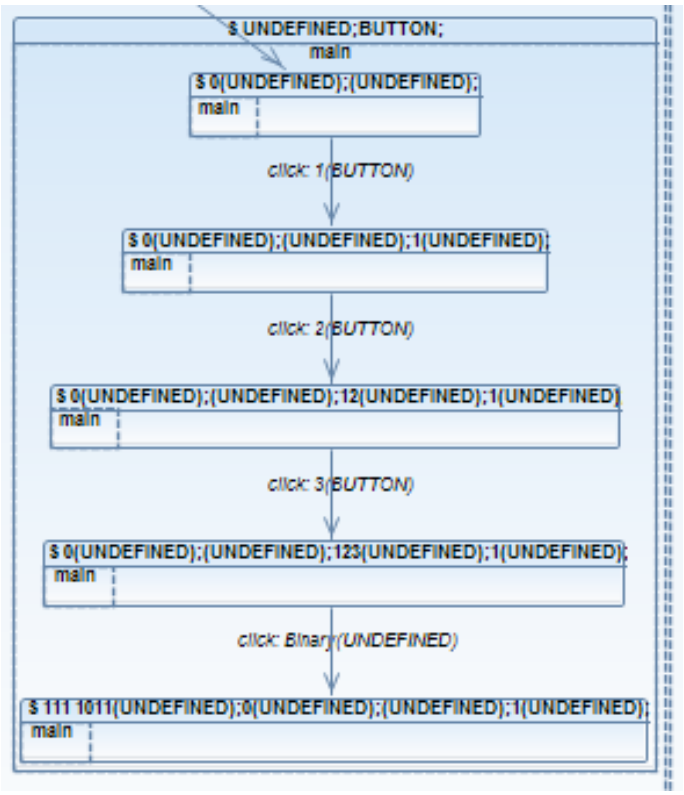


Abbildung 149: Realmodell Ausschnitt – Region: *Main* – Struktur-Zustand: *Programmierer-Modus*.

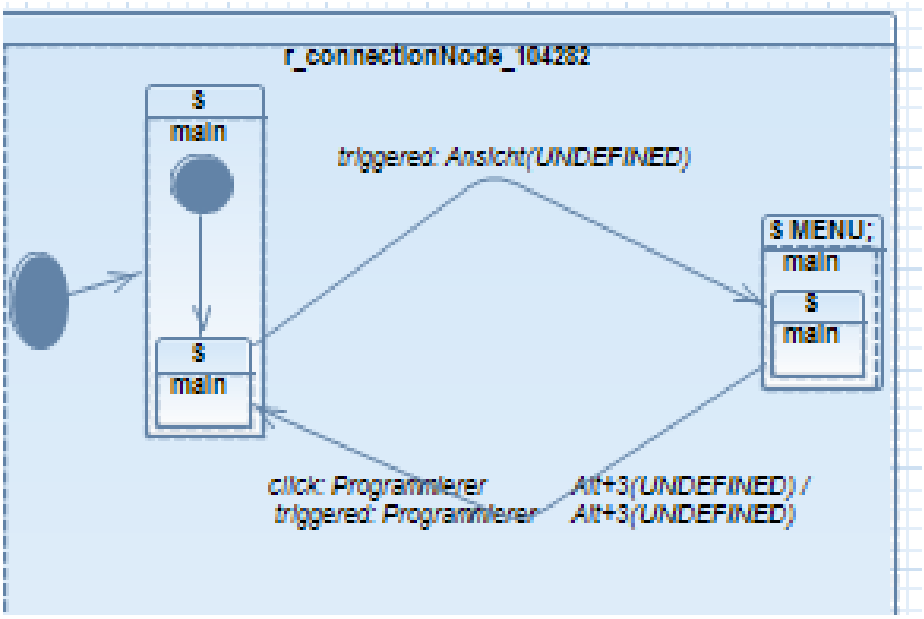


Abbildung 150: Realmodell Ausschnitt – Region: *Ansicht-Menü*.

Literatur

[ABPS12] ARLT, S. ; BERTOLINI, C. ; PAHL, S. ; SCHÄF, M.: Trends in Model-based GUI Testing. In: *Adv. Comput.* Bd. 86 (2012), S. 183–222

[BaMc08] BATH, GRAHAM ; MCKAY, JUDY: *The software test engineer's handbook a study guide for the ISTQB test analyst and technical test analyst advanced level certificates*. 1st. Aufl. Santa Barbara : Rocky Nook, 2008 — ISBN 9781933952246

[Baue11] BAUERSFELD, SEBASTIAN: *A Metaheuristic Approach to Automatic Test Case Generation for GUI-Based Applications*, Humboldt-Universität zu Berlin, 2011

[BaVo12a] BAUERSFELD, SEBASTIAN ; VOS, TANJA E J: A Reinforcement Learning Approach to Automated GUI Robustness Testing. In: *4th Symposium on Search Based-Software Engineering (SSBSE2012), Fast Abstracts, 28-30 September, Riva del Garda, Trento, Italy*. Riva del Garda, Trento, Italy, 2012, S. 7–12

[BaVo12b] BAUERSFELD, SEBASTIAN ; VOS, TANJA E. J.: GUITest: a Java library for fully automated GUI robustness testing. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. New York, New York, USA, ACM Press (2012), S. 330 – 333 — ISBN 9781450312042

[BaWW11] BAUERSFELD, SEBASTIAN ; WAPPLER, STEFAN ; WEGENER, JOACHIM: A Metaheuristic Approach to Test Sequence Generation for Applications with a GUI. In: *Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings*, 2011, S. 173–187

[Beck03] BECK, KENT: Test-Driven Development By Example. In: *Rivers*, *The Addison-Wesley Signature Series*. Bd. 2 : Addison-Wesley, 2003 — ISBN 0321146530, S. 176

[BeHo99] BERGMANN, J.P. ; HOROWITZ, M.A.: Improving coverage analysis and test generation for large designs. In: *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*. San Jose, CA, USA : IEEE, 1999 — ISBN 0-7803-5832-5, S. 580–583

[Bern14] BERNER & MATTNER SYSTEMTECHNIK GMBH: *TESTONA Professional 4.1 User Guide*. Munich, Germany, 2014

[BrMe07] BROOKS, PENELOPE A. ; MEMON, ATIF M.: Automated gui testing guided by usage profiles. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*. New York, New York, USA : ACM Press, 2007 — ISBN 9781595938824, S. 333

[CDDD03] CLAUS, V. ; DEWEß, G. ; DEWEß, M. ; DIEKERT, V. ; FUCHSSTEINER, B. ; GROSCHE, G. ; ZIEGLER, V. ; ZEIDLER, E. ; ZIEGLER, D. (Hrsg.): *Teubner-Taschenbuch der Mathematik - Teil II*. 8. Auflage. Aufl. : Vieweg+Teubner Verlag, 2003 — ISBN 978-3519210085

[GCSY13] GIVENS, PAUL ; CHAKAROV, ALEKSANDAR ; SANKARANARAYANAN, SRIRAM ; YEH, TOM: Exploring the internal state of user interfaces by combining computer vision techniques with grammatical inference. In: NOTKIN, D. ; CHENG, B. H. C. ; POHL, K. (Hrsg.): *ICSE '13 Proceedings of the 2013 International Conference on Software Engineering*. San Francisco, CA, USA : IEEE / ACM, 2013 — ISBN 978-1-4673-3076-3, S. 1165–1168

[Gont13] GONTARIU, FLORIN: *The Calculator in Windows 7 & Windows 8 - A Tool for the Geek in You!* . — http://www.7tutorials.com/windows-calculator-tool-geek-you

[GrGr93] GROCHTMANN, MATTHIAS ; GRIMM, KLAUS: Classification trees for partition testing. In: *Software Testing, Verification and Reliability* Bd. 3 (1993), Nr. 2, S. 63–82

[Groc94] GROCHTMANN, MATTHIAS: Test Case Design Using Classification Trees. In: *STAR'94*. Washington, D.C., 1994

[GrXF09] GRECHANIK, MARK ; XIE, QING ; FU, CHEN: Maintaining and evolving GUI-directed test scripts. In: *2009 IEEE 31st International Conference on Software Engineering*, IEEE Computer Society (2009), S. 408–418 — ISBN 978-1-4244-3453-4

[Hare87] HAREL, DAVID: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* Bd. 8 (1987), S. 231–274

[Harr03] HARRIS, IAN G: Fault Models and Test Generation for Hardware-Software Covalidation. In: *IEEE Design and Test* Bd. 20 (2003), Nr. 4, S. 40–47

[HuCM10] HUANG, SI ; COHEN, MYRA B. ; MEMON, ATIF M.: Repairing GUI Test Suites Using a Genetic Algorithm. In: *ICST 2010: Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation*, IEEE Computer Society (2010), S. 245–254 — ISBN 978-1-4244-6435-7

[Jaco83] JACOB, ROBERT J K: Using Formal Specifications in the Design of a Human-computer Interface. In: *Commun. ACM* Bd. 26. New York, NY, USA, ACM (1983), Nr. 4, S. 259–264

[JaTe07] JANETSCHEK, VOLKER ; TEGOS, NICOS: *ATOSj : Ein Werkzeug für den automatisierten Regressionstest oberflächenbasierter Java-Programme*, Humboldt-Universität zu Berlin, 2007

[KrLu10] KRUSE, P. M. ; LUNIAK, M.: Automated Test Case Generation Using Classification Trees. In: *Software Quality Professional Magazine* Bd. 13 (2010), Nr. 1, S. 4–12

[KrNF14] KRUSE, PETER M ; NASAREK, JIRKA ; FERNANDEZ, NELLY CONDORI: Systematic Testing of Web Applications with the Classification Tree Method. In: *XVII*

Iberoamerican Conference on Software Engineering (CIbSE 2014), 2014, S. 219–232

[Krus11] KRUSE, PETER M: Test Sequence Generation from Classification Trees using Multi-agent Systems. In: *EUMAS 2011*. Maastricht, NL, 2011

[KrWe12] KRUSE, PETER M. ; WEGENER, JOACHIM: Test Sequence Generation from Classification Trees. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Montreal, QC : IEEE, 2012 — ISBN 978-0-7695-4670-4, S. 539–548

[Lano09] LANO, KEVIN: UML 2 Semantics and Applications. In: *UML 2 Semantics and Applications* : John Wiley & Sons, Inc., 2009 — ISBN 9780470522622, S. 78–79

[LeWe00] LEHMANN, ECKARD ; WEGENER, JOACHIM: Test Case Design by Means of the CTE XL. In: *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000)*. Kopenhagen, Denmark, 2000

[MaMc85] MARTIN, JAMES ; MCCLURE, CARMA: Diagramming techniques for analysts and programmers, Prentice-Hall, Inc. (1985) — ISBN 0-13-208794-4

[Mari97] MARICK, BRIAN: Classic Testing Mistakes Theme One : The Role of Testing. In: *STAR'97 Int'l Conference on Software Testing Analysis & Review*, 1997

[MaTo10] MARCHETTO, ALESSANDRO ; TONELLA, PAOLO: Using search-based algorithms for Ajax event sequence generation during testing. In: *Empirical Software Engineering* Bd. 16 (2010), Nr. 1, S. 103–140

[MeBN03a] MEMON, A. ; BANERJEE, I. ; NAGARAJAN, A.: GUI ripping: reverse engineering of graphical user interfaces for testing. In: *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on* : IEEE Computer Society, 2003 — ISBN 0-7695-2027-8, S. 260–269

[MeBN03b] MEMON, A. ; BANERJEE, I. ; NAGARAJAN, A.: What test oracle should I use for effective GUI testing? In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* : IEEE Comput. Soc, 2003 — ISBN 0-7695-2035-9, S. 164–173

[Memo01] MEMON, ATIF M.: *PhD Thesis - A comprehensive framework for testing graphical user interfaces*, University of Pittsburgh, 2001

[Memo02] MEMON, ATIF M: SOFTWARE TECHNOLOGIES GUI Testing : Pitfalls and Process. In: *IEEE Computer* (2002), S. 90–91

[Memo07] MEMON, ATIF M.: An event-flow model of GUI-based applications for testing. In: *Software Testing, Verification & Reliability* Bd. 17, John Wiley and Sons Ltd. (2007), S. 137–157

[MePS00] MEMON, ATIF M. ; POLLACK, MARTHA E. ; SOFFA, MARY LOU: Automated test oracles for GUIs. In: *ACM SIGSOFT Software Engineering Notes* Bd. 25, ACM (2000), Nr. 6, S. 30–39 — ISBN 1-58113-205-0

[MeSo03] MEMON, ATIF M ; SOFFA, MARY LOU: Regression Testing of GUIs. In: *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA : ACM Press, 2003 — ISBN 1581137435, S. 118–127

[NaKr13] NASAREK, JIRKA ; KRUSE, PETER M.: Test graphischer Benutzeroberflächen mit der Klassifikationsbaum-Methode am Beispiel von Webanwendungen. In: *Softwaretechnik-Trends* Bd. 33. Ingolstadt (2013), Nr. 4

[Nasa13] NASAREK, JIRKA: *Diplomarbeit - Testen von Webanwendungen mittels CTE und Selenium*, HUMBOLDT-UNIVERSITÄT ZU BERLIN, 2013

[Nati14] NATIONAL INSTRUMENTS: *Argumente für die Automatisierung*. URL http://www.ni.com/white-paper/14576/de/. - abgerufen am 2015-04-13. — http://www.ni.com/white-paper/14576/de/

[NgMT12] NGUYEN, CU D ; MARCHETTO, ALESSANDRO ; TONELLA, PAOLO: Combining Model-Based and Combinatorial Testing for Effective Test Case Generation. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012*. New York, NY, USA, 2012 — ISBN 9781450314541, S. 100–110

[Parn69] PARNAS, DAVID L.: On the use of transition diagrams in the design of a user interface for an interactive computer system. In: *Proceedings of the 1969 24th national conference on* -. New York, New York, USA : ACM Press, 1969, S. 379–385

[QuNa13] QURESHI, IMRAN ALI ; NADEEM, AAMER: GUI Testing Techniques: A Survey. In: *International Journal of Future Computer and Communication* Bd. 2 (2013), Nr. 2, S. 142–146

[RiTo01] RICCA, F. ; TONELLA, P.: Analysis and testing of Web applications. In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. Washington, DC, USA : IEEE Comput. Soc, 2001 — ISBN 0-7695-1050-7, S. 25–34

[RuJB04] RUMBAUGH, JAMES ; JACOBSON, IVAR ; BOOCH, GRADY: Unified Modeling Language Reference Manual, The (2nd Edition), Pearson Higher Education (2004) — ISBN 0321245628

[RuPr07] RUIZ, A. ; PRICE, Y.W.: Test-Driven GUI Development with TestNG and Abbot. In: *IEEE Software* Bd. 24, IEEE Computer Society Press (2007), Nr. 3, S. 51–57

[Same08] SAMEK, MIRO: *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. 2nd revise. Aufl. : Butterworth Heinemann, 2008. — Seite 63 Referenziert. — ISBN 0750687061

[ShSi97] SHEHADY, R.K. ; SIEWIOREK, D.P.: A method to automate user interface testing using variable finite state machines. In: *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing* : IEEE Comput. Soc, 1997 — ISBN 0-8186-7831-3, S. 80–88

[SRWL14] SPILLNER, ANDREAS ; ROßNER, THOMAS ; WINTER, MARIO ; LINZ, TILO: *Praxiswissen Softwaretest - Testmanagement – Aus- und Weiterbildung zum Certified Tester – Advanced Level nach ISTQB-Standard*. 4., überar. Aufl. : Dpunkt.verlag, 2014 — ISBN 978-3-86491-515-4

[Turp08] TURPIN, MICHAEL: Survey of GUI Testing Processes (2008) — ISBN 3205896300

[VLHS06] VIEIRA, MARLON ; LEDUC, JOHANNE ; HASLING, BILL ; SUBRAMANYAN, RAJESH ; KAZMEIER, JUERGEN: Automation of GUI testing using a model-driven approach. In: *Proceedings of the 2006 international workshop on Automation of software test - AST '06*. New York, New York, USA : ACM Press, 2006 — ISBN 1595934081, S. 9

[Xie06] XIE, QING: Developing cost-effective model-based techniques for GUI testing. In: *Proceeding of the 28th international conference on Software engineering - ICSE '06*. New York, New York, USA : ACM Press, 2006 — ISBN 1595933751, S. 997

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den Datum eintragen .....