TECHNISCHE
UNIVERSITÄT
DARMSTADT

Rekonfigurierbare Software-Systeme: Spezifikation und Testfallgenerierung

VOM FACHBEREICH INFORMATIK
DER TECHNISCHEN UNIVERSITÄT DARMSTADT
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
EINES DOKTORS DER NATURWISSENSCHAFTEN (DR. RER. NAT.)
GENEHMIGTE DISSERTATION

VON

JOHANNES BÜRDEK

ERSTGUTACHTER: PROF. DR. RER. NAT. ANDY SCHÜRR
ZWEITGUTACHTERIN: PROF. DR.-ING. INA SCHAEFER

TAG DER EINREICHUNG: 20.07.2018
TAG DER DISPUTATION: 29.10.2018

DARMSTADT 2018

Bürdek, Johannes
Rekonfigurierbare Software-Systeme: Spezifikation und Testfallgenerierung
Darmstadt, Technische Universität Darmstadt,
Jahr der Veröffentlichung der Dissertation auf TUprints: 2018
URN: urn:nbn:de:tuda-tuprints-81615
Tag der mündlichen Prüfung: 29.10.2018

KURZFASSUNG

Hochkonfigurierbare Software wird heutzutage in vielen modernen Anwendungsdomänen eingesetzt, um immer stärker individualisierte Anforderungen bedienen zu können. Eine Schlüsseltechnologie zum Umgang mit konfigurierbarer Software ist Software-Produktlinienentwicklung, bei der eine Software-Familie, bestehend aus ähnlichen Produkten, entwickelt wird. Die Variabilität der Funktionalität der Produkte wird dabei in Features gekapselt, welche die für Kunden sichtbaren Konfigurationsoptionen einer Software-Produktlinie (SPL) darstellen. Moderne Software-Systeme müssen jedoch nicht nur hochkonfigurierbar sein, sondern auch unter sich kontinuierlich ändernden Bedingungen, wie wechselnden Anforderungen, funktionieren. Beispielsweise müssen Produktivsysteme aus der Automatisierungstechnik immer flexibler werden, um die Anforderungen neuer Produktionsmethoden im Industrie-4.0-Kontext zu erfüllen. Eine Erweiterung von SPLs zur Modellierung solcher individuell konfigurierbaren, laufzeitadaptiven Systeme mit der Fähigkeit der Rekonfiguration zur Laufzeit sind dynamische Software-Produktlinien (DSPL).

Speziell im Kontext von missions- und sicherheitskritischen Systemen erfordert das Design zuverlässiger DSPLs eine präzise Spezifikation des funktionalen und des Rekonfigurationsverhaltens. Aktuellen Arbeiten zur Spezifikation von DSPLs fehlt es jedoch an Möglichkeiten, um komplexe Restriktionen auf Konfigurationsparametern sowie entsprechenden Rekonfigurationsoptionen präzise formulieren zu können. Weiterhin ist eine umfassende Qualitätssicherung von DSPLs mit Fokus auf deren Rekonfigurationsverhalten erforderlich, um ein Fehlverhalten sicherheitskritischer DSPLs möglichst auszuschließen. In der Praxis ist das Testen die etablierteste und am besten skalierende Qualitätssicherungstechnik. Jedoch beschreiben aktuelle Arbeiten zum Testen von DSPLs im Allgemeinen Test-Frameworks, die das Vorhandensein einer initialen, manuell erstellen Test-Suite voraussetzen, aber die Erstellung dieser Test-Suiten außer Acht lassen.

In dieser Arbeit stellen wir einen ganzheitlichen Ansatz zur Spezifikation und Validierung von DSPLs sowie zur automatisierten Generierung von Testfällen für DSPLs vor. Dabei werden existierende Spezifikationsansätze für DSPLs dahingehend erweitert, dass Features in einem flexiblen stufenweisen Konfigurationsprozess mit multiplen Bindungszeiten konfiguriert werden können und logische sowie temporale Abhängigkeiten zwischen Features durch komplexe Bindungszeitrestriktionen präzise ausgedrückt werden können. Darauf aufbauend werden automatisierte Validierungsansätze für DSPL-Spezifikationen vorgestellt, welche essentielle Validitätseigenschaften von stufenweisen Konfigurationsprozessen mit komplexen Bindungszeitrestriktionen sicherstellen.

Weiterhin wird ein Ansatz zur effizienten Test-Suite-Generierung mit Fokus auf dem Rekonfigurationsverhalten von DSPLs präsentiert. Da Testgenerierung aufgrund wiederholter berechnungsintensiver Erreichbarkeitsanalysen für verschiedene Testziele gemäß einem Abdeckungskriterium schon für einzelne Produkte ein schweres Problem darstellt, ist es das Ziel die Wiederverwendbarkeit von Ergebnissen zwischen den Erreichbarkeitsanalysen zu erhöhen. Hierfür werden zuerst existierende Model-Checker-basierte Testgenerierungsansätze mit aktuellen Techniken aus dem Bereich Multi-Property-Checking zur Erhöhung der Wiederverwendung von Erreichbarkeitsanalyseergebnissen zur Abdeckung mehrerer Testziele kombiniert. Dieser Ansatz wird um die Möglichkeit erweitert, Testfälle für ganze Software-Familien zu erstellen und dadurch Erreichbarkeitsanalyseergebnisse zwischen Produkten wiederzuverwenden. Zum Schluss wird der Ansatz so erweitert, dass bei der Testgenerierung das Rekonfigurationsverhalten von DSPLs mitberücksichtigt wird.

Die in dieser Arbeit vorgestellten Konzepte werden anhand von Beispielen aus der Praxis motiviert und illustriert sowie auf Basis prototypischer Implementierungen evaluiert.

ABSTRACT

Nowadays, software becomes more and more configurable to be adaptable to the ever-growing diversity of requirements and customers' needs. A key technology to cope with configurable software is software product line engineering, aiming at developing software families consisting of similar but still distinguishable products. Features encapsulate the variability of the functionality of those products. Hence, features denote user-visible configuration options of a software product line (SPL). However, modern software systems not only have to cope with variability, but also with ever-changing conditions such as changing requirements. Dynamic software product lines (DSPL) extend SPLs to model such individually configurable, runtime-adaptive systems being able to reconfigure at runtime.

Especially in the context of mission- and safety-critical software systems, the design of reliable DSPLs requires capabilities for a precise specification of the systems' functional and reconfiguration behavior. Nevertheless, recent publications related to the specification of DSPLs left the possibility of a precise formulation of complex constraints among configuration parameters and/or respective reconfiguration options out of scope. Besides a precise specification, extensive quality assurance activities for DSPLs with a focus on their reconfiguration capabilities are indispensable to prevent any erroneous behavior of mission- and safety-critical DSPLs at runtime. In practice, software testing is the most established and scalable quality-assurance technique. However, recent work concerning DSPL testing rather provide general testing frameworks expecting a manually created initial test suite as input than a technique to systematically create or even automatically generate DSPL test suites for testing the functional and the reconfiguration behavior of DSPLs.

In this thesis, we present a comprehensive approach for the specification and validation of DSPLs as well as the automated generation of DSPL test suites. To this end, we extend existing DSPL specification approaches by the capabilities to configure features in a flexible staged-configuration process with multiple binding times and to formulate logical as well as temporal dependencies between features by means of complex binding-time constraints. On top of this DSPL specification approach, we further provide automated validation techniques for DSPL specifications to verify essential validity properties of staged configuration processes with complex binding-time constraints during domain engineering.

Furthermore, we present an efficient test-suite generation technique focusing on the reconfiguration behavior of DSPLs. Generating efficient test suites remains computationally expensive even for single products, consisting of repetitive reachability analyses for multiple test goals according to a specific coverage criterion. This situation is even worse when generating test suites for entire software families as apparent in the context of (D)SPLs. Hence, our test-suite generation technique aims at reusing reachability-analysis results to reduce computational efforts at three consecutive levels, the product level, the SPL level and the DSPL level. At product level, we combine existing model-checking-based test-suite generation approaches with recently developed techniques from the area of multi-property checking to increase the reusability of reachability-analysis results among test goals by processing and covering multiple test goals in parallel. At SPL level, this approach is extended by the capability to generate test suites for an entire software family by reusing reachability-analysis results among products. Finally, we extend the approach at DSPL level by considering the reconfiguration behavior of DSPLs beside their functional behavior.

The concepts and techniques contributed in this thesis are primarily motivated and illustrated by means of a real-world high-configurable runtime-adaptive system from the automation domain. Based on prototypical implementations, we provide an evaluation of the presented techniques using various real-world case studies.

INHALTSVERZEICHNIS

AKRONYME

ARG	Abstrakter Erreichbarkeitsgraph (engl. **A**bstract **R**eachability **G**raph)
BDD	**B**inary **D**ecision **D**iagram
CFA	Kontrollflussautomat (engl. **C**ontrol-**F**low **A**utomaton)
DCU	**D**evice **C**ontrol **U**nit
DSPL	**D**ynamische **S**oftware-**P**rodukt**l**inie
FQL	**F**Shell **Q**uery **L**anguage
HIT	**H**eidelberger **I**onenstrahl-**T**herapiezentrum
IUT	**I**nstanz **U**nter **T**est
SPL	**S**oftware-**P**roduktlinie
SPLE	**S**oftware-**P**rodukt**l**inien**e**ntwicklung
SPS	**S**ymbolische **P**fad**s**ensitivität
SSA	**S**tatic **S**ingle **A**ssignment
TGA	Testzielautomat (engl. **T**est-**G**oal **A**utomaton)

1

MOTIVATION UND ÜBERBLICK

Hochkonfigurierbare Software-Systeme sind heutzutage in vielen modernen Anwendungsdomänen essentiell, um immer stärker individualisierte Anforderungen und Kundenbedürfnisse bedienen zu können. Beispielsweise enthalten moderne Software-Systeme in der Automatisierungstechnik eine hohe Bandbreite von Konfigurationsoptionen, um einzelne Geräte oder sogar komplette Produktionsanlagen an kundenspezifische Anforderungen und Wünsche anzupassen [FDG08, MJ10, FFVH12, LBL+14]. Auch in anderen Anwendungsdomänen ist ein Trend hin zu hochkonfigurierbaren Software-Systemen erkennbar, z. B. in Fahrzeug-, Informations- und mobilen Systemen [WCKK06]. In der Literatur wird die Tendenz immer stärker konfigurierbare Software-Systeme zu entwickeln und Kunden dadurch inhärent variantenreiche Software-Systeme anzubieten, häufig unter dem Begriff *Variabilität im Raum* zusammenfasst [AK09].

Bei der Spezifikation solcher konfigurierbaren Software-Systeme werden Konfigurationsoptionen aus der Problemdomäne üblicherweise mehreren Implementierungsartefakten im Lösungsraum zugeordnet. Dies führt dazu, dass die Variabilität solcher Software-Systeme sowohl die Funktionalität der Software-Systeme als auch ihre plattformspezifischen Eigenschaften sowie die unterstützten Anwendungskontexte zunehmend beeinflussen. Somit muss die zugrundeliegende Kontroll-Software eines solchen Software-Systems ebenfalls mit der inhärenten Variabilität des Systems umgehen können.

Eine Schlüsseltechnologie zum Umgang mit konfigurierbarer Software ist Software-Produktlinienentwicklung (SPLE) [CE00, PBvdL05]. SPLE zielt darauf ab, eine Software-Familie, bestehend aus gleichartigen, aber dennoch unterscheidbaren Software-Produkten, auf Basis einer gemeinsamen Kernplattform zu entwerfen und zu entwickeln. Die gemeinsame und die variable Funktionalität der Produkte einer Produktfamilie wird dabei in *Features* gekapselt, welche die für Kunden sichtbaren Konfigurationsoptionen einer SPL darstellen. Somit erlauben Features die Unterscheidung zwischen verschiedenen Software-Produkten einer SPL und die Kommunikation über ihre gemeinsamen und variablen Anteile. Features werden zum Treffen von Design-Entscheidungen verwendet und bilden ein Konzept auf Implementierungsebene, welches Teil der Erstellungsphase einer SPL ist. Zur Realisierung eines Features wird es mit Implementierungsartefakten im Lösungsraum auf verschiedenen Abstraktionsebenen verknüpft. Durch diese Art der Implementierung von Features können Produkte anhand einer Feature-Konfiguration automatisiert abgeleitet werden, indem die mit den ausgewählten Features verknüpften Implementierungsartefakte zu einer entsprechenden Produktimplementierung zusammengesetzt werden.

Zur Realisierung von SPLs definiert SPLE zwei ineinander verzahnte Phasen, das *Domain Engineering* und das *Application Engineering* [CN01]. Das Ziel des Domain Engineerings ist die Identifikation der Features einer SPL und der Festlegung von Restriktionen zwischen diesen Features, welche die Feature-Kombinationen bestimmen, die zu validen Produkten führen. Zur Modellierung der Variabilität von Produktfamilien wurden in der Literatur unterschiedlichste Modellierungsansätze vorgeschlagen [CGR+12]. Ein sehr verbreiteter Modellierungsansatz ist dabei die Verwendung von FODA-Feature-Modellen, welche den validen Konfigurationsraum einer SPL beschreiben [KCH+90]. Feature-Modelle stellen deshalb eine geeignete Basis für die Validierung von SPL-Spezifikationen im Problemraum dar. Beispielsweise können anhand des Feature-Modells einer SPL unerwünschte Anomalien wie Inkonsistenzen zwischen Feature-Restriktionen detektiert werden [BSRC10]. In der Literatur wurden verschiedene Ansätze für die automatisierte und effiziente Analyse von Feature-Modellen vorgeschlagen, die häufig auf einer Übersetzung von (graphischen) Variabilitätsmodellen in entsprechende (logische) Constraint-Solving-Probleme basieren [BMC05, HCH09, MWC09]. Aufbauend auf diesen Constraint-Solving-Problemen können automatisierte Validitätsüberprüfungen für SPL-Spezifikationen durchgeführt werden.

Neben der Variabilität im Raum ist in vielen Anwendungsdomänen ein Trend hin zur Betrachtung von *Variabilität über die Zeit* zu beobachten. Zum Beispiel können sich in der Automatisierungstechnikdomäne Anforderungen an die Funktionalität, an Plattformen oder an Anwendungskontexte einzelner Geräte oder sogar kompletter Produktionsanlagen über die Zeit ändern [CBT+14]. Die Realisierung solcher Adaptionsszenarien erfordert eine kontinuierliche Adaption eines entsprechenden Produktes über seine gesamte Lebensdauer. Somit sind zum Umgang mit solchen Adaptionsszenarien vordefinierte *Rekonfigurationsoptionen* erforderlich. Dies hat zur Folge, dass Rekonfigurationsszenarien bereits während des Domain Engineerings antizipiert werden müssen, sodass eine dynamische Änderung einer Feature-Konfiguration und der zugehörigen Implementierungsartefakte sinnvoll implementiert werden kann.

Die Erweiterung des SPL-Paradigmas um vordefinierte Rekonfigurationsszenarien wird in der Literatur unter dem Begriff *dynamische Software-Produktlinien (DSPL)* zusammengefasst [HHPS08]. Bencomo et al. [BSBG08, BHdA12] waren unter den Ersten, die die Eignung von DSPLs für die präzise Spezifikation von adaptivem Systemverhalten in Form von vordefinierten Rekonfigurationsszenarien erkannt und beschrieben haben. Die variablen Teile einer DSPL werden dabei während der Ableitung eines Produktes nicht in einem einzelnen Schritt, sondern in einem inkrementellen, schrittweisen Prozess selektiert, d. h. in einem stufenweisen Konfigurationsprozess [CHE04]. Die Reihenfolge der Konfigurationsentscheidungen innerhalb eines solchen stufenweisen Konfigurationsprozesses hängt von den Bindungszeiten ab, die jedem Feature zugewiesen werden [HHPS08]. Dabei wird zwischen *statischen* und *dynamischen* Bindungszeiten unterschieden. Features mit statischen Bindungszeiten bilden unveränderliche Konfigurationsentscheidungen ab, die vor der initialen Zusammensetzung der Implementierungsartefakte beim Application Engineering genau einmal getroffen werden. Im Gegensatz dazu sind Features mit dynamischen Bindungszeiten für bereits abgeleitete Produktimple-

mentierungen rekonfigurierbar und ermöglichen so vorgeplante Adaptionen eines Produktes über den gesamten Lebenszyklus. Die Rekonfiguration von Features bewirkt die entsprechende Anpassung der Produktimplementierung an die veränderte Feature-Konfiguration. Auf diese Art und Weise kombinieren DSPLs Variabilität im Raum und Variabilität über die Zeit miteinander in einem einzigen konzeptuellen Framework.

Gerade im Kontext von sicherheits- und missionskritischen Systemen, wie sie beispielsweise in der Automatisierungstechnik vorkommen, sind zuverlässige Spezifikations-, Validierungs- und Verifikationstechniken unverzichtbar, um Korrektheitseigenschaften für jedes mögliche Produkt einer DSPL vor seiner initialen Ableitung sicherzustellen. Durch die Kombination der zwei Variabilitätsdimensionen, Variabilität im Raum und Variabilität über die Zeit, ergeben sich bei der Spezifikation, Validierung und Verifikation jedoch neue Herausforderungen auf die wir im folgenden Abschnitt eingehen werden.

1.1 Spezifikation, Validierung und Verifikation dynamischer Software-Produktlinien

Die Beschreibung der Herausforderungen ist in zwei Abschnitte unterteilt, zum einen in die Spezifikation dynamischer Software-Produktlinien sowie der Validierung dieser Spezifikation und zum anderen in die Verifikation dynamischer Software-Produktlinien durch Testen.

Testen dynamischer Software-Produktlinien

Die Verifikation der Implementierung einer DSPL ist ein wichtiger Teil der DSPL-Entwicklung. In der Industrie ist der meistgenutzte Verifikationsansatz das Testen. Studien zeigen, dass Unternehmen im Durchschnitt rund ein Viertel ihres Budgets, welches für Software-Entwicklung zu Verfügung steht, für Qualitätssicherung ausgeben, davon entfallen durchschnittlich drei viertel alleine auf das Testen der Software [AL13]. Deshalb ist es sinnvoll, möglichst viele Schritte des Testens zu automatisieren. Die Automatisierung der Generierung von Testfällen ist einer der vielversprechendsten und effektivsten Wege, um den Aufwand für das Testen von Software zu reduzieren [BGM13, GLM12, CS13, BCH+04].

Die automatisierte Testgenerierung für DSPLs stellt dabei aufgrund der Kombination der zwei oben genannten Variabilitätsdimensionen eine besondere Herausforderung dar. Die Herausforderungen, die sich bereits aus der Testgenerierung für einzelne Software-Produkte ergeben, werden somit durch die Herausforderungen verstärkt, die sich aus der Testfallgenerierung für variantenreiche Systeme, d. h. SPLs, ergeben und durch die Herausforderungen, die sich aus der Testgenerierung für rekonfigurierbare Systeme ergeben. Im Folgenden wird eine Übersicht über diese Herausforderungen bei der automatisierten Testgenerierung für DSPLs gegeben.

Testgenerierung für einzelne Software-Produkte. Aktuelle Techniken zur automatisierten White-Box-Testgenerierung für Komponententests zielen beispielsweise darauf ab, eine Menge von Testzielen bezüglich eines Abdeckungskri-

teriums abzudecken, die auf Basis des Programm-Codes definiert werden. Eine vielversprechende Technik zur automatisierten Generierung von Testfällen ist Model-Checking. Hierbei wird mit Hilfe eines Model-Checkers ein Testfall aus einem Gegenbeispiel für die Unerreichbarkeit einer Programmeigenschaft (d. h. einem negierten Testziel) abgeleitet [BCH+04, HdMR04]. Zur Abdeckung einer Menge von Testzielen, die anhand eines Abdeckungskriteriums für ein Programm abgeleitet wurden, kann durch wiederholtes Aufrufen eines Model-Checkers für jedes Testziel ein Testfall generiert werden. Für große Programme mit hunderten oder sogar tausenden Zeilen Code ist die Testgenerierung mit Model-Checking jedoch ineffizient, da die Testgenerierung für jedes Testziel immer wieder eine teure Erreichbarkeitsanalyse auf einer Abstraktion des Programm-Codes durchführt, die während der Testgenerierung mehrfach neu berechnet werden muss [HKL+10]. Model-Checker sind eher auf die Falsifizierung einiger weniger Programmeigenschaften hin optimiert, anstatt auf die Behandlung einer großen Menge von Testzielen. Dies führt dazu, dass die Generierung von Test-Suiten durch das wiederholte Aufrufen eines Model-Checkers ineffizient werden kann, da die Erreichbarkeitsanalyse für jedes Testziel erneut durchgeführt wird. Die Erreichbarkeitsanalyse immer wieder erneut durchzuführen, kann für Testziele, die auf den gleichen oder ähnlichen Programmpfaden liegen, zu sehr vielen redundanten Berechnungsschritten führen und sollte dementsprechend vermieden werden. Die Herausforderung bei der Testgenerierung für einzelne Software-Produkte ist es somit, redundante Berechnungsschritte durch die systematische Wiederverwendung von Erreichbarkeitsanalyseergebnissen zwischen Testzielen zu vermeiden.

Variabilität im Raum. Durch die Variabilität im Raum muss die Testgenerierung nicht mehr nur für ein einzelnes Software-Produkt durchgeführt werden, sondern für eine ganze Familie von Software [McG01, PS08, SH11]. Somit erweitert die Testgenerierung für SPLs die Testgenerierung für einzelne Software-Produkte um eine weitere Dimension: dem Variantenreichtum. Da es aufgrund der hohen Anzahl von Software-Produkten, die exponentiell mit der Anzahl der Features einer SPL steigt, üblicherweise unmöglich ist, die Testgenerierung für jedes Software-Produkt separat (produktweise) durchzuführen, wurden in der Literatur Techniken vorgeschlagen, um Testgenerierung, bzw. Model-Checking als Grundlage für automatisierte Testgenerierung, für ganze Software-Familien auf einmal (familienbasiert) durchzuführen [AtBFG11, COLS11, CHSL11, CCP+12, BLB+15]. Bei der familienbasierten Testgenerierung für SPLs sollte der Grundsatz von SPLE, Artefakte während der Entwicklung von SPLs möglichst systematisch zwischen Software-Produkten wiederzuverwenden, beibehalten werden. Für die automatisierte, familienbasierte Testgenerierung ist die Herausforderung somit insbesondere die Wiederverwendung von (Teil-)Ergebnissen von Erreichbarkeitsanalysen (z. B. Testfälle) während der Testgenerierung zwischen Testzielen und Software-Produkten zu maximieren.

Variabilität über die Zeit. DSPLs erweitern SPLs um den Aspekt der Rekonfiguration von Features zur Laufzeit, d. h. um Variabilität über die Zeit, sodass dynamisch zwischen Produktkonfigurationen gewechselt werden kann. Während

der Rekonfiguration einer DSPL von einer Produktkonfiguration zu einer anderen kann es zu Fehlverhalten kommen. Zum Beispiel, wenn sich die DSPL während der Rekonfiguration in einem transienten Zustand befindet, in dem die zur erfolgreichen Rekonfiguration notwendige An- bzw. Abwahl von Features noch nicht vollständig abgeschlossen ist. Weiterhin muss überprüft werden, dass sich eine DSPL nach einer Rekonfiguration gemäß dem Verhalten des Zielproduktes der Rekonfiguration verhält. Somit ist die Herausforderung beim Testen von DSPLs nicht nur das Testen des Verhaltens der einzelnen Produkte, welche die DSPL umfasst, wie es bei SPLs der Fall ist, sondern es muss auch das Rekonfigurationsverhalten von DSPLs getestet werden.

Spezifikation und Validierung dynamischer Software-Produktlinien

Aktuelle Spezifikationsansätze für DSPLs unterscheiden zwischen statischen und dynamischen Features, d. h. Features, die vor der Laufzeit konfiguriert werden, und Features, die zur Laufzeit (re-)konfiguriert werden. Davon ausgehend gibt es in einem stufenweisen (Re-)Konfigurationsprozess für DSPLs zwei Stufen in denen Features konfiguriert werden können: eine statische Stufe gefolgt von einer dynamischen Stufe. Besonders bei sicherheitskritischen Systemen erfordert das Design von zuverlässigen DSPLs jedoch die Möglichkeit präzisere und flexiblere temporale Restriktionen zwischen (Re-)Konfigurationen zu definieren als es die bloße Unterscheidung zwischen statischen und dynamischen Features erlaubt. Aus diesem Grund wird ein stufenweiser Konfigurationsprozess für DSPLs benötigt, der es zum einen erlaubt Features mehrere verschiedene Bindungszeiten zuzuweisen, die jeweils statisch oder dynamisch sein können, sowie temporale Abhängigkeiten und Restriktionen zwischen den Features einer DSPL während des gesamten Lebenszyklus der DSPL präzise zu beschreiben.

Aus diesen Erweiterungen von DSPL-Spezifikationen ergeben sich zusätzliche Arten von Anomalien, welche während der Spezifikation von DSPLs auftreten können. Diese Anomalien können beispielsweise zu einer fehlerhaften Rekonfiguration einer DSPL in einem kritischen Moment führen, wenn die Anomalien nicht rechtzeitig während des Domain Engineering entdeckt werden. Deswegen ist die Validierung der DSPL-Spezifikation insbesondere für sicherheits- und einsatzkritische DSPLs notwendig, um die Korrektheit der Spezifikation des Rekonfigurationsprozesses einer DSPL inklusive seines stufenweisen Konfiguationsprozesses zu zeigen. Zu diesem Zweck wird eine automatisierte Validierungsstrategie für DSPLs benötigt, welche die essentiellen Validitätseigenschaften eines stufenweisen Konfigurationsprozesses mit temporalen Abhängigkeiten und Restriktionen zwischen Features sicherstellt.

Aus den oben beschriebenen Herausforderungen bzgl. der Spezifikation, Validierung und Verifikation von DSPLs ergeben sich eine ganze Reihe wissenschaftlicher Fragestellungen. Im folgenden Abschnitt werden die wissenschaftlichen Fragestellungen beschrieben, die in dieser Arbeit behandelt werden.

1.2 Wissenschaftliche Beiträge

Der Kern dieser Arbeit umfasst Ansätze zur Spezifikation, Validierung und Verifikation von rekonfigurierbaren Software-Systemen, welche als DSPLs umgesetzt sind. Diesbezüglich werden in dieser Arbeit fünf wissenschaftliche Beiträge präsentiert, welche in zwei Bereiche unterteilt sind:

1. Verifikation von DSPLs mittels Software-Testens.
2. Spezifikation von DSPLs und Validierung dieser Spezifikation.

Automatisierte Test-Suite-Generierung für dynamische Software-Produktlinien

Zur Verifikation von DSPLs mit Software-Testen werden in dieser Arbeit automatisiert Test-Suiten erstellt. Ein essentieller Beitrag ist dabei die Erhöhung der Effizienz und der Effektivität der automatisierten Test-Suite-Generierung. Die Effizienz der Test-Suite-Generierung umfasst die zwei Aspekte, Dauer der Test-Suite-Generierung und Größe der generierten Test-Suiten. Im Gegensatz dazu beschreibt die Effektivität der Test-Suite-Generierung, die Fehlerdetektionsraten und die Abdeckungsraten der generierten Test-Suiten. Der Fokus der Test-Suite-Generierung liegt auf einer möglichst hohen Wiederverwendung von Testartefakten und Berechnungsergebnissen. Die wissenschaftlichen Beträge reichen dabei von der Verbesserung der allgemeinen Test-Suite-Generierung für einzelne Software-Produkte, über die Test-Suite-Generierung für ganze Software-Familien bis hin zur Berücksichtigung dynamischer Rekonfigurationsaspekte während der Test-Suite-Generierung, um systematisch rekonfigurierbares Verhalten von Software-Systemen zu testen.

Wissenschaftlicher Beitrag (1) - Effiziente Test-Suite-Generierung für Software-Produkte. Zur Erhöhung der Effizienz der Test-Suite-Generierung für Software-Produkte erweitern wir einen existierenden Test-Suite-Generierungsansatz von Holzer et al. [Hol13, BHTV13]. Der Ansatz von Holzer et al. führt für jedes Testziel mit Hilfe von Model-Checking eine Erreichbarkeitsanalyse zur Testfallableitung durch. Die erste Erweiterung dieses Ansatzes umfasst das Kombinieren der existierenden Test-Suite-Generierung mit Techniken aus dem Multi-Property-Checking, um wiederholte Erreichbarkeitsanalysen für jedes Testziel einzeln zu vermeiden und stattdessen Ergebnisse von Erreichbarkeitsanalysen zwischen Testzielen wiederzuverwenden. Multi-Property-Checking erlaubt es, mehrere Programmeigenschaften (Properties) während einer Erreichbarkeitsanalyse gleichzeitig zu analysieren [ABM+16]. Diese Idee verwenden wir, um für mehrere Testziele gleichzeitig während einer einzigen Erreichbarkeitsanalyse eine Test-Suite zu generieren, die diese Testziele abdeckt. Als zweite Erweiterung verbessern wir die Konstruktion des Programmzustandsraumes, der für die Testfallableitung während der Erreichbarkeitsanalyse benötigt wird. Die Erweiterung besteht aus dem Zusammenfassen von Programmzuständen im Programmzustandsraum, sodass der zu konstruierende Programmzustandsraum verkleinert wird.

Wissenschaftlicher Beitrag (2) - Effiziente Test-Suite-Generierung für Software-Produktlinien. Zum Umgang mit variantenreichen Software-Systemen wie SPLs können familienbasierte Analysetechniken verwendet werden, die nicht jedes Software-Produkt (Variante) einzeln analysieren, sondern eine familienbasierte Darstellung einer SPL verwenden, die sämtliche Software-Produkte in sich vereint [TAK+15]. Diese Idee wurde bereits im Bereich der modellbasierten Test-Suite-Generierung angewendet [COLS11, DPS14, DPL+14]. In dieser Arbeit kombinieren wir die Ideen der familienbasierten Test-Suite-Generierung mit den Techniken, die im wissenschaftlichen Beitrag (1) beschrieben wurden. Das Ziel dabei ist es, Test-Suiten unter der Betrachtung kompletter Software-Familien zu generieren, sodass mit den generierten Test-Suiten sämtliche Software-Produkte einer SPL abgedeckt werden können. Dabei liegt der Fokus insbesondere auf einer möglichst hohen Wiederverwendung von Ergebnissen von Erreichbarkeitsanalysen zwischen Testzielen und Software-Produkten, sodass der Aufwand für die Test-Suite-Generierung im Vergleich zur produktweisen Testableitung möglichst reduziert wird.

Wissenschaftlicher Beitrag (3) - Automatisierte Test-Suite-Generierung für dynamische Software-Produktlinien. Zum hinreichenden Testen rekonfigurierbarer Software-Systeme, die als DSPLs implementiert wurden, muss neben dem funktionalen Verhalten selbst auch das Rekonfigurationsverhalten solcher Software-Systeme getestet werden. Diesbezüglich erweitern wir die aus den wissenschaftlichen Beträgen (1) und (2) resultierenden Techniken zur automatisierten Test-Suite-Generierung um die Anwendung auf rekonfigurierbare Software-Systeme. Der Fokus liegt dabei auf dem Testen der dynamischen Aspekte von Software-Systemen.

Spezifikation und Validierung rekonfigurierbarer Software-Systeme

In dieser Arbeit werden rekonfigurierbare Software-Systeme als DSPLs spezifiziert. Aktuellen DSPL-Spezifikationsansätzen fehlt es an Mitteln, um die Rekonfigurationseigenschaften und -prozesse realer Systeme hinreichend genau beschreiben zu können. Ein Beispiel dafür ist, dass die Möglichkeiten zur Spezifikation temporaler Abhängigkeiten zwischen Konfigurationsparametern während des (Re-)Konfigurationsprozesses einer DSPL bei existierenden Ansätzen stark eingeschränkt sind [MZG03, HHPS08, DS11, HPS12, BHdA12, SLR13]. Deshalb werden diese Ansätze in dieser Arbeit um Konzepte zur detaillierten Spezifikation temporaler Abhängigkeiten zwischen Konfigurationsparametern von DSPLs erweitert. Weiterhin werden Konzepte zur Validierung solcher DSPL-Spezifikationen vorgestellt.

Wissenschaftlicher Beitrag (4) - Modellierungsansatz für rekonfigurierbare Software-Systeme als dynamische Software-Produktlinien. In dieser Arbeit wird ein ganzheitlicher Spezifikationsansatz für DSPLs vorgestellt, welcher existierende DSPL-Spezifikationsansätze erweitert, um Rekonfigurationseigenschaften und -prozesse realer Systeme hinreichend präzise beschreiben zu können. Dies umfasst insbesondere die Möglichkeit der Spezifikation von logischen

und temporalen Abhängigkeiten zwischen Features während eines stufenweisen (Re-)Konfigurationsprozesses einer DSPL und beliebig vielen aufeinanderfolgenden Bindungszeiten.

Wissenschaftlicher Beitrag (5) - Validierung dynamischer Software-Produktlinien. Durch das Betrachten der Kombination logischer und temporaler Abhängigkeiten in einem stufenweisen Konfigurationsprozess mit mehreren Bindungszeiten sowie des Rekonfigurationsverhaltens einer DSPL über ihre gesamte Lebensdauer entstehen neue Anomalietypen in DSPL-Spezifikationen. Ein Beispiel dafür ist eine DSPL-Variante, deren Rekonfiguration aufgrund einer fehlerhaften Spezifikation fehlschlägt. In dieser Arbeit stellen wir Validierungstechniken bereit, um solche Anomalien in DSPL-Spezifikationen zu analysieren. Dafür verwenden wir Constraint-Solving-Techniken. Diese Constraint-Solving-Techniken eignen sich jedoch nur zur Analyse von stufenweisen Konfigurationsprozessen bis zur Konfiguration des ersten initialen Software-Produktes, weil die Constraint-Solving-Techniken zustandslos sind. Somit ist es nicht möglich mit Hilfe der Constraint-Solving-Techniken das Rekonfigurationsverhalten einer DSPL umfassend zu validieren. Deshalb erweitern wir die Validierung von DSPL-Spezifikationen um einen Model-Checking-basierten Ansatz, sodass auch Anomalien bezüglich des Rekonfigurationsverhaltens einer DSPL detektiert werden können. Dies umfasst beispielsweise das Identifizieren von Rekonfigurations-Deadlocks.

1.3 Gliederung der Arbeit

In Kapitel 2 werden ein motivierendes Beispiel aus der Industrie in Form eines hochkonfigurierbaren laufzeitadaptiven medizintechnischen Systems und ein illustrierendes Beispiel vorgestellt, welche zur Veranschaulichung der in dieser Arbeit beschriebenen Konzepte und Techniken verwendet werden. Anhand dieser Beispiele werden weiterhin die zum Verständnis dieser Arbeit benötigten Grundlagen zu den Themen Software-Testen, SPLs und DSPLs vorgestellt. In Kapitel 3 wird ein Ansatz zur effizienten Test-Suite-Generierung für Software-Produkte vorgestellt, welcher in Kapitel 4 um Aspekte der Test-Suite-Generierung für SPLs und in Kapitel 5 um die Berücksichtigung von Rekonfigurationsaspekten zur Test-Suite-Generierung für DSPLs erweitert wird. Weiterhin wird in Kapitel 5 ein ganzheitlicher Ansatz zur Spezifikation und Validierung von DSPLs vorgestellt. Kapitel 6 bildet den Abschluss dieser Arbeit, indem es den Inhalt zusammenfasst.

2

SOFTWARE-TESTEN UND SOFTWARE-PRODUKTLINIEN

In diesem Kapitel werden die in dieser Arbeit zu lösenden Probleme eingeführt und motiviert. Zur Problembeschreibung werden zuerst zwei Beispiele beschrieben, wovon eines ein reales System aus der Automatisierungstechnik ist und das andere eine Menge von vier sehr einfachen C-Funktionen. Danach werden die grundlegende Konzepte und Begriffe der in dieser Arbeit betrachteten Forschungsdomänen erläutert. Begonnen wird dabei mit den allgemeinen Begriffen und Notationen aus der Domäne des Software-Testens, insbesondere des modellbasierten Testens. Weiterhin werden die grundlegenden Ideen und Konzepte von (dynamischen) Software-Produktlinien und Software-Produktlinienentwicklung beschrieben. In jedem Abschnitt werden offene Forschungsherausforderungen zu den jeweiligen Themen herausgearbeitet, welche dann im Haupteil dieser Arbeit behandelt werden.

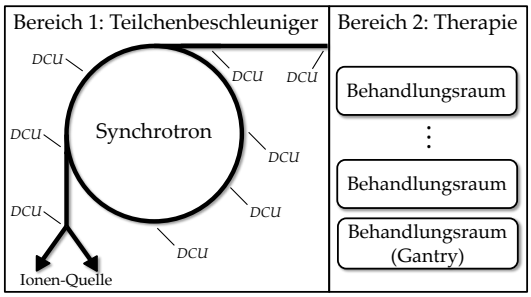
2.1 Motivierende und illustrierende Beispiele

In diesem Kapitel werden zwei Beispiele vorgestellt, die zur Motivation und Illustration der Forschungsherausforderungen und deren Lösungsansätzen verwendet werden. Das erste Beispiel ist ein reales, modellbasiert entwickeltes, medizinisches Gerät aus der Automatisierungstechnik und wird insbesondere zur Motivation und Illustration von Modellierungsansätzen verwendet. Ergänzend dazu besteht das zweite Beispiel aus einer Menge von vier sehr einfachen C-Funktionen, die zur Illustration von Testansätzen dienen, da das erste Beispiel für diesen Zweck viel zu umfangreich ist.

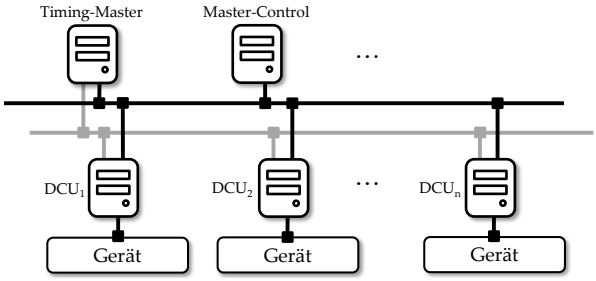
2.1.1 *Fallstudie aus der Automatisierungstechnik: Device Control Unit*

Das erste Fallbeispiel ist ein eingebettetes Software-System zur Steuerung einer Automatisierungsanlage für eine neuartige Strahlentherapie, welches Teil des Heidelberger Ionenstrahl-Therapiezentrums[1] (HIT) ist. Dieses Software-System wurde von der Eckelmann AG[2] entworfen und entwickelt. Die Neuartigkeit des Therapieansatzes ist die Bestrahlung von Tumoren mit Schwerionenstrahlen anstatt Gammastrahlen, welche bei der konventionellen Strahlentherapie verwendet werden. Die Verwendung von Schwerionenstrahlen erlaubt eine genauere Bestrahlung von tiefsitzenden Tumoren durch eine sehr präzise Positionierung des Ionenstrahls,

[1] `http://www.klinikum.uni-heidelberg.de/index.php?id=113005&L=en`
[2] `http://www.eckelmann.de/en`



(a) Hardware-Architektur des HIT



(b) Software-Architektur des HIT-Teilchenbeschleunigers

Abbildung 2.1: Überblick über das HIT und seine Software-Architektur

wodurch der Schaden am tumorumgebenden Gewebe drastisch reduziert werden kann.

Abbildung 2.1 zeigt den Aufbau der Systemarchitektur des Steuerungssystems des HIT, bestehend aus Software- und Hardware-Komponenten [LBHS17]. Das Steuerungssystem umfasst zwei Bereiche: einen *Teilchenbeschleuniger* und einen *Therapiebereich*. Beide Bereiche sind in Abbildung 2.1a dargestellt.

1. Der Teilchenbeschleuniger besteht aus einer Schwerionen-Quelle und dem Synchrotron, welcher aus der Schwerionen-Quelle gespeist wird und die Schwerionen für die Behandlung von Tumoren reguliert.

2. Der Therapiebereich umfasst mehrere unterschiedliche Behandlungsräume. Zum einen gibt es mehrere Räume für die konventionelle Strahlentherapie und zum anderen gibt es die *Gantry*, die eine sehr präzise 360-Grad-Positionierung des Ionenstrahls erlaubt.

Für die Erzeugung und Regulierung des Ionenstrahls werden verschiedene Geräte (z. B. Magnete und E/E-Geräte) verwendet, die um den Synchrotron positioniert sind. Da wir uns in dieser Arbeit auf die Software-Komponenten fokussieren, die für die Automatisierung des Kontrollzyklus des Therapieprozesses verantwortlich sind, werden die Hardware-Komponenten an dieser Stelle nicht weiter beschrieben.

Das Software-System zur Steuerung verschiedener Hardware-Geräte während einer Bestrahlung ist auf mehrere *Device Control Units (DCU)* verteilt, die sich in ihrem Typ unterschieden können. Wie in Abbildung 2.1a zu sehen ist, sind Instanzen der DCU rund um den Synchrotron angeordnet. Zur Erzeugung des Ionenstrahls übernimmt jede DCU-Instanz gemäß ihres Typs eine bestimmte Aufgabe, wie beispielsweise die Kontrolle der Ausrichtungswinkel und Intensität der Magneten, die

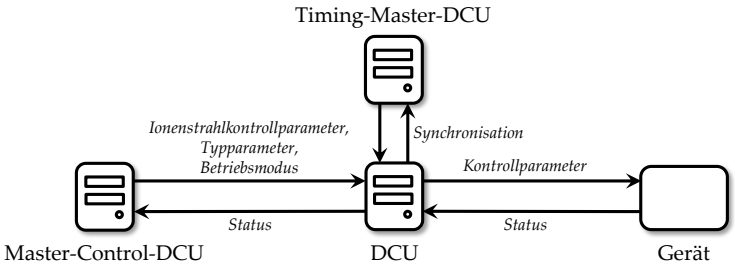


Abbildung 2.2: Kontrollzyklus der DCU

den Ionenstrahl justieren. Der Kontrollzyklus einer DCU-Instanz wird hauptsächlich von zwei Teilsystemen kontrolliert, dem *Timing-Master* und der *Master-Control*, wie in Abbildung 2.1b dargestellt ist. Jede DCU-Instanz ist über ein Echtzeitbussystem integriert, das aus einer hierarchischen Master-Slave-Architektur besteht, in der mit Hilfe des Timing-Masters zeitkritische Aufgaben zwischen den DCU-Instanzen synchronisiert werden. Die Master-Control ist hingegen für die Koordination der Interaktionen zwischen den DCU-Instanzen verantwortlich. Dazu initialisiert die Master-Control die Kontrolle des Ionenstrahls, setzt typspezifische DCU-Parameter, überwacht die DCU-Status und gibt den aktuellen Betriebsmodus der DCU-Instanzen vor.

Abbildung 2.2 zeigt eine Illustration des DCU-Kontrollzyklus [LBHS17]. Vor der Erzeugung des Ionenstrahls setzt die Master-Control für jede DCU-Instanz die zugehörigen DCU-typspezifischen Parameter, die zur Laufzeit einer DCU-Instanz adaptiert werden können. Danach setzt die Master-Control für jede DCU-Instanz weitere Parameter zur Kontrolle des Ionenstrahls, sodass jede DCU-Instanz ihre jeweilige Aufgabe abhängig vom gewählten Therapieprogramm erfüllen kann. Das tatsächliche Verhalten und die Laufzeitadaptivität einer DCU wird durch den aktivierten *Betriebsmodus* sowie den entsprechenden DCU-Parametereinstellungen bestimmt. Zum Beispiel ist im *Experiment*-Modus jeder typspezifische DCU-Parameter frei (re-)konfigurierbar, um eine Konfiguration der DCU-Instanzen für die Behandlung eines bestimmten Patienten zu ermitteln. Im *Therapy*-Modus hingegen folgt jede DCU-Instanz einer strikt vordefinierten Therapieprozedur. Um die Ausführung des gesamten Systems zu unterbrechen, wird es in den *Idle*-Modus versetzt. Aus diesen Beispielen folgt, dass der aktivierte Betriebsmodus einer DCU-Instanz die mögliche Rekonfiguration ihrer Parameter beschränkt.

Die Implementierung der verschiedenen DCU-Typen erfolgte modellbasiert gemäß der europäischen Norm für *Medizinische elektrische Geräte* (EN 60601-1), die unter anderem eine nachweisliche Qualitätssicherung des entwickelten Systems fordert [Fer11]. Jedes Implementierungsmodell eines DCU-Typs wurde bei der Entwicklung von einem Kern-DCU-Modell abgeleitet und jede DCU-Instanz wird vor ihrer Installation im HIT zu einem konkreten Typen vorkonfiguriert. Ein Beispielausschnitt einer Zustandsmaschine, die das Verhalten einer DCU spezifiziert, ist in Abbildung 2.3 dargestellt. Der Ausschnitt zeigt die Initialisierung der Hardware-Boards der DCU. Sind die Hardware-Boards bereits initialisiert, wird der linke Pfad nach der Verzweigung in Zustands *CHECK_HW_BOARDS* ausgeführt und ansonsten der rechte Pfad, der den Zustand *INIT_DEVICE_HW_BOARD_REG*

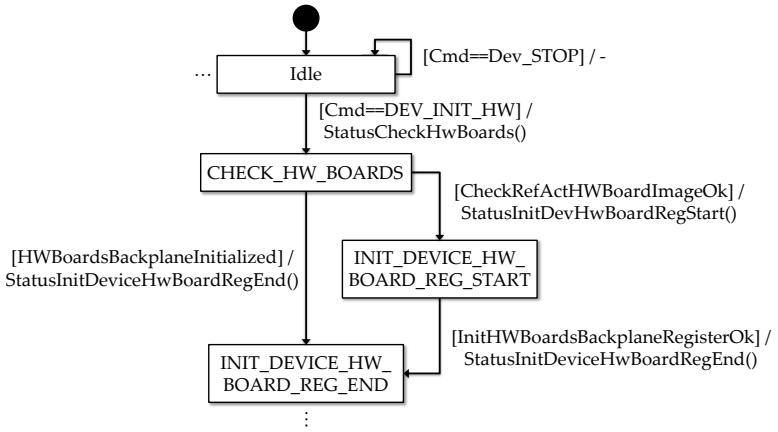


Abbildung 2.3: Ausschnitt der Verhaltenszustandsmaschine der DCU

_START traversiert. Die Transitionen der Zustandsmaschine sind mit einer Bedingung C und Aktionen A nach dem Muster $[C]/A$ beschriftet. Eine Transition darf schalten, wenn die Bedingung C erfüllt ist. Beim Schalten in den Zielzustand der Transition werden die Aktionen A ausgeführt.

Zur Vermeidung von Fehlfunktionen implementiert jeder DCU-Typ wohldefinierte Fehlerbehandlungsprozeduren, welche die gerade ausgeführte Prozedur im Fall von fehlerhaftem Verhalten (z. B. Timeouts) sicher herunterfahren. Zusätzlich zur Implementierung von Fehlerbehandlungsprozeduren muss das Verhalten von sicherheitskritischen Systemen wie der DCU auf sichere Aktionen beschränkt werden können, um mögliche Schädigungen von Patienten im Falle einer Fehlfunktion zu vermeiden. Beispielsweise werden im *Experiment*-Modus alle möglichen Einstellungen für die Bestrahlungstherapie erlaubt. Für die tatsächliche Bestrahlung eines Patienten muss das System allerdings zunächst in den *Therapy*-Modus rekonfiguriert werden, der dann nur noch sichere und auf den Patienten abgestimmte Therapieeinstellungen ermöglicht. Sollte diese Rekonfiguration fehlschlagen, könnte der Patient mit experimentellen Parametereinstellungen bestrahlt werden, was dem Patienten im Ernstfall sogar schaden kann. Deswegen muss das gewünschte Rekonfigurationsverhalten zum einen explizit spezifiziert werden, um solche Fälle auszuschließen, und zum anderen ausführlich getestet werden. Beispielsweise wird gefordert, dass die Rekonfiguration vom *Experiment*-Modus zum *Therapy*-Modus immer über den *Adjustment*-Modus verläuft, damit die DCU-Parameter garantiert für eine sichere Therapie eingestellt werden.

Zusammenfassend ist jede DCU-Instanz Teil eines hoch konfigurierbaren, laufzeitadaptiven und sicherheitskritischen Software-Systems, welches modellbasiert entwickelt wurde. Die Herausforderungen bei der Entwicklung eines solchen Software-Systems sind zum einen die strukturierte und übersichtliche Beschreibung des Systemverhaltens sowie der Konfigurations- und der Rekonfigurationseigenschaften und zum anderen die Verifikation und Validierung des Software-Systems und seiner Spezifikation. Um diese Aspekte wird es im Hauptteil dieser Arbeit gehen.

2.1.2 *Berechnungsfunktionen*

Die Implementierung der DCU ist zur Illustration von Testansätzen für einzelne Software-Komponenten nicht geeignet, da sie keine kompakten, übersichtlichen Code-Beispiele enthält, anhand derer Herausforderungen beim Komponententesten vollständig demonstriert werden können. Deswegen werden wir ergänzend zur DCU vier C-Funktionen zur Illustration von Testansätzen verwenden. Das Listing in Abbildung 2.4a zeigt die Implementierung der C-Funktion GaussianSum. Die Funktion GaussianSum hat den Rückgabetyp *int* und zwei Parameter x und y, die ebenfalls vom Typ *int* sind. Die Funktion ist eine Implementierung der Gaußsche Summenformel und berechnet die Summe der aufeinanderfolgenden natürlichen Zahlen zwischen einschließlich x und y, wenn x kleiner oder gleich y ist (Zeile 4–5) und die Summe der aufeinanderfolgenden natürlichen Zahlen zwischen einschließlich y und x andernfalls (Zeile 7). Die Berechnung findet in einer While-Schleife statt (Zeile 10–13), wobei das Ergebnis der Berechnung in der Variablen r gespeichert wird, deren Wert am Ende der Funktion zurückgegeben wird (Zeile 15). Die Funktion arbeitet auf der Menge $\{x, y, r, n, i\}$ von getypten Programmvariablen, wobei x und y Eingabevariablen sind und r eine Ausgabevariable ist.

Neben der C-Funktion GaussianSum werden wir noch drei weitere C-Funktionen beispielhaft verwenden. Die C-Funktion Factorial aus Abbildung 2.4b berechnet die Fakultät des Werte zwischen einschließlich x und y für positive, natürliche Zahlen. Sie unterscheidet sich zu der Funktion GaussianSum in zwei Punkte: zum einen wird die Variable r mit dem Wert 1 statt mit 0 initialisiert (Zeile 2) und zum anderen wird in der Schleife multipliziert statt addiert (Zeile 11). Die C-Funktionen GaussianSumChk (Abbildung 2.4c) und FactorialChk (Abbildung 2.4d) unterscheiden sich zu den Funktionen GaussianSum und Factorial durch eine zusätzliche Fehlerbehandlungsroutine für die Eingabevariablen x und y. Ist einer der Werte der Variablen x oder y kleiner als 1, wird direkt -1 als Fehlerwert zurückgegeben. Die Fehlerbehandlung der C-Funktionen GaussianSumChk und FactorialChk findet jeweils in den Zeilen 4–6 statt.

2.2 Software-Test

In diesem Abschnitt führen wir Grundlagen und offenen Forschungsherausforderungen des Software-Testens und des modellbasierten Testens ein, die in dieser Arbeit benötigt und behandelt werden. Wir beginnen mit den grundlegenden Notationen und Konzepten des Testprozesses, gefolgt von den Grundlagen des White-Box-Testens und des modellbasierten Testens.

2.2.1 *Allgemeiner Testprozess*

Eine der kritischsten Aktivitäten der Software-Entwicklung ist der Software-Test, der möglichst in jedem Schritt des Software-Entwicklungszyklus angewendet werden sollte. Gemäß des IEEE ist Software-Testen definiert als

```c
int GaussianSum(int x, int y) {
    int i, n, r = 0;

    if (x <= y) {
        i = x; n = y;
    } else {
        i = y; n = x;
    }

    while (i <= n) {
        r = r + i;
        i = i + 1;
    }

    return r;
}
```

(a) Gaußsche Summenformel

```c
int Factorial(int x, int y) {
    int i, n, r = 1;

    if (x <= y) {
        i = x; n = y;
    } else {
        i = y; n = x;
    }

    while (i <= n) {
        r = r * i;
        i = i + 1;
    }

    return r;
}
```

(b) Fakultät

```c
int GaussianSumChk(int x, int y) {
    int i, n, r = 0;

    if (x < 1 || y < 1) {
        return -1;
    }

    if (x <= y) {
        i = x; n = y;
    } else {
        i = y; n = x;
    }

    while (i <= n) {
        r = r + i;
        i = i + 1;
    }

    return r;
}
```

(c) Gaußsche Summenformel mit Eingabeüberprüfung

```c
int FactorialChk(int x, int y) {
    int i, n, r = 1;

    if (x < 1 || y < 1) {
        return -1;
    }

    if (x <= y) {
        i = x; n = y;
    } else {
        i = y; n = x;
    }

    while (i <= n) {
        r = r * i;
        i = i + 1;
    }

    return r;
}
```

(d) Fakultät mit Eingabeüberprüfung

Abbildung 2.4: C-Funktionen zur Berechnung der Gaußschen Summenformel/Fakultät der Werte zwischen x und y mit optionaler Überprüfung auf invalide Eingabewerte

> „The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.“ [IEE90]

Gemäß dieser Definition umfasst Software-Testen jede Aktivität, bei der die zu testenden Software-Instanzen ausgeführt wird, um Qualitätseigenschaften zu überprüfen, die das Software-System erfüllen muss. In der Testliteratur werden diese Qualitätseigenschaften üblicherweise in *funktionale* und *nicht-funktionale* Eigenschaften unterteilt [Som07].

- **Funktionale Eigenschaften** spezifizieren das grundlegende Verhalten eines Systems und seiner Komponenten, d. h. die Systemanforderungen. Somit beschreiben sie, *was* ein System leisten soll.
- **Nicht-funktionale Eigenschaften** beschreiben quantifizierbare Attribute des funktionalen Verhaltens eines Systems. Somit beschreiben sie, *wie gut* ein System seine Funktionalität erfüllt, d. h. bezüglich Performance, Robustheit, Verfügbarkeit, Zuverlässigkeit, etc.

In dieser Arbeit fokussieren wir uns auf die funktionalen Eigenschaften von Software, d. h. wir wollen testen, ob sich ein Software-System gemäß seiner funktionalen Spezifikation verhält.

Im Allgemeinen ist es für nicht-triviale Systeme unmöglich, das gesamte System in einem Schritt zu testen. Deswegen beginnen Tester damit einzelne Komponenten eines Systems zu testen, fahren dann mit dem Testen der Integration der Komponenten fort und testen danach das gesamte System. Hierfür werden die Aktivitäten des Software-Testens in mehrere Phasen unterteilt, die den integrierten Software-Testprozess bilden. Ein typischer Software-Testprozess umfasst vier Phasen [SL12]:

1. **Komponententest:** Ein Software-System wird typischerweise in mehrere Teile unterteilt, die Komponenten oder Units genannt werden. Diese Komponenten werden üblicherweise isoliert voneinander entwickelt, um später zu einem System integriert zu werden. Der Komponententest hat zwei Ziele. Auf der einen Seite zielt der Komponententest darauf ab sicherzustellen, dass sich jede Komponente gemäß ihrer Spezifikation verhält. Auf der anderen Seite ist es das Ziel sicherzustellen, dass sämtlicher geschriebener/generierter Code ausführbar ist. Insbesondere zum Erreichen des ersten Ziels ist es notwendig, dass jede Komponente vor der Integration isoliert gegen ihre Spezifikation getestet wird. Dies eliminiert Einflüsse von anderen Komponenten, die möglicherweise zu Fehlern führen können, und erlaubt eine präzise Lokalisierung von Fehlerursachen. Beispielsweise werden die Netzwerkkommunikation und die Synchronisation zeitkritischer Aufgaben einer DCU-Instanz in eigenen Komponenten entwickelt und getestet.
2. **Integrationstest:** Nachdem die Komponenten entwickelt wurden, werden die Komponenten zu Sub-Systemen zusammengesetzt. Dieser Schritt nennt sich Integration. Integrationstesten zielt darauf ab Fehler in Schnittstellen und den Interaktionen zwischen Komponenten aufzudecken, indem die zusammengesetzten Sub-Systeme getestet werden. Ein Beispiel für den Integrationstest

im Rahmen der Entwicklung DCU ist das Testen, ob bei einer fehlerhaften, zeitlichen Synchronisation zwischen DCU-Instanzen (z. B. durch Timeouts) die richtigen Nachrichten über die Netzwerkkommunikationskomponenten verschickt werden, sodass alle DCU-Instanzen geeignete Fehlerbehandlungsroutinen ausführen können.

3. **Systemtest:** An einem Punkt der Software-Entwicklung werden alle Sub-Systeme zu einem System zusammengesetzt. Systemtesten zielt darauf ab, das gesamte, zusammengesetzte System gegen seine Spezifikation zu testen. Der Systemtest ist wichtig, da beim Systemtest im Gegensatz zu Komponententests und Integrationstests das Testen aus der Perspektive des Kunden ausgeführt wird. Für die DCU werden verschiedene DCU-Instanzen verschiedener Typen zu einem System zusammengesetzt und zusammen getestet.

4. **Akzeptanztest:** Die letzte Phase des Testprozesses ist üblicherweise der Akzeptanztest. Häufig ist der Akzeptanztest die einzige Phase, die den Kunden berücksichtigt und seine Sicht sowie sein Urteil involviert.

Die zu testende Software (z. B. Komponenten oder Systeme) wird im Folgenden als *Instanz unter Test (IUT)* bezeichnet.

Um IUTs ausreichend zu testen, müssen in jeder Phase des Software-Testprozesses entsprechende Testfälle abgeleitet werden. Die Grundlage für die Ableitung und Auswahl von Testfällen ist die Testbasis der entsprechenden Phase, z. B. ein Spezifikationsdokument oder die Struktur des Programm-Codes. Abhängig von der Testbasis können Testansätze in drei verschiedene Kategorien eingeteilt werden [SL12].

- **Black-Box-Test:** Beim Black-Box-Test erfolgt die Auswahl von Testfällen auf Basis der Analyse einer funktionalen (oder nicht-funktionalen) Spezifikation einer Komponente oder des Systems ohne seine interne Struktur (z. B. Implementierungsdetails) zu berücksichtigen. Das Hauptziel von Black-Box-Testen ist das Finden von Fehlern, die fehlende Features in der Implementierung oder eine fehlerhafte Spezifikation aufdecken. Black-Box-Testen wird üblicherweise in den späteren Phasen des Testprozesses angewendet.

- **White-Box-Test:** Beim White-Box-Test wird Wissen über die interne Struktur (die Implementierung) von Komponenten oder das System ausgenutzt, um Testfälle abzuleiten. Während der Testausführung wird der interne Fluss des Testobjektes analysiert, um nicht abgedeckte Testziele zu identifizieren. Dabei sind Testziele Teile der Implementierung, z. B. Zeilen im Programm-Code. White-Box-Testen wird oft während des Komponenten- und Integrationstestens verwendet.

- **Gray-Box-Test:** Gray-Box-Testansätze sind Testansätze, die weder eindeutig Black-Box- noch White-Box-Testansätze sind. Diese Testansätze verwenden zur Ableitung von Testfällen sowohl Wissen über die Spezifikation als auch über die Implementierung. Gray-Box-Testansätze erlauben es Schlüsse über die Beziehung zwischen der Spezifikation und der Implementierung zu ziehen, z. B. ob die Implementierung unspezifiziertes Verhalten enthält.

Der Fokus dieser Arbeit liegt auf dem funktionalen Testen von Komponenten mit Hilfe von White-Box-Testansätzen. Zusätzlich wird demonstriert, wie die für White-Box-Ansätze entwickelten Ideen ebenfalls auf das Black-Box-Testen und somit auf das Systemtesten übertragen werden können.

2.2.2 *White-Box-Testen*

Das Ziel von White-Box-Testen ist es, auf Basis einer Implementierung systematisch *Eingabevektoren* (eine Belegungen für die Eingabevariablen) als Eingabe für die IUT abzuleiten, um die IUT zu stimulieren und dadurch ein bestimmtes Verhalten der IUT zu untersuchen. Häufig wird ebenfalls ein *Testorakel* abgeleitet, d. h. eine Spezifikation der erwarteten Ausgabe der IUT basierend auf dem abgeleiteten Eingabevektor. Beim Komponententest kann dies beispielsweise die erwartete Belegung der Ausgabevariablen in Form eines Ausgabevektors sein. Die Ableitung des Testorakels funktioniert allerdings nur, wenn die Ableitung der Testfälle anhand der Spezifikation der IUT erfolgt und nicht auf der zu testenden Implementierung selbst. In dem Fall, dass sowohl die Ableitung als auch die Ausführung eines Testfalls mit einem Testorakel auf der Implementierung erfolgt, wird ein möglicher Fehler in der Implementierung bei der Ausführung des Testfalls nicht entdeckt, da der Testfall ebenfalls diesen Fehler enthält. Wird ein Testorakel dennoch auf Basis der Implementierung abgeleitet, muss es manuell von einem Tester überprüft werden. Ein *Testfall* besteht aus einem Eingabevektor und einem optionalen Testorakel.

Definition 2.1 (Testfall)**.** Gegeben sei ein Programm mit einer Menge von Programmvariablen V, einer Menge von Eingabevariablen $V_I \subseteq V$ und einer Menge von Ausgabevariablen $V_O \subseteq V$. Ein Testfall $tc = (I, O)$ für dieses Programm besteht aus einem Eingabevektor I, d. h. einer Belegung der Variablen aus V_I, und einem optionalen Ausgabevektor O, d. h. einer Belegung der Variablen aus V_O.

Ein Testfall wird ausgeführt, indem die IUT mit den Werten für die Eingabevariablen aus dem Eingabevektor stimuliert wird. Entspricht die Ausgabe der IUT der erwarteten Ausgabe des Testorakels, war der Test erfolgreich, andernfalls nicht. Besteht der Testfall nur aus einem Eingabevektor und enthält kein vordefiniertes Testorakel für eine automatisierte Überprüfung der Ausgabe der IUT, ist der Tester selbst das Testorakel und muss die Ausgabe manuell auf ihre Richtigkeit hin überprüfen.

Beispiel 2.2 (Testfall)**.** Ein Eingabevektor für die GaussianSum-Funktion aus Abbildung 2.4a, der den If-Zweig (Zeile 5) und einmal die While-Schleife (Zeile 10–13) traversiert ist $[x = 2, y = 2]$. Das zugehörige Testorakel ist $[r = 2]$ und der daraus folgende Testfall, bestehend aus Eingabevektor und Testorakel, $tc_1 = ([x = 2, y = 2], [r = 2])$.

Um die verschiedenen Funktionalitäten eines Programms zu testen, wird üblicherweise eine Menge von Testfällen benötigt. Idealerweise müssten alle möglichen Eingabewerte für die einzelnen Eingabevariablen und deren Kombinationen

getestet werden. In der Praxis ist dies jedoch für Programme mit der sehr großen und theoretisch sogar unbegrenzt großen Wertedomänen der Variablen (in diesem Fall *int*) sowie für reaktive, nicht-terminierende Systeme im Allgemeinen unrealistisch bzw. unmöglich. Deswegen wird ein Abdeckungskriterium für die sinnvolle Auswahl von Testfällen verwendet, die dann zu einer *Test-Suite* zusammengefasst werden.

Definition 2.3 (Test-Suite)**.** Eine Test-Suite $TS = \{tc_1, \ldots, tc_n\}$ ist eine endliche Menge von Testfällen.

Eine Auswahl etablierter Abdeckungskriterien kann in [SL12] gefunden werden. Für White-Box-Testansätze werden üblicherweise Code-Abdeckungsmaße als Abdeckungskriterien verwendet, die eine Menge von syntaktischen Programmelementen als *Testziele* beschreiben, die durch eine Test-Suite abgedeckt werden müssen. Ein Testziel kann beispielsweise eine Anweisung in der Implementierung sein. Ein Testziel abzudecken bedeutet, einen Eingabevektor und einen optionalen Ausgabevektor (d. h. einen Testfall) zu finden, der das Testziel (z. B. eine bestimmte Anweisung) durchläuft. Um ein Abdeckungskriterium vollständig zu erfüllen, wird eine Test-Suite benötigt, die alle erreichbaren, Testziele durch mindestens einen Testfall abdeckt, die durch das Abdeckungskriterium spezifiziert sind.

Beispiel 2.4 (Abdeckungskriterium)**.** Ein Beispiel für ein Abdeckungskriterium ist die *Anweisungsüberdeckung*. Die Anweisungsüberdeckung fordert, dass es für jede Anweisung mindestens einen Testfall in der Test-Suite gibt, der die Anweisung ausführt. Angewendet auf die C-Funktion GaussianSum aus Abbildung 2.4a sind die durch Anweisungsüberdeckung spezifizierten Testziele die Programmzeilen 2, 5, 7, 11, 12 und 15. Der Testfall $tc_1 = ([x = 2, y = 2], [r = 2])$ aus Beispiel 2.2 deckt die Anweisungen im If-Zweig und der While-Schleife ab, sowie die Return-Anweisung, d. h. die Zeilen 2, 5, 11, 12 und 15. Für eine komplette Anweisungsüberdeckung wird ein weiterer Testfall benötigt, der die Anweisung im Else-Zweig (Zeile 7) abdeckt. Ein möglicher Testfall hierfür ist $tc_2 = ([x = 2, y = 1], [r = 3])$. Diese beiden Testfälle bilden eine Test-Suite $TS = \{tc_1, tc_2\}$ für die *vollständige* Anweisungsüberdeckung der GaussianSum-Funktion.

Die Anweisungsüberdeckung ist ein schwaches Abdeckungskriterium, da es zumeist durch Abdeckung weniger Pfade im System erfüllt werden kann. Beispielsweise muss die While-Schleife der C-Funktion GaussianSum nur einmal durchlaufen werden, um die Anweisungsüberdeckung zu erfüllen, d. h. das wiederholte Durchlaufen der While-Schleife muss nicht getestet werden. Für sicherheitsrelevante Systeme reicht die Anweisungsüberdeckung somit nicht aus. In verschiedenen sicherheitsrelevanten Domänen, wie in der Luftfahrt (DO-178B und DO-178C) oder dem Automotive-Bereich (ISO 26262), werden stärkere Abdeckungskriterien wie MC/DC (Modified Condition/Decision Coverage) gefordert [CM94]. Im Gegensatz zur Anweisungsüberdeckung fordert MC/DC die Abdeckung komplexerer Testziele der Form (l, φ), wobei l eine Programm-Location ist und φ ein Prädikat über den Programmvariablen V. Ein Beispiel für ein komplexeres Testziel für die C-Funktion GaussianSum ist $(11, x \leq y)$, welches fordert, dass Programmzeile 11 von

einem Testfall abgedeckt wird, wobei während der Traversierung von Programmzeile 11 die Bedingung $x \leq y$ erfüllt sein muss.

Beim White-Box-Testen werden Testfälle üblicherweise direkt auf der Implementierung der IUT abgeleitet. Dies führt dazu, dass abgeleitete Testorakel manuell überprüft werden müssen. Im Gegensatz dazu werden Testfälle beim modellbasierten Testen auf Basis eines Testmodells abgeleitet, das zusätzlich zur Implementierung angegeben wird und die Spezifikation der IUT umfasst. Durch die Trennung von Implementierung und Testmodell können Testorakel für Testfälle automatisiert, und somit ohne manuelle Überprüfung, aus einem Testmodell abgeleitet werden.

2.2.3 *Modellbasiertes Testen*

In Abschnitt 2.1.1 wurde die DCU-Fallstudie eingeführt, ein modellbasiert entwickeltes Software-System, welches ebenfalls modellbasiert getestet werden kann. Modellbasiertes Testen ist ein Black-Box-Testansatz, der sich zur automatisierten Testgenerierung eignet. Wie in Abschnitt 2.2.1 beschrieben wurde, erfordert automatisiertes Black-Box-Testen ein (Test-)Modell, das als Spezifikation für die IUT dient. Das Testmodell dient unter anderem als Basis für die Testgenerierung, die Testausführung und zur Analyse der Testergebnisse [UL07]. Der Kern von modellbasiertem Testen ist die mehr oder weniger komplette Automatisierung dieser Schritte und somit von Testen im Allgemeinen. Dies ist auch der wichtigste Teil der Definition von modellbasiertem Testen von Utting und Legeard.

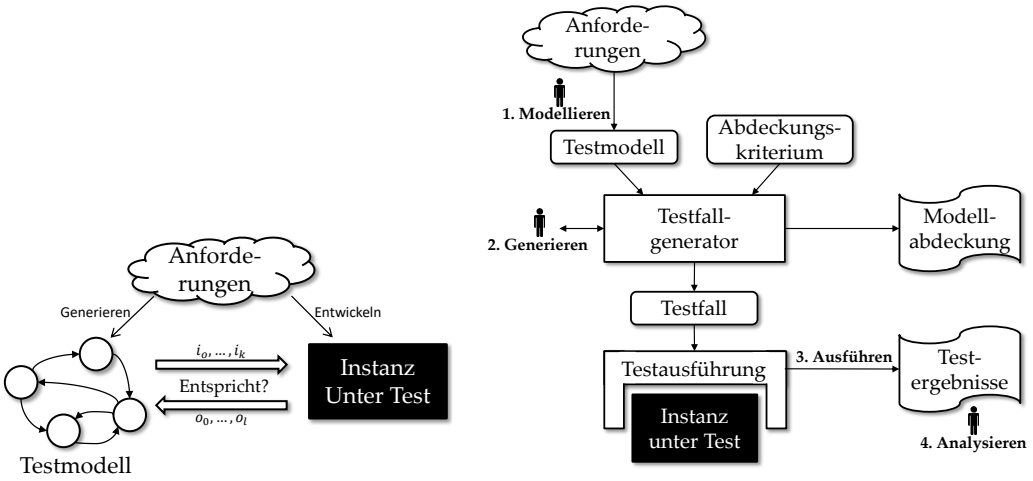
> „Model-based testing is the automation of black box tests.“ [UL07]

Da sich diese Definition auf Black-Box-Testen fokussiert, wird explizit der Bedarf nach Wissen über die tatsächliche Implementierung der IUT (z. B. für das Messen von Code-Abdeckung) ausgeschlossen. Explizit mit eingeschlossen werden hingegen die Automatisierung der Testgenerierung basierend auf einem angemessenen Kriterium, die Testfallausführung mittels einer angemessenen Testumgebung und die automatisierte Analyse der Testergebnisse. Um dies zu realisieren, werden mindestens

- ein Testmodell, das die Verhaltensspezifikation der IUT beschreibt, und
- die tatsächliche Implementierung der IUT

benötigt. Als Testmodell können viele unterschiedliche Modellarten verwendet werden, z. B. Zustandsmaschinen, Aktivitätsdiagramme und Use-Case-Diagramme [UL07]. Ein Beispiel für ein Testmodell ist die Zustandsmaschine aus Abbildung 2.3, die einen Teil des Verhaltens eines DCU-Typs spezifiziert.

Abbildung 2.5a [UL07] zeigt die grundlegende Ausgangslage beim modellbasierten Testen, bestehend aus den Anforderungen an die IUT, einem Testmodell und der Implementierung der IUT als Black-Box. Die Anforderungen werden zum einen als Spezifikation der tatsächlichen Implementierung und zum anderen als Basis für das Testmodell verwendet. Daraus folgt, dass das Testmodell und die Implementierung vollständig voneinander isoliert sind und im besten Fall sogar von verschiedenen Entwicklern erstellt werden. Die IUT ist eine Black-Box mit einem



(a) Ausgangslage **(b)** Modellbasierter Testprozess

Abbildung 2.5: Modellbasiertes Testen

vordefinierten Interface, das seine Ein- und Ausgabeparameter spezifiziert, aber kein internes Wissen enthält, wie beispielsweise Datenstrukturen, Kontrollflüsse oder die Verwendung der Hardware bei der Testausführung. Der tatsächliche Testprozess, der überprüft, ob die IUT der Spezifikation entspricht, die durch das Testmodell gegeben ist, wird wie folgt ausgeführt:

1. Aus dem Testmodell wird eine Sequenz von Eingaben abgeleitet, mit der die IUT stimuliert wird.

2. Die aus der Ausführung resultierende Sequenz von Ausgaben der IUT wird mit den erwarteten Ausgaben verglichen, die ebenfalls im Testmodell kodiert sind.

3. Wenn die Ausgaben der IUT mit den erwarteten Ausgaben übereinstimmen, dann entspricht die Implementierung der Spezifikation und der Test ist erfolgreich, andernfalls nicht.

Die vollständige Trennung von Testmodell und Implementierung ist ein wesentlicher Bestandteil automatisierter Testableitung durch modellbasiertes Testen. Durch die Trennung können wiederholte, versteckte Fehler vermieden werden, die aus der Beziehung der Testartefakte stammen [UL07].

Abbildung 2.5b zeigt die Integration der grundlegenden Konzepte von modellbasiertem Testen in einen ganzheitlichen modellbasierten Testprozess in Anlehnung an Utting und Legeard [UL07]. Das Ziel dieses Prozesses ist die Automatisierung des Designs von Testfällen. Hierfür erzeugt der Test-Designer ein abstraktes Testmodell der IUT, welches als Eingabe für ein automatisiertes Testgenerierungswerkzeug verwendet wird. Durch die Verwendung des Testmodells für die Testgenerierung kann die Design-Zeit von Testfällen im Gegensatz zur manuellen Ableitung von Testfällen reduziert werden und es können, durch die Verwendung verschiedener Abdeckungskriterien, viele verschiedene Test-Suiten generiert werden. Der modellbasierte Testprozess besteht aus vier Schritten:

1. **Modellierung der IUT und/oder ihrer Umgebung:** Als erstes wird ein abstraktes Testmodell der IUT geschrieben, das die zu testenden Anforderungen umfasst. Das abstrakte Testmodell sollte sich auf die Schlüsselaspekte fokussieren, die getestet werden sollen und Details der IUT weglassen. Das Weglassen von Details führt optimalerweise dazu, dass das Testmodell kleiner als die IUT selbst ist.

2. **Generierung von Tests aus dem Testmodell:** Der zweite Schritt ist die Generierung einer Test-Suite aus dem Testmodell durch die Anwendung eines Abdeckungskrtierums, um festzulegen, welche Tests aus dem Testmodell generiert werden sollen. Beispielsweise legt das Abdeckungskriterium fest, dass alle Transitionen des Testmodells durch die generierte Test-Suite abgedeckt werden sollen. Die Ausgabe dieses Schrittes sind Testfälle, die eine Sequenz von Operationen aus dem Testmodell enthalten. Ein Nebenprodukt dieses Schrittes ist ein Abdeckungsbericht, der angibt, wie gut die generierte Test-Suite das im Testmodell spezifizierte Verhalten abdeckt.

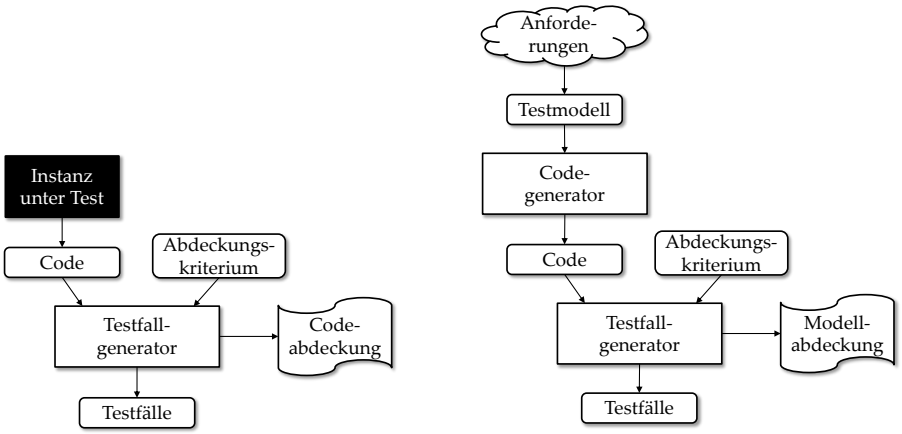
3. **Ausführung der Testfälle auf der IUT:** Der dritte Schritt ist die Ausführung der Testfälle der Test-Suite, wobei das Testausführungswerkzeug die Testausführungsergebnisse protokolliert.

4. **Analyse der Testergebnisse:** Der letzte Schritt ist die Analyse der Testergebnisse. Dies umfasst ebenfalls das Herausfinden, warum Testfälle fehlschlagen und die Behebung der Ursache. Der Grund muss nicht unbedingt an einer fehlerhaften Implementierung liegen, sondern kann auch aus einem inkorrekten Testmodell oder einer fehlerhaften Spezifikation resultieren.

In dieser Arbeit legen wir den Fokus auf Schritt 2, d. h. auf die Automatisierung der Generierung von Testfällen aus einem Testmodell der IUT. Die Schritte 1, 3 und 4 werden explizit nicht berücksichtigt.

Wie bereits in Abschnitt 2.2.1 erwähnt wurde, betrachten wir in dieser Arbeit zwei verschiedene Testgenerierungsszenarien: ein White-Box-Testszenario, das direkt auf der Implementierung der IUT arbeitet und ein modellbasiertes Black-Box-Testszenario. Abbildung 2.6 zeigt die beiden Testgenerierungsszenarien, die in dieser Arbeit betrachtet werden. Das White-Box-Szenario, das in Abbildung 2.6a dargestellt ist, erhält den originalen Programmcode der IUT als Eingabe für die Testfallgenerierung sowie ein Abdeckungskriterium. Die Ausgabe ist eine Test-Suite, die das Abdeckungskriterium erfüllt. Der Black-Box-Ansatz aus Abbildung 2.6b erhält als Eingabe eine Testmodell, das als Basis für die Testfallgenerierung dient. Aus dem Testmodell wird Code generiert, der das Testmodell enthält. Der generierte Code wird dann, wie bei der White-Box-Testgenerierung, als Eingabe für den Testgenerator verwendet. Dadurch können für beide Ansätze die gleichen Techniken verwendet werden.

Beispiel 2.5 (Modellbasierte Testgenerierung)**.** Als Beispiel für modellbasierte Testgenerierung verwenden wir die Zustandsmaschine aus Abbildung 2.3. Die Transitionen sind mit einer Bedingung C und Aktionen A nach dem Muster $[C]/A$ beschriftet. Im ersten Schritt wird aus der Zustandsmaschine Code



(a) White-Box-Szenario **(b)** Black-Box-Szenario

Abbildung 2.6: Testgenerierungsszenarien, die in dieser Arbeit betrachtet werden

generiert. Ein Ausschnitt aus dem Code, der aus der Zustandsmaschine als Eingabe für die White-Box-Testgenerierung erzeugt wird, ist in dem Listing aus Abbildung 2.7 dargestellt. Bei der Code-Generierung wird eine While-Schleife erzeugt (Zeile 2), in der in jedem Durchlauf der aktuelle Zustand der Zustandsmaschine sowie die Bedingungen möglicher Zustandsübergänge abgefragt und Zustandsübergänge ggf. ausgeführt werden. Für jeden Zustand wird ein If-Block erzeugt, der den entsprechenden Zustand behandelt, z. B. in den Zeilen 3–12 für den Zustand *Idle*. In dem If-Block wird zur Behandlung jeder ausgehenden Transition des Zustands ein weiterer If-Block erzeugt. Als Bedingung enthält der Transitions-If-Blocks die Bedingung *C* der Transition und der Körper des If-Blocks besteht aus den Aktionen *A* der Transition. Beispielsweise enthalten die Zeilen 5–8 den Code für die ausgehende Transition *DEV_INIT_HW* des Zustands *Idle*. Wenn die Transition geschaltet werden darf, d. h. das System im *Idle*-Zustand ist und die Bedingung *C* der Transition wahr ist (Zeile 5), werden die Aktionen der Transition ausgeführt, d. h. Zeile 7. Nach der Ausführung der Aktionen wird der aktuelle Zustand auf den Zielzustand der Transition gesetzt, in diesem Fall auf *CHECK_HW_BOARDS* (Zeile 8).

Zur Rückverfolgung der Zustände und Transitionen der Zustandsmaschine im Code, werden C-Labels direkt nach den zugehörigen If-Bedingungen eingefügt. Zum Beispiel das Label *STATE_IDLE* für den *Idle*-Zustand (Zeile 4) und das Label *TRANSITION_DEV_INIT_HW* für die Transition *DEV_INIT_HW* (Zeile 6). Mit Hilfe dieser Labels können Abdeckungskriterien auf Basis der Zustandsmaschine für den C-Code spezifiziert werden. Für Zustandsabdeckung des Ausschnitts der Zustandsmaschine müssen beispielsweise die zu den Zuständen gehörigen Labels

- *STATE_IDLE*,
- *STATE_CHECK_HW_BOARDS*,

```
1  ...
2  while (...) {
3      if (state == Idle) {
4          STATE_IDLE:
5          if (cmd == DEV_INIT_HW) {
6              TRANSITION_DEV_INIT_HW:
7              StatusCheckHwBoards();
8              state = CHECK_HW_BOARDS;
9          } else if (cmd == DEV_STOP) {
10             TRANSITION_DEV_STOP:
11             state = Idle;
12         }
13     } else if (state == CHECK_HW_BOARDS) {
14         STATE_CHECK_HW_BOARDS: ...
15     } else if (state == INIT_DEVICE_HW_BOARD_REG_START) {
16         STATE_INIT_DEVICE_HW_BOARD_REG_START: ...
17     } else if (state == INIT_DEVICE_HW_BOARD_REG_END) {
18         STATE_INIT_DEVICE_HW_BOARD_REG_END: ...
19     }
20 }
21 ...
```

Abbildung 2.7: Aus dem Verhaltensmodell der DCU generierter Code

- *STATE_INIT_DEVICE_HW_BOARD_REG_START* und
- *STATE_INIT_DEVICE_HW_BOARD_REG_END*

im C-Code abgedeckt werden.

Auf Basis des generierten Programm-Codes können Testfälle, wie bei der White-Box-Testgenerierung, erzeugt werden.

Mit Hilfe dieses Vorgehens können Testfälle für modellbasiert entwickelte rekonfigurierbare Systeme wie der DCU generiert werden. Zu diesem Zweck wird aus dem Verhaltensmodell des Systems Code generiert, welcher als Eingabe für die White-Box-Testfallgenerierung verwendet wird, wie am Beispiel der DCU demonstriert wurde (vgl. Beispiel 2.5).

2.2.4 *Offene Forschungsherausforderungen*

Systematisches Testen von Software ist in der Praxis die etablierteste und elaborierteste Qualitätssicherungstechnik, da Testen direkt auf reale Anwendungen auf jeder Abstraktionsstufe angewendet werden kann [UL07]. Ein wichtiger Bestandteil von systematischem Software-Testen ist die Ableitung und Auswahl von geeigneten Testfällen. Zur Minimierung des Aufwands zur Testableitung kann diese automatisiert werden. Die automatisierte Testableitung und -auswahl wird dabei häufig durch Abdeckungskriterien getrieben, die eine Reihe von abzudeckenden Testzielen auf der IUT spezifizieren (vgl. Abschnitt 2.2.2). Abhängig von der Anwendungsdomäne und ob das System sicherheitskritisch oder missionskritisch ist, muss dabei eine bestimmte Abdeckung vom Code gewährleistet werden und es

müssen somit bestimmte Abdeckungskriterien angewendet werden [CM94]. Das der Testfallgenerierung zugrundeliegende Berechnungsproblem auf Basis von Abdeckungskriterien besteht aus einer sehr teuren Analyse des Zustandsraumes der IUT. Ein vielversprechender Ansatz für die Zustandsraumanalyse ist dabei Model-Checking [BCH+04]. Allerdings gibt es hier gerade für größere Systeme wie der DCU Skalierungsprobleme, wenn für jedes Testziel eine neue Analyse des Programmzustandsraumes mit Hilfe von Model-Checking durchgeführt wird. Deshalb sollte die Wiederverwendung von Berechnungsergebnissen zwischen Testzielen zur Verbesserung der Skalierbarkeit der Testgenerierung während der Zustandsraumanalyse maximiert werden. Daraus leiten wir die folgende Forschungsherausforderung ab, die wir im Hauptteil dieser Arbeit betrachten werden:

- **FH1**: Verbesserung der Wiederverwendung von Berechnungsergebnissen der Programmzustandsraumanalyse zwischen Testzielen.

2.3 Grundlagen von Software-Produktlinien

In diesem Abschnitt werden wir grundlegende Begriffe und Konzepte von Software-Produktlinien und Software-Produktlinienentwicklung beschreiben. Weiterhin werden wir auf Qualitätssicherungsstrategien für Software-Produktlinien eingehen und offene Forschungsherausforderungen im Bereich Testen von Software-Produktlinien beschreiben.

2.3.1 *Software-Produktlinien*

Ein Schlüsselprinzip der industriellen Revolution ist die *Massenproduktion*. Massenproduktion erlaubt die Produktion eines Produktes in sehr großen Stückzahlen mit Hilfe von standardisierten, zerlegbaren Prozessen, was zu einer kürzeren Produkteinführungszeit führen kann. Die grundsätzliche Annahme bei der Massenproduktion ist, dass alle Kunden dieselben Anforderungen an ein Produkt stellen und somit keine Anpassung der Produkte an individuelle Kundenwünsche notwendig ist. Ein prominentes Beispiel hierfür ist die Automobilproduktion bis zum Ende des 20. Jahrhunderts [Liu11]. Insbesondere heutzutage erlaubt die Massenproduktion großen Firmen die Produktion von großen Stückzahlen in kurzer Zeit, was zu geringen Kosten und somit zu niedrigen Preisen führt. Zusätzlich können sich Firmen auf ein Produkt oder wenige Produkte während der Produktion und im Verkauf fokussieren.

Heutzutage werden Firmen mit hohem Wettbewerbsdruck und einem stark segmentierten Markt konfrontiert. Zur Erhaltung ihrer Konkurenzfähigkeit müssen ihre Produkte deshalb immer kundenspezifischere Anforderungen erfüllen. Die Produktion kundenspezifischer Produkte kann durch Massenproduktion alleine nicht erreicht werden und muss deshalb zu kundenindividueller Massenproduktion erweitert werden [BMC05, Par76]. Als erste haben Tseng und Jiao kundenindividuelle Massenproduktion definiert als

> „producing goods and services to meet individual customer's needs with near mass production efficiency". [TJ01]

Somit erlaubt kundenindividuelle Massenproduktion die Herstellung ähnlicher Produkte, die dennoch maßgeschneidert an die individuellen Wünsche verschiedener Kunden sind, während die hohe Produktionseffizienz von Massenproduktion (annähernd) beibehalten werden kann. Im Allgemeinen werden diese ähnlichen Produkte als *Produktlinie* bezeichnet.

Ein ähnlicher Verlauf kann im Bereich der Software-Entwicklung beobachtet werden. *Traditionelle Software-Entwicklungsprozesse* legen den Fokus entweder auf die Implementierung einer domänenspezifischen Software für einen bestimmten Kunden oder auf die Entwicklung einer Software, die alle Funktionen enthält und in großer Stückzahl verkauft wird. Als Konsequenz liegt der Fokus bei der traditionellen Software-Entwicklung auf genau einem Software-Produkt unter der Annahme, dass alle Kunden die gleichen Anforderungen teilen. Ähnlich zur Massenproduktion verlangen die Kunden jedoch immer mehr nach Software-Lösungen, die an ihre individuellen Anforderungen und Wünsche anpassbar sind, d. h. kundenindividuelle Massenproduktion in der Software-Entwicklung [PBvdL05].

In der Literatur wird die kundenindividuelle Massenproduktion von Software unter dem Begriff *Software-Produktlinien (SPL)* zusammengefasst [CN01]. Eine SPL umfasst mehrere ähnliche Software-Produkte, die eine gemeinsame Kernfunktionalität haben, sich aber im Detail unterscheiden, sodass die Anforderungen und Wünsche verschiedener Kunden erfüllt werden können [CN01, PBvdL05].

Beispiel 2.6 (Software-Produktlinien)**.** Ein Beispiel für eine SPL ist die DCU. Sie umfasst mehrere ähnliche Software-Produkte, im Sinne von verschiedenen DCU-Typen. Alle DCU-Typen sind von einer gemeinsamen Kern-DCU abgeleitet und teilen somit implizit eine Menge von Kern-Komponenten. Diese Kern-Komponenten bilden die Basis für die automatisierte Ableitung von DCU-Typen, indem die Komponenten zu konkreten DCU-Typen zusammengesetzt werden.

Ein weiteres Beispiel für eine SPL sind die Berechnungsfunktionen aus Abschnitt 2.1.2. Auch sie haben eine gemeinsame Kernfunktionalität, namentlich die Durchführung einer Berechnung eines Wertes auf Basis des kleineren Wertes der Variablen x und y. Jedoch unterscheiden sich die Berechnungsfunktionen im Ziel der Berechnung, da sie entweder die Gaußsche Summenformel oder die Fakultät berechnen und optional eine Fehlerbehandlungsroutine enthalten. Im Folgenden werden wir diese SPL als *Calc-SPL* bezeichnen.

Die variable Funktionalität der Produkte einer SPL wird üblicherweise in *Features* gekapselt, welche die für den Nutzer sichtbaren Konfigurationsparameter einer SPL darstellen. Apel et al. definieren ein Feature wie folgt:

> „A feature is a characteristic or end-user-visible behavior of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle.“ [ABKS13]

Aus dieser Definition können wir zwei grundlegende Eigenschaften von Features identifizieren:

1. Features werden zum Treffen von Design-Entscheidungen verwendet und bilden ein Konzept auf Implementierungsebene, welches Teil der Erstellungsphase einer SPL ist.

2. Features erlauben die Unterscheidung zwischen verschiedenen Software-Produkten einer SPL und die Kommunikation über ihre gemeinsamen und variablen Anteile.

Beispiel 2.7 (Features)**.** Ein hoch konfigurierbares und komplexes Software-System wie die DCU umfasst eine Menge von Features, um an technische Gegebenheiten angepasst werden zu können. Einige Beispiel-Features der DCU und ihre Beschreibung werden in der folgenden Aufzählung beschrieben:

- **Typ-Features:** Die DCU-SPL umfasst mehrere Typ-Features, um den Typ einer DCU festzulegen (z. B. Typ-Feature *T* und Typ-Feature *Z*).
- **ErrorMonEn:** Aktiviert die Fehlerüberwachung.
- **EnableSyncSupplyFrequency:** Aktiviert die Synchronisationsfrequenz.
- **EnableShutdown:** Aktiviert eine vordefinierte Ausschaltprozedur, um einen Betriebsmodus zu verlassen.

Auch die Produkte der Calc-SPL werden durch Features beschrieben. Dies sind:

- **Sum:** Berechnung der Gaußschen Summenformel.
- **Fac:** Berechnung der Fakultät.
- **Chk:** Aktiviert eine Fehlerbehandlungsroutine zur Überprüfung der Eingabewerte.

Features werden mit wiederverwendbaren Implementierungsartefakten in Beziehung gesetzt, die wiederum zu verschiedenen Software-Produkten zusammengesetzt werden können. Somit können Software-Produkte gemäß der individuellen Anforderungen von Kunden durch das Auswählen gewünschter Features konfiguriert und durch das Zusammensetzen der zu den Features gehörigen Implementierungsartefakte instanziiert werden. Daraus folgt, dass SPLs die individuellen Wünsche verschiedener Kunden adressieren und die Mittel bereitstellen, um automatisiert Produkte abzuleiten, die diese individuellen Wünsche erfüllen. Deswegen definieren Clement und Northrop SPL wie folgt:

> „A software product line is a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a precise way.“ [CN01]

Dabei bilden die wiederverwendbaren Implementierungsartefakte die Menge von Kern-Komponenten, die durch Wiederverwendung zu den Produkten einer SPL

zusammengesetzt werden können. Die potentielle Wiederverwendung dieser Implementierungsartefakte zwischen Software-Produkten hat mehrere Vorteile, z. B. eine Steigerung der Implementierungseffizient, eine Verringerung von Entwicklungskosten sowie eine Steigerung der Qualität der entwickelten Software. Um das Potential dieser Vorteile auszuschöpfen, wird jedoch ein strukturierter Software-Entwicklungsprozess benötigt, der die Wiederverwendung von Implementierungsartefakten zwischen Software-Produkten maximiert. Ein Software-Entwicklungsprozess mit dem Ziel der Maximierung der Wiederverwendung von Implementierungsartefakten ist *Software-Produktlinienentwicklung (SPLE)*, der von Clements et al. vorgeschlagen wurde [CN01]. SPLE wird im folgenden Abschnitt vorgestellt.

2.3.2 *Software-Produktlinienentwicklung*

SPLE verschiebt den Fokus der Software-Entwicklung von einem einzelnen System auf eine komplette Familie von ähnlichen Software-Produkten. Die grundlegende Idee dabei ist es, wiederverwendbare Implementierungsartefakte zu entwickeln, die mit Features verknüpft werden, und die Basis für verschiedene Software-Produkte bilden, die eine gemeinsame Menge von Software-Charakteristiken besitzen [CN01, PBvdL05]. Dies erlaubt die Zusammensetzung von wiederverwendbaren Implementierungsartefakten zu Software-Produkten, die individuell an die Wünsche eines Kunden oder technische Aspekte angepasst sind. Zusammenfassend ermöglicht das SPL-Paradigma Entwicklern effizient Software-Produkte von einer gemeinsamen Menge an Kern-Komponenten abzuleiten, anstatt jedes Software-Produkt von Grunde auf isoliert zu entwickeln.

Abbildung 2.8, welche aus [PBvdL05] übernommen wurde, zeigt eine Übersicht über SPLE. SPLE umfasst die zwei Disziplinen: *Domain Engineering* und *Application Engineering* [CN01, PBvdL05, vdLSR07, WL99, ABKS13].

- **Domain Engineering:** Domain Engineering adressiert die Analyse der Domäne einer SPL und die Entwicklung von wiederverwendbaren Implementierungsartefakten, welche die Plattform einer SPL bilden. Während des Domain Engineerings wird zwischen Problemraum und Lösungsraum unterschieden. Im Problemraum wird definiert, welche Produkte eine SPL enthalten soll und somit auch welche Features relevant für die SPL sind. Im Lösungsraum werden Artefakte implementiert, die zur Realisierung der Produkte und Features einer SPL benötigt werden. Das Ziel von Domain Engineering ist nicht die Entwicklung von realen Produkten, sondern von Artefakten, aus denen Produkte abgeleitet werden können. Somit zielt Domain Engineering auf die *Entwicklung für Wiederverwendung* ab.

- **Application Engineering:** Application Engineering adressiert die Entwicklung von spezifischen Software-Produkten, welche die Wünsche und Anforderungen individueller Kunden erfüllen. Dabei wird im Problemraum des Application Engineering evaluiert, wie sich die kundenspezifischen Wünsche auf die Features abbilden lassen, die während des Domain Engineerings definiert wurden. Die konkrete Ableitung der Produkte und somit die Zusammensetzung der entsprechenden Implementierungsartefakte findet dann

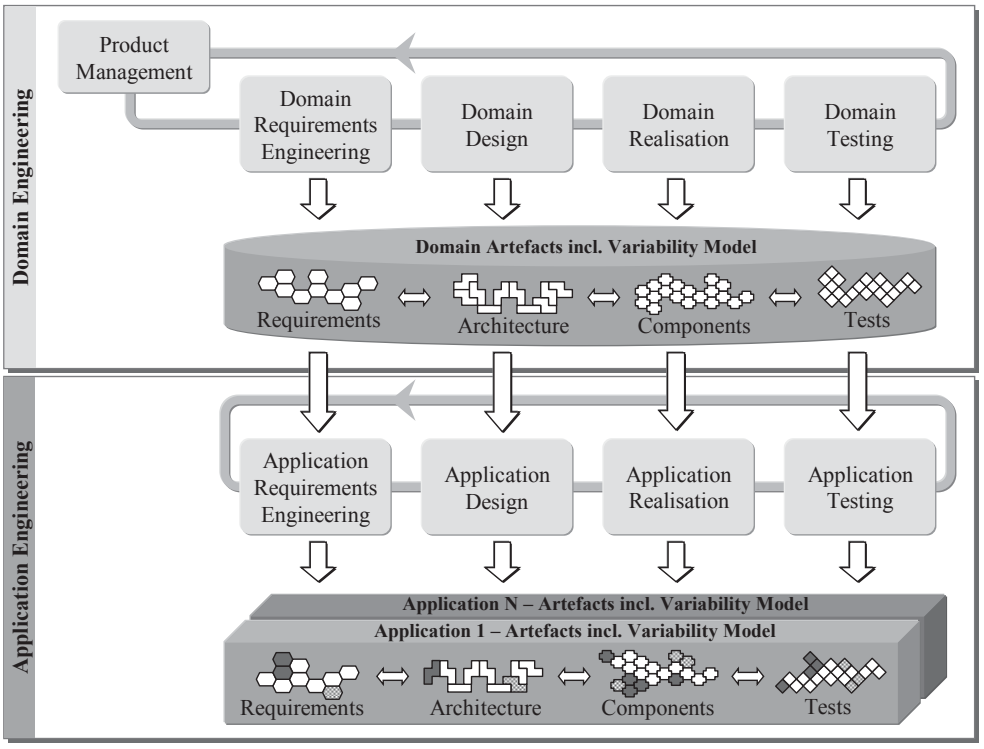


Abbildung 2.8: Software-Produktlinienentwicklung

im Lösungsraum statt. Die Entwicklung folgt dem traditionellen Software-Entwicklungsprozess, wobei der Fokus auf der Wiederverwendung von Implementierungsartefakten liegt, die während des Domain Engineerings entwickelt wurden und mit applikationsspezifischen Artefakten kombiniert werden. Somit ist das Ziel von Application Engineering die *Entwicklung mit Wiederverwendung*.

Die Wiederverwendung in großem Maßstab, wie es bei SPLE praktiziert wird, kann zu einer Kostenreduktion, einer kürzeren Produkteinführungszeit und einer Verbesserung der Qualität von entwickelten Produkten führen [AK09]. Verglichen mit traditionellen Wiederverwendungsansätzen können die Kosten und die Produkteinführungszeit um bis zu 90% reduziert werden [vdLSR07]. Das endgültige Ergebnis der Software-Entwicklung mit SPLE ist eine Menge von Software-Produkten (d. h. eine Software-Familie), die Gemeinsamkeiten teilen sowie produktspezifische Variationen enthalten.

Beispiel 2.8 (Produkte)**.** Die DCU-SPL umfasst verschiedene DCU-Produkte, die unterschiedliche Typen implementieren (vgl. Abschnitt 2.1.1). Jedes DCU-Produkt besitzt, unabhängig von ihrem tatsächlichen Typ, eine Kern-Funktionalität, die durch die Kern-DCU *DCU-0* definiert wurde und von der jedes DCU-Produkt abgeleitet ist. Beispielsweise ist der grundlegende Ablauf des Ausführungszyklus eines DCU-Produktes in bestimmte Phasen unterteilt (z. B. der Hardware-Initialisierung, einem manuellen Service-Modus und der Kontrolle des Ionenstrahls), welche unabhängig vom Typ sind und somit in jeder DCU

vorhanden sind. Dennoch gibt es gewisse Details in jeder Ausführungsphase, die sich für jeden DCU-Typ unterscheiden, sodass jede DCU ihre spezielle Aufgabe ausführen kann. Beispielsweise unterscheidet sich die Implementierung der DCU vom Typ *RB*, welche die Magnete des Synchrotron kontrolliert, von der Implementierung der Timing-Master-DCU (eine DCU vom Typ *Z*).

Die Produkte der Calc-SPL sind die einzelnen Funktionen, die in Abbildung 2.4 dargestellt sind.

Die Produkte einer SPL werden durch ihre Features und durch die Beziehung zwischen den Features beschrieben. Ein bestimmtes Produkt wird durch die An- oder Abwahl von Features definiert, die *Produktkonfiguration* genannt wird. Ein *angewähltes* Feature ist Teil einer Produktkonfiguration und somit des zugehörigen Produktes, während ein *abgewähltes* Feature weder Teil der Produktkonfiguration noch des Produktes ist. Nicht alle Kombinationen von angewählten Features führen zu sinnvollen Produkten. Deshalb werden Restriktionen zwischen Features benötigt, um nicht sinnvolle Produktkonfigurationen zu verbieten [CE00]. Diese Restriktionen zwischen Features werden Feature-Abhängigkeiten genannt und zusammen mit den Features einer SPL in einem Variabilitätsmodell definiert [CE00, GSC+04]. Somit schränken Feature-Abhängigkeiten die möglichen Feature-Kombinationen und somit die Menge der ableitbaren Produkte entsprechend ein. Apel et al. definieren ein Produkt wie folgt:

> A product of a product line is specified by a valid feature selection (a subset of the features of the product line). A feature selection is valid if and only if it fulfills all feature dependencies.

Beispiel 2.9 (Produktkonfiguration)**.** Zu Demonstrationszwecken leiten wir zwei Produkte von unserer DCU-SPL auf Basis der in Beispiel 2.7 beschriebenen Features ab. Produkt P_1 ist eine DCU vom Typ *T*, d. h. eine gepulste DCU mit ansteuerbaren Gatterausgängen, welche die folgende Produktkonfiguration implementiert

- Typ *T*,
- *ErrorMonEn* und
- *EnableShutdown*,

wohingegen Produkt P_2 eine DCU vom Typ *Z*, d. h. eine Timing-Master-DCU, mit den angewählten Features

- Typ *Z*,
- *ErrorMonEn* und
- *EnableSyncSupplyFrequence*,

ist.

Beide Produkte haben das Feature *ErrorMonEn* angewählt und implementieren somit eine Fehlerüberwachung, wohingegen nur P_1 eine vordefinierte Ausschaltprozedur enthält, d. h. das Feature *EnableShutdown* ist angewählt. Da sich die verschiedenen Typ-Features der DCU gegenseitig ausschließen, d. h. jede DCU genau einen Typ implementiert, ist P_1 vom Typ T und P_2 vom Typ Z, aber keine der DCUs implementiert beide Typen.

Ein weiterer Unterschied der Produkte ist, dass nur P_2 das Feature *EnableSyncSupplyFrequence* angewählt hat. Der Grund dafür ist, dass das Feature *EnableSyncSupplyFrequence* Teil der Synchronisation von zeitkritischen Aufgaben ist und somit zur Timing-Master-DCU gehört. Deswegen muss immer, wenn der DCU-Typ Z ausgewählt wurde, auch das Feature *EnableSyncSupplyFrequence* ausgewählt werden. Diese Restriktion verhindert ebenfalls, dass *EnableSyncSupplyFrequence* zusammen mit einem anderen DCU-Typ, wie DCU-Typ T, angewählt wird.

Die Produkte der Calc-SPL implementieren die folgenden Features:

- GaussianSum implementiert das Feature Sum.
- Factorial implementiert das Feature Fac.
- GaussianSumChk implementiert die Features Sum und Chk.
- FactorialChk implementiert die Features Fac und Chk.

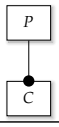
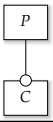
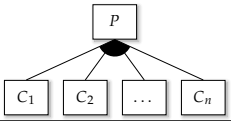
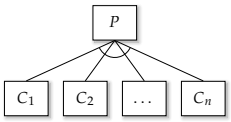
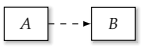
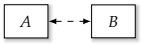
Die Definition von Produkten einer SPL im Sinne von Features und ihren Abhängigkeiten durch Variabilitätsmodelle ist Teil des Domain Engineering. In der Literatur sind *Feature-Modelle* der häufigste Modelltyp für diesen Zweck. Feature-Modelle werden im nächsten Abschnitt eingeführt.

2.3.3 *Feature-Modelle*

Die Variabilität einer SPL kann mit Hilfe von Modellen, wie beispielsweise Entscheidungsmodellen [SRG11], orthogonalen Variabilitätsmodellen [PBvdL05] oder (Domänen)-Feature-Modellen [KCH+90], spezifiziert werden. Ein Vergleich zwischen den populärsten Variabilitätsmodellierungsansätzen wurde von Czarnecki et al. durchgeführt [CGR+12]. In dieser Arbeit fokussieren wir uns auf Feature-Modelle für die Modellierung von Variabilität. Feature-Modelle sind die am häufigsten verwendeten Modelle für Variabilitätsmodellierung [ABKS13].

Feature-Modelle beschreiben die Variabilität einer SPL durch valide Produkte, welche durch unterschiedliche Feature-Kombinationen (Produktkonfigurationen) definiert werden, wobei ein Feature eine Charakteristik oder ein für den Benutzer sichtbares Verhalten eines Systems ist (vgl. Abschnitt 2.3.2). Da nicht alle Kombinationen von Features einer SPL zu sinnvollen oder benutzbaren Produkten führen, bieten Feature-Modelle Konzepte an, um bestimmte Feature-Kombinationen mit Hilfe von Restriktionen zwischen diesen Features zu verbieten. Somit definieren Feature-Modelle die Features einer SPL sowie die Restriktionen zwischen diesen Features. Jede valide Produktkonfiguration einer SPL (d. h. Feature-Kombination, welche den Restriktionen entspricht) führt dann zu einem validen Produkt.

Tabelle 2.1: Abbildung zwischen Feature-Model-Artefakten und Aussagenlogik

Beziehung	Grafische Repr.	Aussagenlogik
Mandatory		$P \leftrightarrow C$
Optional		$C \rightarrow P$
Or		$P \leftrightarrow (C_1 \vee C_2 \vee \ldots \vee C_n)$
Alternative		$(C_1 \leftrightarrow (\neg C_2 \wedge \ldots \wedge \neg C_n \wedge P)) \wedge$ $(C_2 \leftrightarrow (\neg C_1 \wedge \ldots \wedge \neg C_n \wedge P)) \wedge$ $\ldots$ $(C_n \leftrightarrow (\neg C_1 \wedge \neg C_2 \wedge \ldots \wedge \neg C_{n-1} \wedge P))$
Require		$A \rightarrow B$
Exclude		$\neg(A \wedge B)$

Zusammen mit der Feature-orientierten Domänenanalyse haben Kang et al. *Feature-Diagramme* eingeführt, welche eine oft verwendete grafische Repräsentation von Feature-Modellen sind [KCH+90]. Feature-Diagramme werden üblicherweise als hierarchische Bäume dargestellt, welche Features in einer Eltern-Kind-Beziehung sowie Abhängigkeiten und Restriktionen zwischen Features visualisieren. Ihre formale Semantik ist durch eine Übersetzung in Aussagenlogik definiert [Bat05].

In Tabelle 2.1 werden die in der Literatur etablierten Konstrukte zur Spezifikation von Restriktionen zwischen den Features eines Feature-Modells durch ihre grafische Repräsentation in Feature-Diagrammen und durch ihre Semantik als aussagenlogische Terme dargestellt. Wir unterscheiden zwischen Feature-Modalitäten (*mandatory* und *optional* Kind-Features), Feature-Gruppen (*Or*-Gruppen und *Alternative*-Gruppen) und Constraints (*Require*-Constraints und *Exclude*-Constraints).

Beispiel 2.10 (Feature-Modell)**.** Ein Ausschnitt aus dem Feature-Modell der DCU, der alle etablierten Feature-Beziehungen aus der SPL-Literatur enthält, wird in Abbildung 2.9 dargestellt. Das DCU-Feature-Modell enthält 47 Features und fünf Constraints.

Das Feature-Modell der Calc-SPL ist in Abbildung 2.10 dargestellt. Es enthält vier Features. Die Features Sum und Fac befinden sich in einer Alternative-Gruppe, da sie sich gegenseitig ausschließen und das Feature Chk ist optional.

Im Folgenden werden wir die klassischen Feature-Beziehung mit Hilfe des DCU-Feature-Modells erläutern. Es wird zwischen den zwei Feature-Modalitäten mandatory und optional unterschieden.

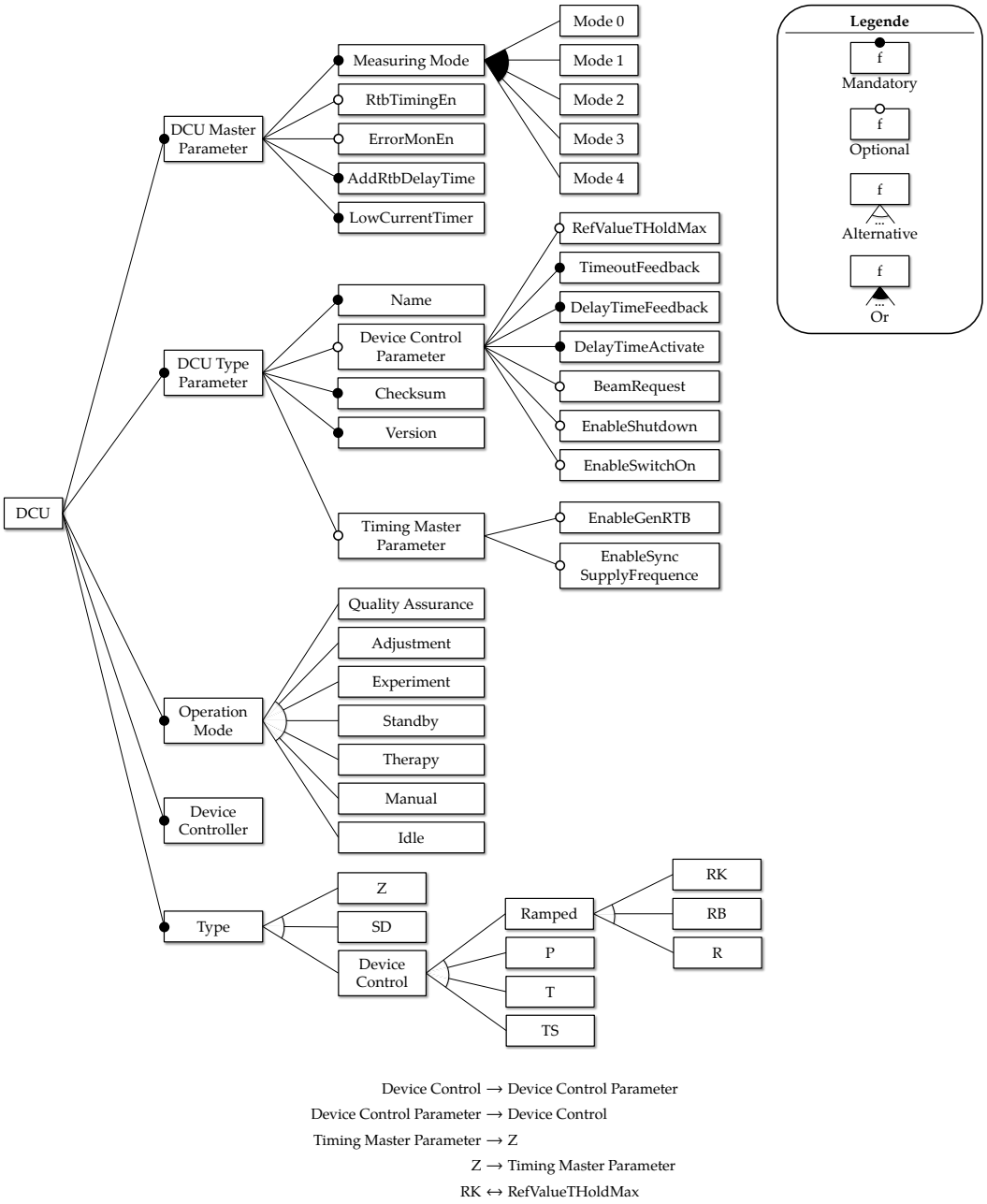


Abbildung 2.9: Ausschnitt aus dem DCU-Feature-Modell

- **Mandatory Features**: Ein mandatory Feature ist immer Teil einer Produktkonfiguration, wenn sein Eltern-Feature angewählt ist und umgekehrt. Somit gilt, wenn ein Eltern-Feature angewählt ist, müssen auch seine mandatory Kind-Features angewählt werden und wenn ein mandatory Kind-Feature angewählt ist, dann muss auch sein Eltern-Feature angewählt werden.
- **Optional Features**: Ein optional Feature kann Teil einer Produktkonfiguration sein, wenn sein Eltern-Feature angewählt ist. Daraus folgt, dass ein Eltern-Feature angewählt werden muss, wenn sein optional Kind-Feature angewählt ist.

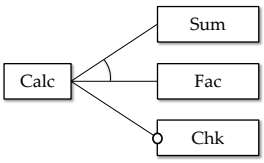


Abbildung 2.10: Feature-Modell der Calc-SPL

Beispiel 2.11 (Feature-Modalitäten)**.** Jede DCU muss einen spezifischen Typ implementieren. Somit ist *Type* ein mandatory Kern-Feature und muss in jeder Produktkonfiguration angewählt sein. Da nicht jede DCU vom Typ *Z* (d. h. eine Timing-Master-DCU) ist, dürfen Parameter die zur Timing-Master-DCU gehören nicht Teil einer DCU eines anderen Typs (z. B. DCUs vom Typ *T* oder *R*) sein. Aus diesem Grund ist das Feature *Timing Master Parameter* optional und muss für jede DCU vom Typ *Z* angewählt werden und für DCUs eines anderen Typs abgewählt werden.

Für logisch benachbarte Features ist es sinnvoll, sie zu gruppieren. Hierfür wurden Feature-Gruppen eingeführt. Wir unterscheiden zwischen Or-Gruppen und Alternative-Gruppen.

- **Or-Gruppen**: Eine Or-Gruppe erfordert, dass *mindestens eines* seiner Mitglieder angewählt wird, wenn das Eltern-Feature der Or-Gruppe angewählt ist. Somit gilt, wenn das Eltern-Feature der Or-Gruppe angewählt ist, dann muss mindestens ein Kind-Feature angewählt sein und wenn mindestens ein Kind-Feature angewählt ist, muss das Eltern-Feature der Or-Gruppe angewählt werden.
- **Alternative-Gruppen**: Eine Alternative-Gruppe erfordert, dass *genau ein* Mitglied der Gruppe angewählt werden muss, wenn das Eltern-Feature der Alternative-Gruppe angewählt ist.

Beispiel 2.12 (Feature-Gruppen)**.** Für jede DCU muss ein *Messungsmodus* (engl. Measuring Mode) ausgewählt werden. Die spezifischen Messungsmodi *Mode 0* bis *Mode 4* sind in einer Or-Gruppe zusammengefasst. Somit muss mindestens ein spezifischer Messungsmodus in jeder DCU angewählt werden, aber es sind auch Kombinationen mehrere Messungsmodi erlaubt. Im Gegensatz dazu ist jede DCU von genau einem Typ, aber implementiert nicht mehrere Typen. Solche Restriktionen können durch eine Alternative-Gruppe modelliert werden. Somit sind unter dem Feature *Type* die spezifischen Typ-Features (z. B., *Z* und *T*) in Alternative-Gruppen zusammengefasst. Diese Gruppen sind wieder in mehreren hierarchischen Alternative-Gruppen angeordnet, sodass es für verschiedene Stakeholder einfacher ist, die Struktur der DCU zu verstehen.

Feature-Modalitäten und Feature-Gruppen sind noch nicht ausdruckmächtig genug, um die inhärente Variabilität einer SPL zu spezifizieren. In einigen Fällen wollen wir verschiedene Features aus unterschiedlichen Teilen des Feature-Baums und unterschiedlicher Feature-Gruppen miteinander in Beziehung setzen. Zu diesem Zweck werden binäre Require- und Exclude-Constraints verwendet.

- **Require-Constraints**: Wenn ein Feature A die Anwahl eines Features B erfordert, dann muss B in jeder Produktkonfiguration angewählt werden, in der A angewählt ist.
- **Exclude-Constraints**: Wenn ein Feature A ein Feature B ausschließt, dann können A und B nicht Teil derselben Produktkonfiguration sein.

Beispiel 2.13 (Constraints)**.** DCUs vom Typ Z erfordern die Konfiguration der Timing-Master-Parameter und umgekehrt. Somit gibt es zwischen den Features Z und *Timing Master Parameter* zwei Require-Constraints, jeweils eine in jede Richtung. Die Kombination des Require-Constraints und der Alternative-Gruppe, die das Feature Z enthält, impliziert ebenfalls, dass Timing-Master-Parameter nicht für eine DCU eines anderen Typs als Z konfiguriert werden können. Daraus folgt, dass Constraints zwischen Features, wie diese impliziten Exclude-Constraints zwischen dem Feature *Timing Master Parameter* und anderen DCU-Typen als Z, nicht immer für den Stakeholder sichtbar sind. Ein explizites Exclude-Constraint gibt es zwischen dem DCU-Typ *RK* und dem Feature *RefValueTHoldMax*. Beide dürfen nicht zusammen in eine Konfiguration gewählt werden und schließen sich somit durch ein Exclude-Constraint aus.

Die Semantik von Feature-Modellen gibt die Menge der validen Produktkonfigurationen an, die durch das Feature-Modell beschreiben werden. Somit ist die Konfigurationsraumsemantik von Feature-Modellen wie folgt definiert.

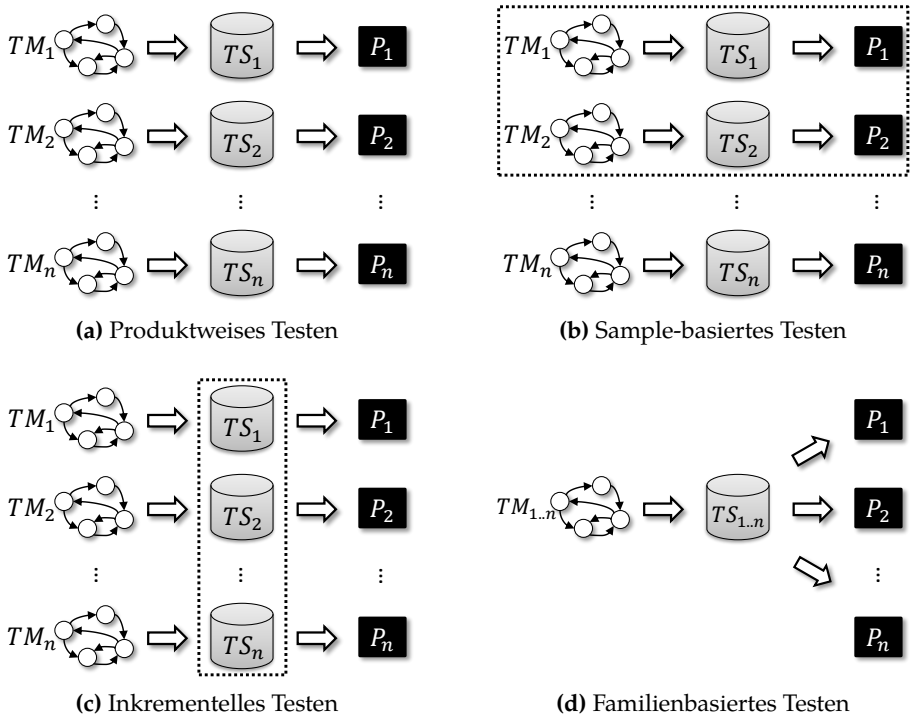
Definition 2.14 (Konfigurationsraumsemantik)**.** Sei FM ein Feature-Modell über der Feature-Menge F. Die Konfigurationsraumsemantik von FM wird angegeben als

$$[\![FM]\!]_{pc} \subseteq 2^F.$$

Eine Teilmenge von Features $pc \subseteq F$ ist eine valide Produkt-Konfiguration gdw. $pc \in [\![FM]\!]_{pc}$ gilt.

2.3.4 *Testen von Software-Produktlinien*

Wie bereits in Abschnitt 2.2 herausgearbeitet wurde, ist die Qualitätssicherung einer der kritischsten Schritte beim Entwickeln konventioneller Software. SPLE fügt noch eine weitere Dimension der Qualitätssicherung von Software-Systemen hinzu, den Variantenreichtum. Beim Entwickeln kundenindividueller Software wie SPLs muss nicht nur die Qualität eines einzelnen Software-Produktes sichergestellt werden, sondern die Qualität einer ganzen Software-Familie. Um mit dieser Komplexität umzugehen, die aus dem Variantenreichtum resultiert, wurden mehrere (formale) Qualitätssicherungsansätze vorgeschlagen, die in verschiedenen Surveys und Studien zusammengetragen wurden [TAK+15, BTL+15], speziell im Bereich des Softwaretestens [ER11, OWES11, dMSNdCMM+11, LKL12, JHF11]. Die grundsätzliche Idee hinter all diesen Ansätzen ist die Reduktion des Qualitätssicherungsaufwandes durch das Anwenden der Wiederverwendungsprinzipien von SPLE auf Testartefakte.



(a) Produktweises Testen

(b) Sample-basiertes Testen

(c) Inkrementelles Testen

(d) Familienbasiertes Testen

Abbildung 2.11: Übersicht über die verschieden Kategorien von SPL-Teststrategien

Abbildung 2.11 zeigt vier Kategorien aus [Loc13], die typischerweise zur Unterteilung von SPL-Testansätzen verwendet werden: produktweises Testen, Sample-basiertes Testen, inkrementelles Testen und familienbasiertes Testen.

- **Produktweises Testen:** Die grundlegende Idee von produktweisem Testen ist es, konventionelles Testen auf jedes einzelne Produkt einer SPL anzuwenden, wie in Abbildung 2.11a zu sehen ist. Das bedeutet, es wird jedes Software-Produkt einer SPL instanziiert und eines nach dem anderen gegen seine Spezifikation getestet. Somit gibt es kein Potential für die Wiederverwendung von Ergebnissen zwischen ähnlichen Produkten. Nichtsdestotrotz nutzen einige produktweise Ansätze die Vorteile der Wiederverwendungsprinzipien von SPLE durch das Verwenden von wiederverwendbaren Testmodellen, anstatt für jedes Produkt ein neues Testmodell zu entwickeln. Hierfür wird das Testmodell mit den Features der SPL annotiert, um individuelle Testmodelle für jedes Produkt gemäß der zugehörigen Feature-Auswahl abzuleiten. Grundsätzlich ist jedoch ein kompletter Test aller Produkte einer SPL im Allgemeinen, aufgrund der hohen Anzahl von Produkten, die exponentiell zur Anzahl der Features steigt, unmöglich.

- **Sample-basiertes Testen:** Um das vollständige Testen aller Produkte wie beim produktweisen Testen zu vermeiden, testen Sample-basierte Ansätze nur eine repräsentative Auswahl von Produkten einer SPL. Beispielsweise werden in der schemenhaften Darstellung aus Abbildung 2.11b nur die Produkte getestet, die von dem gestrichelten Rahmen umfasst werden. Die Idee ist es eine getroffene Auswahl von Produkten umfassend zu testen und von den

Testergebnissen auf die Korrektheit der anderen Produkte zu schließen. Der Vorteil dieser Strategie ist, dass der Testaufwand massiv reduziert werden kann. Die Nachteile sind jedoch eine Menge ungetesteter Produkte, bei denen Fehler übersehen werden können, und dass die Qualität der Tests stark von der Auswahl der getesteten Produkte abhängt. Die Auswahlstrategien für eine angemessene Menge von Produkten wurde stark von den Auswahlstrategien beim kombinatorischen I/O-Testen inspiriert [SL12], wobei Sample-basiertes Testen erfordert, dass bestimmte Feature-Kombinationen durch eine Auswahl von Produktkonfigurationen abgedeckt werden [Gus07, OZML11]. Etablierte Auswahlstrategien sind unter anderem paarweise Feature-Abdeckung, bei der alle Feature-Kombinationen von jeweils zwei Features in mindestens einer Produktkonfiguration vorhanden sein müssen, wie von Oster et al. vorgeschlagen [OMR10], und T-weise Feature-Abdeckung, bei der alle Feature-Kombinationen bis zur Anzahl von T in mindestens einer Produktkonfiguration vorhanden sein müssen, wie von Perrouin et al. vorgeschlagen wurde [PSK+10].

- **Inkrementelles Testen:** Abbildung 2.11c zeigt inkrementelles SPL-Testen. Die Idee des inkrementellen SPL-Testens ist es zu testen, ob die Unterschiede zwischen zwei Produkten beabsichtigt sind. Hierfür werden die Ideen vom Regressionstesten auf SPL-Testen adaptiert. Regressionstesten zielt auf eine Reduzierung des Testaufwandes für neue Revisionen einer Software durch die Wiederverwendung von Test-Suiten für frühere Revisionen ab, indem Testfälle für neue Software-Teile hinzugefügt und obsolete Testfälle für entfernte Software-Teile gelöscht werden [ER10]. Basierend auf dieser Idee sortiert inkrementelles SPL-Testen die Produkte einer SPL, beginnt dann mit dem Testen des ersten Produktes und testet für jedes folgende Produkt nur noch das Delta zu seinem vorherigen Produkt [LSKL12].

- **Familienbasiertes Testen:** Familienbasierte Testansätze zielen auf eine Wiederverwendung von so vielen Testartefakten wie möglich ab, wobei sie die komplette SPL-Familie (d. h. alle Produkte der SPL) zusammen betrachten, wie in Abbildung 2.11d zu sehen ist. Hierfür wird ein wiederverwendbares Testmodell erstellt, das die Spezifikation aller Produkte der SPL enthält. Das wiederverwendbare Testmodell ist über den Features der SPL annotiert, um die Modellelemente auf die zugehörigen Produkte abzubilden. Aus dem annotierten Testmodell werden wiederverwendbare Testfälle abgeleitet, welche ebenfalls mit Features annotiert sind, um die Testfälle den verschiedenen Produkten zuzuordnen. Diese Testfälle können auf den unterschiedlichen Produkten ausgeführt werden, was zu einer hohen Wiederverwendung von Testfällen zwischen Produkten führt und den Testaufwand im Allgemeinen reduziert. Der Nachteil des familienbasierten Testens ist, dass die Analyse von allen Produkten zusammen im schlimmsten Fall aufwändiger sein kann als die Analyse aller Produkte nacheinander.

Ein Ziel dieser Arbeit ist es, einen effizienten Testansatz zu entwickeln, mit dem alle Produkte einer SPL getestet werden können. Deswegen entwickeln wir in dieser Arbeit einen (modellbasierten) familienbasierten Testansatz in dem Sinne, dass wir

```
1  int Calc_SPL(int x, int y) {
2      int i, n, r;
3
4  #if SUM
5      r = 0;
6  #elif FAC
7      r = 1;
8  #endif
9
10 #if CHK
11     if (x < 1 || y < 1) {
12         return -1;
13     }
14 #endif
15
16     if (x <= y) {
17         i = x; n = y;
18     } else{
19         i = y; n = x;
20     }
21
22     while (i <= n) {
23 #if SUM
24         r = r + i;
25 #elif FAC
26         r = r * i;
27 #endif
28         i = i + 1;
29     }
30
31     return r;
32 }
```

Abbildung 2.12: SPL-Implementierung der Calc-SPL

ein annotiertes Testmodell oder annotierten C-Code für die Testgenerierung verwenden, das/der alle Produkte einer SPL enthält und eine sehr hohe systematische Wiederverwendung von Testartefakten im Lösungsraum erlaubt. Solche annotierten Testmodelle bzw. solchen annotierten C-Code werden wir im Folgenden mit *150%-Testmodell* bzw. *150%-C-Code* oder als *SPL-Implementierung* bezeichnen. Eine etablierte Möglichkeit zur Annotation von Feature-spezifischen Implementierungsartefakten auf C-Code-Ebene sind Präprozessordirektiven [KGR+11].

Beispiel 2.15 (150%-Code)**.** In Abbildung 2.4 sind die vier Produkte der Calc-SPL dargestellt. Abbildung 2.12 zeigt eine 150%-Darstellung dieser vier Produkte. Variable Code-Teile, die nur in einigen Produkten enthalten sind, werden durch #if-Bedingungen über (Boole'sche) Feature-Parameter annotiert. Diese Code-Teile sind somit nur in einem Produkt enthalten, wenn die #if-Bedingungen wahr ist, d. h. das Vorhandensein dieser Code-Teile ist abhängig von der Belegung der Feature-Parameter, die durch die Produktkonfiguration eines Produktes festgelegt ist. Beispielsweise sind die Zeilen 11–13 nur in einem

Produkt enthalten, wenn das optionale Feature Chk ausgewählt wurde. Die Zeilen 5 und 24 sind nur in einem Produkt enthalten, wenn das Feature Sum ausgewählt wurde und die Zeilen 7 und 26 sind nur enthalten, wenn das Feature Fac ausgewählt wurde. Aus der 150%-Darstellung können somit alle Produkte aus Abbildung 2.4 mit Hilfe der entsprechenden Feature-Belegungen abgeleitet werden.

Durch Verwendung einer 150%-Darstellung kann aus einer einzigen Implementierung eine Test-Suite abgeleitet werden, die zum Testen aller Produkte einer SPL verwendet werden kann. Dafür wird aus der 150%-Darstellung jedes Produkt nacheinander abgeleitet und es werden für jedes Produkt auf die gleiche Art und Weise Testfälle erstellt, wie es in Abschnitt 2.2.2 beschrieben wurde.

Beispiel 2.16 (Produkteweise Testableitung)**.** Als Basis für die produktweise Testableitung wird die 150%-Implementierung der Calc-SPL aus Abbildung 2.12 verwendet. Das Testziel g für das eine Test-Suite erstellt werden soll, ist Zeile 28 der Calc-SPL ($i = i + 1$). Zuerst wird durch eine entsprechende Feature-Belegung die Funktion GaussianSum aus der 150%-Darstellung abgeleitet, die das Feature Sum implementiert. Beispielsweise deckt der Testfall $tc_1 = ([x = 2, x = 3], [r = 5])$ das Testziel g in der Funktion GaussianSum ab. Als nächstes wird die Funktion GaussianSumChk anhand der Feature-Belegung {Sum, Chk} abgeleitet. Es wird der Testfall $tc_2 = ([x = 2, x = 3], [r = 5])$ abgeleitet, der das Testziel g abdeckt, d. h. es wird wieder der gleiche Testfall abgeleitet, der das Testziel g auf der GaussianSum-Funktion abdeckt. Für die Funktionen Factorial und FactorialChk werden auf die gleiche Art und Weise die Testfälle $tc_3 = ([x = 2, x = 3], [r = 6])$ und $tc_4 = ([x = 2, x = 3], [r = 6])$ abgeleitet. Anzumerken ist, dass der Eingabevektor aller Testfälle der gleiche ist. Weiterhin unterscheidet sich die Ausgabe der C-Funktion je nachdem, welche Features ausgewählt wurden. Wenn das Feature Sum ausgewählt wurde, ist die Ausgabe $[r = 5]$, wohingegen die Ausgabe $[r = 6]$ ist, wenn das Feature Fac ausgewählt wurde. Die Test-Suite $TS = \{tc_1, tc_2, tc_3, tc_4\}$ deckt das Testziel g auf allen Produkten der Calc-SPL ab.

Wie in Beispiel 2.16 zu sehen ist, muss für SPL-Implementierungen die Notation von Code-Abdeckung angepasst werden. Eine vollständige Abdeckung eines Testziel einer SPL-Implementierung ist erst dann erreicht, wenn es eine Test-Suite gibt, die das Testziel in jedem Produkt, in dem es erreichbar ist, durch einen Testfall abdeckt. Bei der produkteweise Ableitung einer Test-Suite, die Testziele auf einer SPL-Implementierung abdeckt, werden viele redundante Ableitungsschritte ausgeführt und es werden viele redundante Testfälle erzeugt. Um dies zu vermeiden eignen sich familienbasierte Testansätze, die Testfälle direkt aus der SPL-Implementierung ableiten und so die Wiederverwendung von Berechnungsergebnissen und Testfällen zwischen Produkten verbessern.

2.3.5 *Offene Forschungsherausforderungen*

Automatisierte Testfallgenerierung ist selbst für konventionelle Software-Systeme eine Herausforderung (vgl. Abschnitt 2.2.4). SPLs fügen der Testfallgenerierung

eine weitere Dimension hinzu, sodass nicht nur für ein Software-System eine Test-Suite generiert werden muss, sondern für eine Menge von Software-Systemen. Zur Vermeidung von produktweiser Testgenerierung mit vielen redundanten Testgenerierungsläufen werden familienbasierte Ansätze verwendet, welche direkt auf SPL-Implementierungen angewendet werden können [COLS11, McG01]. Wie bei der automatisierten Testgenerierung für einzelne Software-Produkte, bei der Ergebnisse einer Analyse des Zustandsraumes des Produktes wiederverwendet werden sollten, ist es auch hier sinnvoll solche Ergebnisse wiederzuverwenden. Die Wiederverwendung sollte allerdings dahingehend erweitert werden, dass Analyseergebnisse nicht nur zwischen Testzielen, sondern auch produktübergreifend wiederverwendet werden. Daraus leiten wir die folgenden Forschungsherausforderungen ab, die wir im Hauptteil dieser Arbeit betrachten werden:

- **FH2.1**: Verbesserung der Wiederverwendung von Berechnungsergebnissen der Programmzustandsraumanalyse zwischen Produkten.
- **FH2.2**: Kombination der Verbesserung der Wiederverwendung von Berechnungsergebnissen der Programmzustandsraumanalyse zwischen Testzielen und Produkten.

2.4 Dynamische Software-Produktlinien

In diesem Abschnitt führen wir die grundlegenden Begriffe und Konzepte von dynamischen Software-Produktlinien ein und beschreiben offene Forschungsherausforderungen im Bereich von dynamischen Software-Produktlinien.

2.4.1 *Grundlagen dynamischer Software-Produktlinien*

Heutzutage müssen Software-Systeme nicht nur extrem anpassbar sein, sondern auch unter sich kontinuierlich ändernden Bedingungen, wie wechselnde Anwendungskontexte, Sub-Systemausfälle und sich ändernde Benutzeranforderungen, funktionieren [LK06, HPS12]. Zusätzlich werden Software-Systeme oft ohne Unterbrechung betrieben. Beispiele für solche selbst-adaptiven Systeme sind Smartphones und Tablets, aber auch Produktivsysteme aus der Automatisierungstechnik, welche immer flexibler werden müssen, um die Anforderungen neuer Produktionsmethoden aus der Industrie 4.0 zu erfüllen [VHRF+16]. Zur Modellierung von individuell konfigurierbaren Systemen, die sehr flexibel zur Laufzeit sind, um sich an sich ändernde Anforderungen und Anwendungskontexte anzupassen, sind dynamische Software-Produktlinien eine geeignete Methodik [HSSF06].

Dynamische Software-Produktlinien (DSPL) erweitern SPLE um die Möglichkeit, SPLs dynamisch zu rekonfigurieren, indem zur Laufzeit zwischen verschiedenen Produktkonfigurationen gewechselt wird. Somit erlauben es DSPLs auf der einen Seite, ein individuell konfigurierbares Software-Produkt am Anfang des Lebenszyklus eines Produktes abzuleiten (z. B. um benutzerspezifische Wünsche zu erfüllen) und auf der anderen Seite während des gesamten Lebenszyklus eines Produktes zur Laufzeit zwischen verschiedenen Software-Produkten zu wechseln (z. B. um das Produkt an neue Anwendungskontexte anzupassen). DSPLs basieren auf den

Prinzipien von SPLE und erweitern SPLs um Laufzeitvariabilität, die zusätzlich die Implementierung von einem Rekonfigurationsmechanismus erfordert [LK06]. Dies wurde bereits erfolgreich in verschiedenen Anwendungsdomänen, wie der industriellen Automatisierung, der Haushaltsrobotik und Informationssystemen, umgesetzt[3].

Beispiel 2.17 (DSPL)**.** In Abschnitt 2.3.2 haben wir bereits die statischen Konfigurationscharakteristiken der DCU herausgearbeitet, wie beispielsweise die verschiedenen DCU-Typen, weshalb die DCU als SPL betrachtet werden kann. Zusätzlich hat die DCU ebenfalls dynamische Konfigurationsaspekte und kann somit auch als DSPL betrachtet werden. Zum Beispiel hängt der ausgewählte Betriebsmodus der DCU vom tatsächlichen Anwendungskontext ab und kann zur Laufzeit rekonfiguriert werden. Der Standardbetriebsmodus ist *Idle*. Um die DCU-Parameter für die Therapie eines bestimmten Patienten einzustellen, muss die DCU vom *Idle*-Modus in den *Adjustment*-Modus wechseln. Dieser Wechsel ist durch eine Rekonfiguration der Produktkonfiguration durch das Abwählen des Features *Idle* und der Anwahl des Features *Adjustment* des DCU-Feature-Modells implementiert (vgl. Abbildung 2.9).

Beispiel 2.17 zeigt, dass eine DSPL statische und dynamische Charakteristiken umfasst, welche typischerweise durch *statische Features* und *dynamische Features* implementiert werden. Statische Features beziehen sich auf die Charakteristiken einer DSPL, die genau einmal während des initialen Konfigurationsprozesses zur Design-Zeit konfiguriert werden. Somit werden statische Features nicht zur Laufzeit rekonfiguriert.

Beispiel 2.18 (Statische Features)**.** Die verschiedenen Typ-Features der DCU (z. B. *Z* und *T*) sind statische Features. Der Typ einer DCU wird genau einmal, während der Installation der DCU in den Synchrotron, festgelegt und ist abhängig von der Aufgabe, welche die DCU bei der Erzeugung des Ionenstrahls erfüllt. Eine Rekonfiguration des DCU-Typs zur Laufzeit wird ausdrücklich nicht angeboten, da die Hardware einer DCU für jeden DCU-Typ unterschiedlich ist.

Im Gegensatz zu statischen Features werden dynamische Features explizit für die Rekonfiguration zur Laufzeit entworfen. Somit beschreiben dynamische Features die dynamischen Charakteristiken einer DSPL. Aus diesem Grund können dynamische Features, abhängig vom aktuellen Anwendungskontext oder wechselnden Benutzeranforderungen, kontinuierlich zur Laufzeit angewählt und abgewählt werden.

Beispiel 2.19 (Dynamische Features)**.** Die Betriebsmodus-Features der DCU sind prädestiniert dafür, dynamische Features zu sein, da sich der ausgewählte Betriebsmodus der DCU während des Lebenszyklus einer DCU kontinuierlich ändern kann. Um zum Beispiel die DCU zu warten, wird die DCU in den *Quality Assurance*-Modus versetzt, wohingegen die DCU für die Behandlung

[3] `http://dspl.lero.ie/`

eines Patienten in den *Therapy*-Modus versetzt wird. Jedoch ist nicht nur der Betriebsmodus einer DCU rekonfigurierbar, sondern auch einige Master-Parameter und typspezifische Parameter, wie beispielsweise *RefValueTHoldMax* und *EnableShutdown*.

DSPLs enthalten statische und dynamische Features und umfassen somit die statischen Konfigurationsaspekte einer SPL sowie dynamische (Re-)Konfigurationsaspekte. Dementsprechend ist die Konfiguration einer DSPL zweigeteilt und besteht zum einen aus der initialen statischen Konfiguration (bei der auch dynamische Aspekte konfiguriert werden können), sowie aus der dynamische Rekonfiguration. Zur Unterscheidung der statischen und der dynamischen (Re-)Konfiguration wurden verschiedene Bindungszeitpunkte für Features eingeführt [SvGB05]. Wir unterscheiden zwischen statischen Bindungszeiten für statische Features und dynamischen Bindungszeiten für dynamische Features.

Der Konfigurationsprozess einer DSPL beginnt mit der Konfiguration statischer Features während der statischen Bindungszeiten. Die statische Konfiguration erfolgt dabei anhand von Benutzeranforderungen und -wünschen, wie es auch für SPLs der Fall ist und ist durch die Abhängigkeiten zwischen den Features eines Feature-Modells beschränkt. Danach folgt der dynamische Teil des Konfigurationsprozesses und das Produkt kann kontinuierlich zur Laufzeit rekonfiguriert werden, z. B. aufgrund wechselnder Anwendungskontexte. Ähnlich zur statischen Konfiguration ist die dynamische Konfiguration ebenfalls durch das Feature-Modell beschränkt und es kann nur von einem validen Produkt zu einem anderen validen Produkt rekonfiguriert werden. Die Beschränkung der Rekonfiguration eines Produktes durch das Feature-Modell ist jedoch für (sicherheitskritische) DSPLs unzureichend, da die Rekonfiguration von jedem validen Produkt zu jedem anderen validen Produkt zu fehlerhaften Verhalten führen kann. Um sicherzustellen, dass nur valide Rekonfigurationen durchgeführt werden, wird das Rekonfigurationsverhalten in der Literatur häufig durch ein Rekonfigurationsmodell beschränkt.

Beispiel 2.20 (Rekonfigurationsbeschränkungen der DCU)**.** Ein Ausschnitt der Spezifikation des erlaubten Rekonfigurationsverhaltens der DCU-Betriebsmodi ist in Abbildung 2.13 zu sehen. Die Behandlung eines Patienten im HIT folgt einem vorgegebenen Protokoll, um die Schädigung des Patienten zu vermeiden. Gemäß der Betriebsmodi der DCU bedeutet dies, dass eine DCU-Instanz zur Vorbereitung der Bestrahlung eines Patienten und während der Bestrahlung selbst die folgenden Schritte durchläuft (rechte Hälfte des Modells aus Abbildung 2.13):

1. **Idle-Modus:** Initial befindet sich jede DCU-Instanz im Idle-Modus.
2. **Experiment-Modus:** Der erste Schritt einer Bestrahlung ist die grobe Konfiguration von DCU-Parametern gemäß der Bedürfnisse des aktuellen Patienten. Deswegen sind im Experiment-Modus alle Parameter frei konfigurierbar.

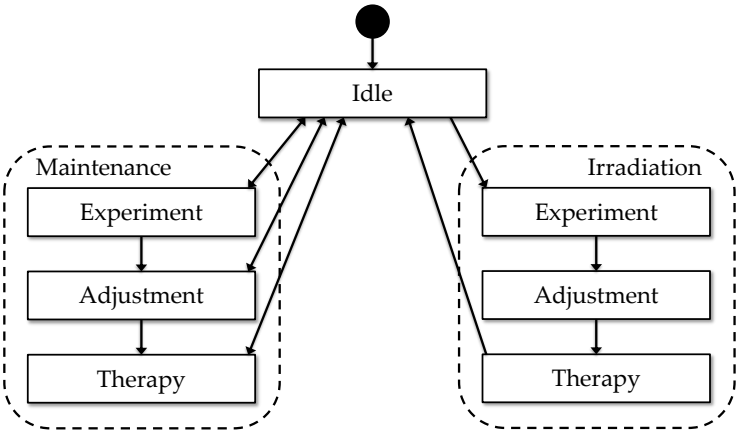


Abbildung 2.13: Ausschnitt des Rekonfigurationsmodells der DCU

3. **Adjustment-Modus:** Eine Feinjustierung der groben Konfiguration erfolgt im Adjustment-Modus.

4. **Therapy-Modus:** Schlussendlich wird der Patient im Therapy-Modus behandelt. In diesem Modus sind alle Parameter auf die Konfiguration aus dem Adjustment-Modus festgelegt, um eine Schädigung des Patienten durch unbeabsichtigte Parameteränderungen zu vermeiden.

Im Gegensatz dazu darf während der Wartung einer DCU beliebig zwischen dem *Idle*-Modus und den anderen Betriebsmodi gewechselt werden (linke Hälfte des Modells aus Abbildung 2.13).

Zusammenfassend erweitern DSPLs SPLs um dynamische Konfigurationsfähigkeiten, welche eine Rekonfiguration zwischen Produkten zur Laufzeit erlauben. Somit umfassen DSPLs statische und dynamische Charakteristiken, die durch statische und dynamische Features beschrieben werden. Ein weitverbreiteter Mechanismus zur Beschränkung des Rekonfigurationsverhaltens sind Rekonfigurationsmodelle.

2.4.2 *Offene Forschungsherausforderungen*

Aktuelle Modellierungsansätze für DSPLs unterschieden zwischen statischen und dynamischen Features, d. h. Features, die vor der Laufzeit konfiguriert werden, und Features, die zur Laufzeit (re-)konfiguriert werden. Davon ausgehend gibt es zwei Stufen in denen Features konfiguriert werden können, eine statische Stufe gefolgt von einer dynamischen Stufe. Besonders bei sicherheitskritischen Systemen erfordert das Design von zuverlässigen DSPLs jedoch die Möglichkeit präzisere und flexiblere temporale Restriktionen zwischen (Re-)Konfigurationen zu definieren als es die bloße Unterscheidung zwischen statischen und dynamischen Features erlaubt. Aus diesem Grund wird ein stufenweiser Konfigurationsprozess für DSPLs benötigt, der eine präzise Beschreibung von temporalen Abhängigkeiten und Restriktionen zwischen Features einer DSPL erlaubt. Insbesondere für sicherheits- und einsatzkritische DSPLs, wie beispielsweise der DCU, ist die Korrektheit der Spe-

zifikation des Rekonfigurationsprozesses inklusive seines stufenweisen Konfiguationsprozesses unbedingt notwendig. Deswegen wird eine automatisierte Validierungsstrategie für DSPLs benötigt, welche die essentiellen Validitätseigenschaften eines stufenweisen Konfigurationsprozesses mit temporalen Abhängigkeiten und Restriktionen zwischen Features sicherstellt. Weiterhin ist neben der präzisen Spezifikation und der Validierung einer DSPL die Verifikation der Implementierung der DSPL essentiell. Neben der Überprüfung der korrekten Implementierung funktionaler Anforderungen an die Produkte einer DSPL umfasst die DSPL-Verifikation zusätzlich die Überprüfung der korrekten Implementierung des Rekonfigurationsverhaltens der DSPL. Daraus leiten wir die folgende Forschungsherausforderung ab, die wir im Hauptteil dieser Arbeit betrachten werden:

- **FH3.1**: Entwicklung eines stufenweisen Konfigurationsprozesses für DSPLs, der die präzise Spezifikation von temporalen Abhängigkeiten zwischen Features erlaubt.
- **FH3.2**: Entwicklung automatisierter Strategien zur Validierung von (Re-)Konfigurationsprozessen von DSPLs inklusive stufenweiser (Re-)Konfiguration.
- **FH3.3**: Entwicklung eines automatisierten Test-Suite-Generierungsansatzes für DSPLs, der insbesondere das Rekonfigurationsverhalten von DSPLs berücksichtigt.

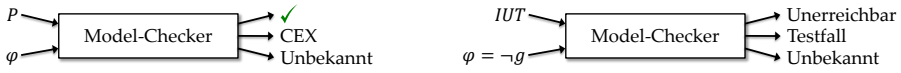
3

AUTOMATISIERTE TESTGENERIERUNG MIT SYMBOLISCHEM MODEL-CHECKING

Qualitätssicherung ist ein aufwändiger und kritischer Teil der Software-Entwicklung. Auf die Qualitätssicherung entfallen, je nach Quelle, zwischen 16% und 37% des gesamten Softwareentwicklungsaufwandes [Lig02], wobei beispielsweise der Testaufwand für sicherheitsrelevante Systeme in einzelnen Fällen sogar auf bis zu 80% steigen kann [SL12]. Deshalb ist es sinnvoll, möglichst viele Schritte der Qualitätssicherung zu automatisieren. Die Automatisierung der Generierung von Testfällen ist einer der vielversprechendsten und effektivsten Wege, um den Aufwand für die Qualitätssicherung zu reduzieren [BGM13, GLM12, CS13, BCH+04]. Aktuelle Techniken zur automatisierten White-Box-Testgenerierung für Komponententests zielen beispielsweise darauf ab, eine Menge von Testzielen bezüglich eines Abdeckungskriteriums abzudecken, die auf Basis des Programm-Codes definiert werden.

Eine vielversprechende Technik zur automatisierten Generierung von Testfällen ist Model-Checking. Hierbei wird mit Hilfe eines Model-Checkers ein Testfall aus einem Gegenbeispiel für die Unerreichbarkeit einer Programmeigenschaft (d. h. einem negierten Testziel) abgeleitet [BCH+04, HdMR04]. Zur Abdeckung einer Menge von Testzielen, die anhand eines Abdeckungskriteriums für ein Programm abgeleitet wurden, kann durch wiederholtes Aufrufen eines Model-Checkers für jedes Testziel ein Testfall generiert werden. Für große Programme mit hunderten oder sogar tausenden Zeilen Code ist die Testgenerierung mit Model-Checking jedoch ineffizient, da die Testgenerierung für jedes Testziel immer wieder eine teure Erreichbarkeitsanalyse auf einer Abstraktion des Programm-Codes durchführt, die während der Testgenerierung mehrfach neu berechnet werden muss [HKL+10]. Model-Checker sind eher auf die Falsifizierung einiger weniger Programmeigenschaften hin optimiert, anstatt auf die Behandlung einer großen Menge von Testzielen. Dies führt dazu, dass die Generierung von Test-Suiten durch das wiederholte Aufrufen eines Model-Checkers ineffizient werden kann, da die Erreichbarkeitsanalyse für jedes Testziel von vorne durchgeführt wird. Die Erreichbarkeitsanalyse immer wieder von vorne durchzuführen, kann bei nahe beieinander liegenden Testzielen zu sehr vielen redundanten Berechnungsschritten führen und sollte dementsprechend vermieden werden.

In diesem Kapitel werden wir einen Ansatz für die automatisierte Testgenerierung mit Model-Checking mit Techniken aus dem Multi-Property-Checking kombinieren, um wiederholte Erreichbarkeitsanalysen für jedes Testziel einzeln zu vermeiden und stattdessen Ergebnisse von Erreichbarkeitsanalysen zwischen Testzielen wiederzuverwenden (vgl. Forschungsherausforderung FH1). Multi-Pro-



(a) Model-Checking **(b)** Test-Generierung

Abbildung 3.1: Test-Generierung mit Model-Checking

perty-Checking erlaubt es, mehrere Programmeigenschaften (Properties) während einer Erreichbarkeitsanalyse gleichzeitig zu analysieren [ABM+16]. Diese Idee verwenden wir, um für mehrere Testziele gleichzeitig während einer einzigen Erreichbarkeitsanalyse eine Test-Suite zu generieren, die diese Testziele abdeckt. Bisherige Erfahrungen zeigen, dass ein Programm im Allgemeinen wesentlich mehr Testziele enthält als kritische Programmeigenschaften. Deswegen sind Anpassungen und Verbesserungen an der Testfallgenerierung mit Multi-Property-Checking notwendig, um die Effizienz und die Korrektheit der Testfallgenerierung zu verbessern. Diese werden wir ebenfalls in diesem Kapitel beschreiben.

3.1 Automatisierte White-Box-Testgenerierung

In diesem Abschnitt beschreiben wir einen Ansatz zur White-Box Test-Suite-Generierung für Komponenten auf Basis von Model-Checking. Neben der grundlegenden Idee der Generierung einzelner Testfälle mit Hilfe von Model-Checking, werden wir einen Algorithmus beschreiben, der ganze Test-Suiten mit Hilfe von Model-Checking generiert. Weiterhin werden wir Verbesserungen der Technik zur Generierung von Test-Suiten beschreiben, die sich insbesondere auf die Wiederverwendung von Berechnungsergebnissen bei der Testgenerierung fokussieren. Die in diesem Abschnitt beschriebenen Ideen zur Testgenerierung mit Model-Checking und zur Wiederverwendung von Berechnungsergebnissen während der Testgenerierung basieren hauptsächlich auf Veröffentlichungen von Beyer et al. [BCH+04] und Holzer et al. [HSTV09, Hol13] und werden in den folgenden Abschnitten erweitert.

3.1.1 *Generierung von Testfällen mit symbolischem Model-Checking*

Abbildung 3.1a zeigt die Anwendung eines Software-Model-Checkers. Als Eingabe erhält der Model-Checker ein Programm P und eine Programmeigenschaft φ, für die gezeigt werden soll, dass die Programmeigenschaft φ für jede Ausführung des Programms P gilt. Dazu führt der Model-Checker eine Analyse auf dem erreichbaren Zustandsraum des Programms durch und überprüft, ob die Programmeigenschaft φ in einem der erreichbaren Programmzustände verletzt ist. Ist φ in einem Programmzustand verletzt, gibt der Model-Checker ein sogenanntes Gegenbeispiel *CEX* (engl. Counterexample) zurück, d. h. einen Pfad im Programmzustandsraum, der zu dem Programmzustand führt, in dem φ verletzt ist. Andernfalls gilt φ im gesamten Programm. In einigen Fällen gibt der Model-Checker für eine Programmeigenschaft φ *unbekannt* zurück, da für keine Abstraktion des Programms entschieden werden kann, ob φ gilt oder nicht. Dies resultiert aus der Unentscheidbarkeit

des Halteproblems und kann beispielsweise vorkommen, wenn das analysierte Programm eine potentielle Endlosschleife enthält.

Die Analyse des erreichbaren Programmzustandsraums durch einen Model-Checker kann gerade für größere Programme sehr ineffizient sein, da für jede Variablenbelegung überprüft werden muss, ob die Variablenbelegung zu einer Verletzung der Programmeigenschaft φ führt. Beispielsweise muss zur Verifikation der Programmeigenschaft φ auf der C-Funktion GaussianSum aus Abbildung 2.4a für alle möglichen Kombinationen von Wertebelegungen für die Eingabevariablen x und y überprüft werden, ob mindestens eine der Kombinationen zu einer Verletzung der Programmeigenschaft φ führt. Um eine Überprüfung aller Eingabewerte zu vermeiden, haben Burch et al. *symbolisches* Model-Checking vorgeschlagen [BCM+90]. Symbolisches Model-Checking erlaubt es, mehrere für das betrachtete Programm äquivalente Variablenbelegungen mit Hilfe von Prädikaten über den Programmvariablen zusammenzufassen und somit gleichzeitig zu analysieren. Das Prädikat $x \leq y$ beschreibt beispielsweise alle Variablenbelegungen, für die x einen kleineren Wert als y zugewiesen bekommen hat. Für die automatisierte Testgenerierung werden wir in dieser Arbeit symbolisches Model-Checking verwenden.

Beyer et al. zeigen in [BCH+04] wie Testfälle aus den Gegenbeispielen eines symbolischen Model-Checkers generiert werden können. Eine Skizze der Idee ist in Abbildung 3.1b dargestellt. Als Eingabe erhält der Model-Checker die IUT und die Programmeigenschaft $\varphi = \neg g$, d. h. die Negierung des abzudeckenden Testziels g. Der Model-Checker führt eine durch das Testziel getriebene Erreichbarkeitsanalyse auf der IUT durch und überprüft, ob das Testziel g erreichbar ist, d. h. ob es einen Programmzustand gibt, in dem die Programmeigenschaft φ verletzt ist. Wenn das Testziel g erreichbar ist und somit die Programmeigenschaft φ verletzt ist, gibt der Model-Checker ein Gegenbeispiel zurück, d. h. einen Pfad durch den Programmzustandsraum der IUT, der zum Testziel g führt und der als *abstrakter* Testfall interpretiert werden kann. Da wir symbolisches Model-Checking verwenden, beschreibt der abstrakte Testfall eine *Menge* von konkreten Testfällen, d. h. konkreten Eingabewerten (vgl. Definition 2.1), die den Pfad im Programmzustandsraum der IUT traversieren. Ist das Testziel g im Programmzustandsraum nicht erreichbar, nennen wir g *unerfüllbar* und wenn der Model-Checker nicht entscheiden kann, ob g erreichbar ist, nennen wir g *unbekannt*.

Basierend auf dieser Idee werden wir im Folgenden an einem Beispiel zeigen, wie die Testziel-getriebene Generierung von Testfällen mit Hilfe von symbolischem Model-Checking durchgeführt werden kann. Die Basis dafür bilden existierende Ansätze aus der Literatur von Holzer et al. [HSTV09, Hol13]. Weiterhin werden wir das Vorgehen zur Testgenerierung formal beschreiben und einen Algorithmus zur Generierung von Test-Suiten präsentieren, der die Testziel-getriebene Generierung von Testfällen mit Hilfe von symbolischem Model-Checking für jedes Testziel anwendet. Auf dieser Grundlage werden wir weitergehende Techniken beschreiben, welche die Testgenerierung effizienter machen können, indem die Wiederverwendung von Berechnungsergebnissen der Erreichbarkeitsanalyse zwischen Testzielen ermöglicht wird (vgl. Forschungsherausforderung FH1).

3.1.2 *Programmsyntax und -semantik*

Software-Model-Checker arbeiten üblicherweise auf Basis einer abstrakten Repräsentation der Programm-Syntax, z. B. Kontrollflussautomaten (CFA, engl. Control-flow Automaton). In diesem Abschnitt werden wir eine Programmsemantik basierend auf CFAs definieren, die uns als Grundlage für die Beschreibung der Testgenerierung dienen wird.

Zuerst werden die Programmvariablen und Programmoperationen eines Programms beschrieben. Die Menge der Programmvariablen V eines Programms ist über einer zusammengesetzen Domäne $\mathcal{D}$ definiert, die wir in dieser Arbeit auf Boole'sche und ganzzahlige Variablen einschränken, d. h. $\mathcal{D} = \mathbb{B} \cup \mathbb{Z}$. Die Domäne $\mathbb{B} = \{0, 1\}$ enthält die zwei Werte „falsch" (false/0) und „wahr" (true/1). Die Spache der Programmoperationen ist gegeben als $\mathcal{L}(V) = \mathcal{L}_{assume}(V) \cup \mathcal{L}_{assign}(V)$, wobei $\mathcal{L}_{assume}(V) = (V \to \mathcal{D}) \to \mathbb{B}$ das Teilalphabet der Prädikate über V darstellt und $\mathcal{L}_{assign}(V) = (V \to \mathcal{D}) \to (V \to \mathcal{D})$ das Teilalphabet der Wertezuweisungen über V [BLB+15]. Die Elemente der Sprache der Programmoperationen werden mit $a \in \mathcal{L}(V)$ bezeichnet. Mit Hilfe von Programmvariablen und Programmoperationen können Kontrollflussautomaten für ein Programm definiert werden.

Definition 3.1 (CFA)**.** Sei V eine Menge von Programmvariablen und $\mathcal{L}(V)$ die Sprache der Programmoperationen über V. Ein CFA ist ein Quadrupel (L, ℓ_0, ℓ_t, E), wobei

- L eine endliche Menge von Programm-Locations,
- $\ell_0 \in L$ die initiale Programm-Location,
- $\ell_t \in L$ die terminale Programm-Location und
- $E \subseteq L \backslash \{\ell_t\} \times \mathcal{L}(V) \times L \backslash \{\ell_0\}$ eine über Programmoperationen beschriftete Transitionsrelation

ist.

Damit ein CFA wohlstrukturiert ist, fordern wir, dass der Initialknoten ℓ_0 keine eingehenden Kanten hat und alle Knoten, außer des Terminierungsknotens ℓ_t, entweder genau eine ausgehende Kante haben, die mit einer Wertezuweisung aus $\mathcal{L}_{assign}(V)$ beschriftet ist oder dass alle ausgehenden Kanten mit sich gegenseitig ausschließenden Prädikaten aus $\mathcal{L}_{assume}(V)$ beschriftet sind.

Beispiel 3.2 (CFA)**.** Abbildung 3.2 zeigt den CFA für die C-Funktion GaussianSum aus Abbildung 2.4a. Programm-Locations sind mit Programmzeilennummern beschriftet und werden entweder durch Kreise dargestellt, wenn die Programm-Locations genau eine ausgehende Kante haben, die mit einer Wertezuweisung aus der Menge $\mathcal{L}_{assign}(V)$ beschriftet ist oder durch Rauten, wenn die Programm-Locations genau zwei ausgehende Kanten haben, die mit sich gegenseitig ausschließenden Prädikaten aus der Menge $\mathcal{L}_{assume}(V)$ beschriftet sind. Die Programm-Location 2 hat genau eine ausgehende Transition, die mit

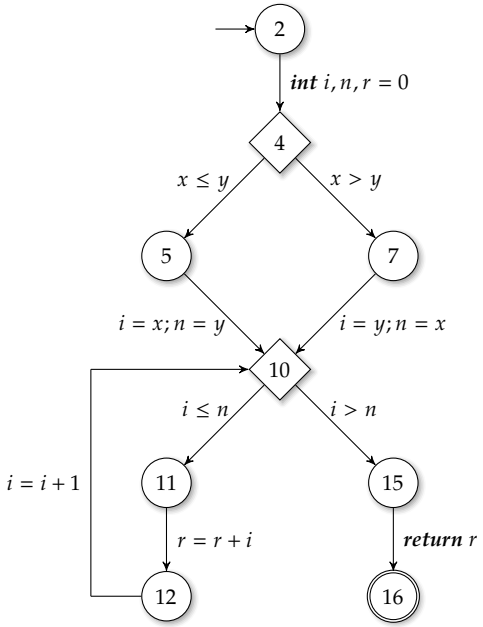


Abbildung 3.2: CFA der C-Funktion GaussianSum

der Wertezuweisung ***int*** $i, n, r = 0$ beschriftet ist. Die Bedingung in Programm-Location 4 führt hingegen zu zwei ausgehenden Kanten, die jeweils mit einem der sich gegenseitig ausschließenden Prädikate $x \leq y$ und $x > y$ beschriftet sind. Die initiale Programm-Location des CFAs ist Programm-Location 2 und die terminale Programm-Location ist Programm-Location 16.

Jede Ausführung eines Programms traversiert genau einen *CFA-Pfad*

$$\ell_0 \xrightarrow{a_0} \ell_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{k-1}} \ell_k,$$

der bei der initialen Programm-Location ℓ_0 beginnt, sodass $(\ell_i, a_i, \ell_{i+1}) \in E$ für $0 \leq i < k$ gilt. Beim Traversieren des CFA-Pfades während einer Programmausführung werden die Programm-Locations des CFA-Pfades um Informationen zur Variablenbelegung angereichert, mit der die Programm-Locations während der Programmausführung erreicht wurden. Ein Programmzustand einer Programmausführungen ist somit ein Paar (ℓ, c), bestehend aus einer Programm-Location $\ell \in L$ und einer Wertebelegung von Variablen $c : V \to \mathcal{D}$ mit der die Programm-Location erreicht wurde. Die Menge aller konkreten Wertebelegungen von Variablen wird mit C bezeichnet. Ist die Programmoperation a bei einem Übergang $c_i \xrightarrow{a} c_{i+1}$ zwischen zwei Programmzuständen c_i und c_{i+1} eine Zuweisung (d. h. $a \in \mathcal{L}_{assign}(V)$), werden die Werte der Programmvariablen gemäß der Programmoperation a in c_{i+1} angepasst. Andernfalls ist die Programmoperation a bei einem Übergang $c_i \xrightarrow{a} c_{i+1}$ ein Prädikat (d. h. $a \in \mathcal{L}_{assume}(V)$) und der Übergang von c_i nach c_{i+1} ist nur erlaubt, wenn die Werte in c_i das Prädikat in a erfüllen, d. h. es muss $c_i \models a_i$ gelten. Der Operator $\varphi_1 \models \varphi_2$ ist so definiert, dass φ_2 für alle Variablenbelegungen wahr ist, für die φ_1 wahr ist.

Definition 3.3 (Programmausführung [BLB+15])**.** Sei (L, ℓ_0, ℓ_t, E) ein CFA. Eine Programmausführung entspricht einer Sequenz

$$\sigma = (\ell_0, c_0) \xrightarrow{e_0} (\ell_1, c_1) \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} (\ell_k, c_k)$$

von *Programmzuständen*, d. h. Paaren (ℓ_i, c_i), $0 \leq i \leq k$, bestehend aus Programm-Locations $\ell_i \in L$ und Belegungen von Programmvariablen, die durch eine Abbildung $c_i : V \to \mathcal{D}$ angegeben werden, sodass

$$c_{i+1} = \begin{cases} a_i(c_i) & \text{falls } a_i \in \mathcal{L}_{assign}(V) \\ c_i \text{ und } c_i \models a_i & \text{falls } a_i \in \mathcal{L}_{assume}(V) \end{cases}$$

gilt mit $e_i = (\ell_i, a_i, \ell_{i+1}) \in E$.

Wir nennen eine Programmausführung *vollständig*, wenn die Programm-Location des letzten Programmzustands dem Programmende entspricht, d. h. $\ell_k = \ell_t$, andernfalls nennen wir die Programmausführung *partiell*.

Beispiel 3.4 (Programmausführung)**.** Gegeben sei der Testfall $tc_1 = ([x = 2, y = 2], [r = 2])$ für die GaussianSum-Funktion aus Beispiel 2.2. Der CFA-Pfad, der durch den Eingabevektor $[x = 2, y = 2]$ des Testfalls tc_1 traversiert wird, ist

$$2 \xrightarrow{\textbf{int } i,n,r=0} 4 \xrightarrow{x \leq y} 5 \xrightarrow{i=x;n=y;} 10 \xrightarrow{i \leq n} 11 \xrightarrow{r=r+i} 12 \xrightarrow{i=i+1} 10 \xrightarrow{i>n} 15 \xrightarrow{\textbf{return } r} 16.$$

Die entsprechende Programmausführung zu dem CFA-Pfad ist

$$\begin{aligned} & (2, [x = 2, y = 2, i = \bot, n = \bot, r = \bot]) \\ \xrightarrow{\textbf{int } i,n,r=0} \; & (4, [x = 2, y = 2, i = \bot, n = \bot, r = 0]) \\ \xrightarrow{x \leq y} \; & (5, [x = 2, y = 2, i = \bot, n = \bot, r = 0]) \\ \xrightarrow{i=x;n=y;} \; & (10, [x = 2, y = 2, i = 2, n = 2, r = 0]) \\ \xrightarrow{i \leq n} \; & (11, [x = 2, y = 2, i = 2, n = 2, r = 0]) \\ \xrightarrow{r=r+i} \; & (12, [x = 2, y = 2, i = 2, n = 2, r = 2]) \\ \xrightarrow{i=i+1} \; & (10, [x = 2, y = 2, i = 3, n = 2, r = 2]) \\ \xrightarrow{i>n} \; & (15, [x = 2, y = 2, i = 3, n = 2, r = 2]) \\ \xrightarrow{\textbf{return } r} \; & (16, [x = 2, y = 2, i = 3, n = 2, r = 2]). \end{aligned}$$

Das Symbol $\bot$ bedeutet, dass einer Variablen noch kein Wert zugewiesen wurde, d. h. dass der Variablenwert undefiniert ist.

Für die Analyse des Programmzustandsraums konstruiert ein Model-Checker einen Erreichbarkeitsgraphen, der den erreichbaren Programmzustandsraum eines Programms repräsentiert und somit alle Programmausführungen eines Programms enthält. Der Erreichbarkeitsgraph enthält alle Programmzustände (ℓ, c)

eines Programms sowie die Transitionen zwischen diesen Programmzuständen, welche die erlaubten Programmzustandswechsel in einem Programm gemäß des CFAs des Programms beschreiben. Somit beschreibt der Erreichbarkeitsgraph die Semantik eines Programms.

Definition 3.5 (Erreichbarkeitsgraph)**.** Ein Erreichbarkeitsgraph zu einem CFA (L, ℓ_0, ℓ_t, E) ist ein Tripel (S_K, s_{0_K}, T_K), wobei

- $S_K \subseteq L \times C$ eine Menge von Programmzuständen,
- $s_{0_K} \in S_K$ der initiale Programmzustand und
- $T_K \subseteq S_K \times E \times S_K$ eine beschriftete Transitionsrelation mit $(\ell, c) \xrightarrow{e} (\ell', c') \in T_K$, wenn $e = (\ell, a, \ell') \in E$ und

$$c' = \begin{cases} a(c) & \text{falls } a \in \mathcal{L}_{assign}(V) \\ c \text{ und } c \models a & \text{falls } a \in \mathcal{L}_{assume}(V) \end{cases}$$

  gilt,

ist.

Wir nennen einen Erreichbarkeitsgraphen *vollständig*, wenn es für jede konkrete Programmausführung

$$\sigma = (\ell_0, c_0) \xrightarrow{e_0} (\ell_1, c_1) \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} (\ell_k, c_k)$$

eines Programms einen entsprechenden Pfad im Erreichbarkeitsgraphen gibt.

Beispiel 3.6 (Erreichbarkeitsgraph)**.** Abbildung 3.3 zeigt einen Ausschnitt des Erreichbarkeitsgraphen der C-Funktion GaussianSum. Jeder erreichbare Programmzustand enthält eine Programm-Location und eine Variablenbelegung mit welcher der Programmzustand erreicht werden kann. Das Symbol $\perp$ steht für einen undefinierten Wert einer Variablen. Der Pfad $1 - 2 - 4 - 5 - 10 - \ldots$ entspricht einem Ausschnitt der Programmausführung aus Beispiel 3.4. Die Transitionsbeschriftungen enthalten nur die Programmoperationen der entsprechenden CFA-Kanten für eine bessere Übersichtlichkeit.

Um den Erreichbarkeitsgraphen zu vervollständigen, müsste für jeden validen Eingabevektor der entsprechende Programmausführungspfad ergänzt werden. Dies würde in diesem Fall je einen Programmausführungspfad für jede Kombination der Werte der Integer-Variablen x und y entsprechen. Unter der Annahme, dass jeder Integer-Wert aus 32 Bits besteht, gibt es 2^{32} verschiedene Integer-Werte und somit $2^{32} * 2^{32}$ verschiedene Programmausführungen der GaussianSum-Funktion.

Wie in Beispiel 3.6 zu sehen ist, kann der Programmzustandsraum selbst für kleine Programme schnell sehr groß werden. Eine Analyse des Programmzustandsraums durch einen Model-Checker ist deshalb für mittelgroße und große Programme im Allgmeinen nicht innerhalb einer realistischen Laufzeit möglich. Um

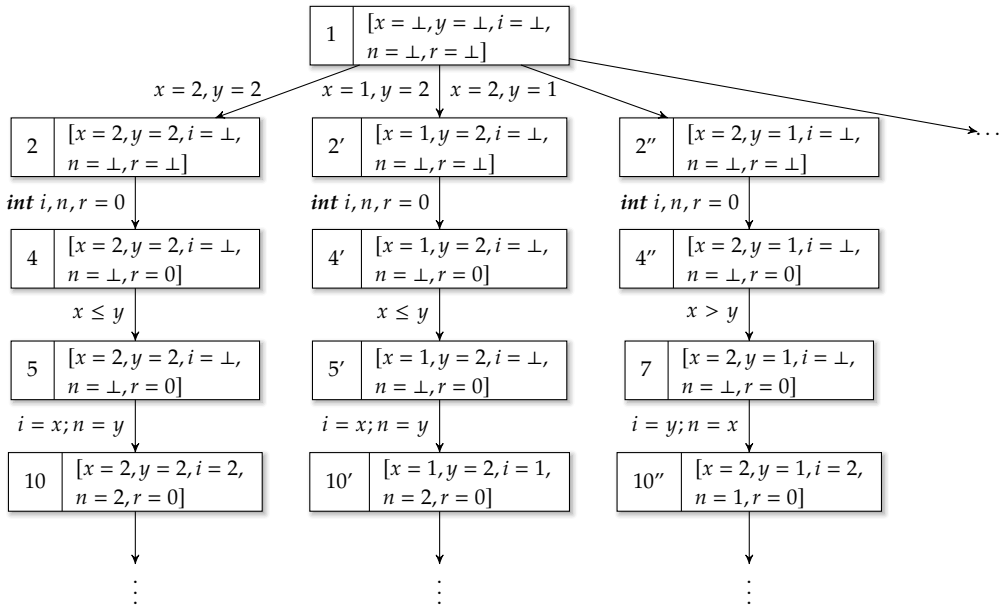


Abbildung 3.3: Ausschnitt des Erreichbarkeitsgraphen der C-Funktion GaussianSum

dennoch größere Programme analysieren zu können, wurden symbolische Model-Checker eingeführt. Im Gegensatz zu konkreten Model-Checkern berechnen symbolische Model-Checker nicht den konkreten Programmzustandsraum wie in Beispiel 3.6, sondern eine Abstraktion des Programmzustandsraums [BCM+90]. Bei der Berechnung einer Abstraktion eines Programmzustandsraums werden mehrere Programmzustände zu einem *abstrakten* Programmzustand zusammengefasst. Zum Zusammenfassen mehrere Programmzustände werden *Zustandsprädikate* $r \in \mathcal{L}_{assume}(V)$ verwendet, die mehrere Variablenbelegungen charakterisieren. Diese Zustandsprädikate nennen wir im Folgenden *Regionen* und mit $R(V)$ bezeichnen wir die Menge aller Regionen über V. Zur Unterscheidung verschiedener Wertezuweisungen zu einer Variablen v in einer Region, wird für jede neue Wertezuweisung eine neue Variable v_i mit einem inkrementierten Index i eingeführt. Beispielsweise bei der Neuzuweisung eines Wertes zur Variablen r basierend auf dem alten Wert der Variablen r, kann die Region $r_1 = r_0 + 1$ verwendet werden, wobei r_0 den alten Wert der Variablen r enthält und r_1 den neuen Wert der Variablen r. Dieses Konzept nennt sich *Static Single Assignment (SSA)* [RWZ88]. Ein abstrakter Zustand (ℓ, r) auf einem Pfad durch den Programmzustandsraum ist ein Paar, bestehend aus einer Programm-Location $\ell \in L$ und einer *Pfadbedingung* r. Die Pfadbedingung ist eine Region, die eine Menge von Programmzuständen $C = \{c_0, \ldots, c_k\}$ charakterisiert, sodass $c_i \models r$ für $0 \leq i \leq k$ gilt. Beispielsweise beschreibt der abstrakte Zustand $(5, x \leq y)$ alle Programmzustände mit der Programm-Location 5 in denen der Variablen x ein kleinerer oder der gleiche Wert wie der Variablen y zugewiesen wurde. Abstrakte Programmzustände sind Verallgemeinerungen von konkreten Programmzuständen. Somit kann jeder konkrete Programmzustand auch als abstrakter Programmzustand mit einer Region dargestellt werden, die genau eine konkrete Variablenbelegung beschreibt.

Zur Berechnung symbolscher Ausführungen verwenden wir den *Strongest-Post-condition-Operator*, der für die Anwendung einer Programmoperation auf eine Re-

gion die minimale Menge aller konkreten nachfolgenden Variablenbelegungen der Region berechnet, die wieder als Region charakterisiert werden. Für einen Übergang $r \xrightarrow{a} r'$ von einer Region r zu einer Region r' mit einer Programmoperation a bedeutet das, dass die Programmoperationen a auf alle konkreten Variablenbelegungen c gemäß Definition 3.3 angewendet werden, die durch r charakterisiert werden. Die aus der Anwendung der Programmoperationen a resultierenden Variablenbelegungen c' sind die nachfolgenden Variablenbelegungen, die durch r' beschrieben werden können.

Definition 3.7 (Strongest-Postcondition [BHT08])**.** Der *Strongest-Postcondition-Operator* $sp_V : R(V) \times \mathcal{L}(V) \rightarrow R(V)$ ist definiert, sodass für $r' \Leftrightarrow sp_V(r, a_i)$ und für jedes $c_i \models r$ entweder

- $c_{i+1} = a_i(c_i)$ und $c_{i+1} \models r'$, wenn $a_i \in \mathcal{L}_{assign}(V)$ oder
- $c_i \wedge a_i \models r'$, wenn $a_i \in \mathcal{L}_{assume}(V)$

gilt.

Basierend auf Regionen und der Strongest-Postcondition können Programmausführungspfade symbolisch dargestellt werden. Eine symbolische Programmausführung charakterisiert dabei im Allgemeinen unendlich viele konkrete Programmausführungspfade, wodurch die Anzahl der Programmausführungspfade für die Analyse des Programmzustandsraums eines Programms durch einen symbolischen Model-Checker potentiell reduziert und endlich dargestellt werden kann.

Beispiel 3.8 (Symbolische Programmausführung)**.** Für einen CFA-Pfad umfasst eine symbolische Programmausführung mehrere konkrete Programmausführungen. Eine symbolische Programmausführung, welche unter anderem die konkrete Programmausführung aus Beispiel 3.4 umfasst, ist

$$
\begin{aligned}
& (2, \mathit{true}) \\
\xrightarrow{\textbf{int } i,n,r=0} \quad & (4, r_0 = 0) \\
\xrightarrow{x \leq y} \quad & (5, r_0 = 0 \wedge x_0 \leq y_0) \\
\xrightarrow{i = x; n = y;} \quad & (10, r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0) \\
\xrightarrow{i \leq n} \quad & (11, r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0 \wedge i_0 \leq n_0) \\
\xrightarrow{r = r + i} \quad & (12, r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0 \wedge i_0 \leq n_0 \\
& \qquad \wedge r_1 = r_0 + i_0) \\
\xrightarrow{i = i + 1} \quad & (10, r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0 \wedge i_0 \leq n_0 \\
& \qquad \wedge r_1 = r_0 + i_0 \wedge\ i_1 = i_0 + 1) \\
\xrightarrow{i > n} \quad & (15, r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0 \wedge i_0 \leq n_0 \\
& \qquad \wedge r_1 = r_0 + i_0 \wedge\ i_1 = i_0 + 1 \wedge i_1 > n_0)
\end{aligned}
$$

$$\xrightarrow{\textit{return } r} (16, r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0 \wedge i_0 \leq n_0 \\ \wedge\ r_1 = r_0 + i_0 \wedge\ i_1 = i_0 + 1 \wedge i_1 > n_0).$$

Diese symbolische Programmausführung beschreibt alle Programmausführungen, die dem CFA-Pfad $2 \to 4 \to 5 \to 10 \to 11 \to 12 \to 10 \to 15 \to 16$ entsprechen. Zur Unterscheidung der verschiedenen Werte, die eine Variable v während einer symbolischen Programmausführung durch Mehrfachzuweisungen haben kann, wurde unter Verwendung von SSA für jede neue Zuweisung eines Wertes zu v eine neue Variable v_i mit einem inkrementierten Index i eingeführt, z. B. für die zweite Zuweisung eines Wertes zur Variable r in Zeile 11 wurde die Variable r_1 eingeführt, die den neuen Wert der Variable r erhält.

Mit Hilfe von symbolischen Programmausführungen kann der erreichbare Programmzustandsraum eines Programms symbolisch dargestellt werden, um die Anzahl der erreichbaren Programmzustände und der zugehörigen Programmzustandsübergänge durch Abstraktion zu reduzieren. Zur Darstellung des abstrakten Programmzustandsraums verwenden wir einen *abstrakten Erreichbarkeitsgraphen (ARG, engl. Abstract Reachability Graph)*. Im Gegensatz zum Erreichbarkeitsgraphen wird der ARG auf abstrakten Programmzuständen anstatt auf konkreten Programmzuständen definiert.

Definition 3.9 (Abstrakter Erreichbarkeitsgraph (ARG) [BHT08])**.** Ein ARG für einen CFA (L, ℓ_0, ℓ_t, E) ist ein Tripel (S, s_0, T), wobei

- $S = L \times R(V)$ eine Menge von abstrakten Programmzuständen,
- $s_0 = (\ell_0, \mathit{true}) \in S$ der initiale abstrakte Programmzustand und
- $T \subseteq S \times E \times S$ eine beschriftete Transitionsrelation mit $(\ell, r) \xrightarrow{e} (\ell', r') \in T$, wenn $e = (\ell, a, \ell') \in E$ und $r' \models sp_V(r, a)$ gelten,

ist.

Wir nennen einen ARG *vollständig*, wenn es für jede konkrete Ausführung

$$\sigma = (\ell_0, c_0) \xrightarrow{e_0} (\ell_1, c_1) \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} (\ell_k, c_k)$$

eines Programms einen entsprechenden abstrakten ARG-Pfad

$$\omega = (\ell_0, r_0) \xrightarrow{e_0} (\ell_1, r_1) \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} (\ell_k, r_k)$$

mit $c_i \models r_i$ für $0 \leq i \leq k$ gibt.

Beispiel 3.10 (Abstrakter Erreichbarkeitsgraph (ARG))**.** Abbildung 3.4 zeigt einen Ausschnitt des ARGs der C-Funktion GaussianSum aus Abbildung 3.2. Jeder abstrakte Programmzustand besteht aus einer Programm-Location und einer Pfadbedingung, die beschreibt, mit welchen konkreten Variablenbelegung der abstrakte Programmzustand erreichbar ist. Jeder ARG-Pfad entspricht einem Pfad des CFAs der GaussianSum-Funktion.

Wie bei den symbolischen Ausführungspfaden, werden mit Hilfe von SSA die verschiedenen Wertezuweisungen zu Variablen unterschieden. Beispiels-

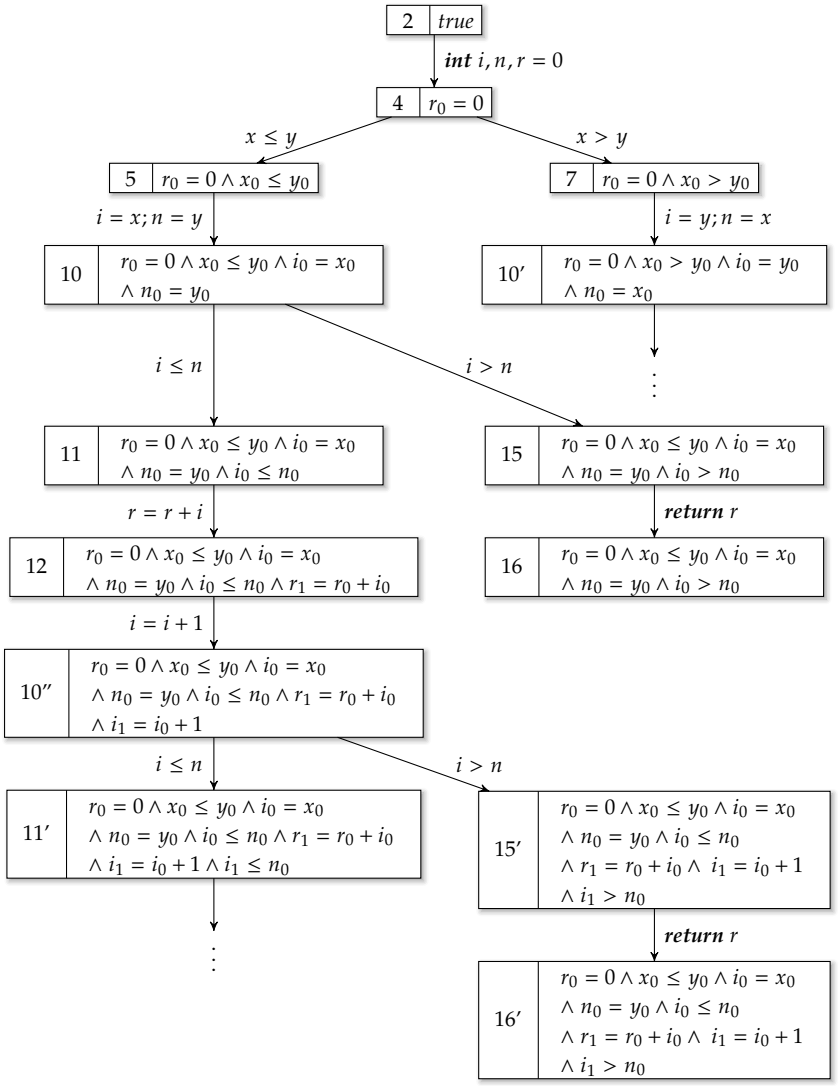


Abbildung 3.4: Ausschnitt des ARGs der C-Funktion GaussianSum

weise wird bei jedem Durchlaufen der Programm-Location 11 eine neue Variable $r_0, r_1, r_2, \ldots$ eingefügt, um die verschiedenen Werte der Variablen r während der Programmausführung zu speichern (z. B. in Programmzustand 12). Die konjugierten Zustandsprädikate über den Programmvariablen eines CFA-Pfades ergeben die Pfadbedingung eines Programmzustands, welche durch die Werte der Programmvariablen erfüllt sein muss, um die entsprechende Programm-Location während einer konkreten Programmausführung zu erreichen. Beispielsweise entspricht der ARG-Pfad 2 – 4 – 5 – 10 dem CFA-Pfad

$$2 \xrightarrow{\textbf{int } i,n,r=0} 4 \xrightarrow{x \leq y} 5 \xrightarrow{i=x;n=y} 10.$$

Die Pfadbedingung von Programmzustand 10 ergibt sich aus der Konjunktion der Zustandsprädikate der CFA-Kanten unter Berücksichtigung der SSA-In-

dizes für Wertezuweisungen zu Programmvariablen. Somit ist die Pfadbedingung von Programmzustand 10

$$r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0.$$

Die Verwendung von Regionen als Pfadbedingung von abstrakten Programmzuständen im ARG unterstützt Abstraktion und erlaubt es, Informationen über den Programmzustandsraum wegzulassen, die für eine Analyse nicht notwendig sind, d. h. die abstrakten Programmzustände können eine Überapproximation einer Menge von konkreten Programmzuständen sein und somit mehr konkrete Programmzustände enthalten als im eigentlichen Programm erreichbar sind. Durch die Anwendung von Techniken wie Counterexample-guided Abstraction Refinement (CEGAR) stellt dies jedoch kein Problem dar, da Gegenbeispiele, die aus der Überapproximation des Programmzustandsraums resultieren (und somit nicht im konkreten Programm selbst enthalten sind), identifiziert und ausgeschlossen werden können [Cla03].

Basierend auf den Definitionen für CFAs und Programmausführungen können wir im folgenden Abschnitt Testziele definieren sowie spezifizieren in welchen Fällen ein Testziel durch eine Programmausführung abgedeckt wird.

3.1.3 *Spezifikation von Testzielen*

Testziele wurden bisher zur besseren Verständlichkeit der allgemeinen Konzepte als einzelne Anweisungen angegeben, die durch die Ausführung eines Testfalls traversiert werden müssen, um abgedeckt zu werden. Um in der Praxis relevante Abdeckungskriterien zu spezifizieren, ist es jedoch notwendig, Testziele nicht nur als einzelne Anweisungen beschreiben zu können, sondern beispielsweise als Sequenz von Anweisungen. Aus diesem Grund verwenden wir im Folgenden *Testzielautomaten* (TGA, engl. Test-Goal Automaton) zur Spezifikation von Testzielen. TGAs wurden von Holzer et al. eingeführt [HSTV10, HTSV10]. Ein TGA ist ein nichtdeterministischer, endlicher Automat, der eine Menge von CFA-Pfaden beschreibt, die ein Testziel spezifizieren. Dadurch können beispielsweise Testziele spezifiziert werden, die das sequentielle Durchlaufen zweier Programm-Locations in einem Testfall fordern. In dieser Arbeit verwenden wir TGAs für zwei Zwecke:

1. Zur Spezifikation von Testzielen.
2. Für die Testziel-getriebene Generierung von Testfällen mit symbolischem Model-Checking.

Definition 3.11 (Testzielautomat (TGA) [Hol13])**.** Ein Testzielautomat $A = (Q, \Sigma, \Delta, q_0, F)$ für einen CFA (L, ℓ_0, ℓ_t, E) und eine Menge von Programmvariablen V ist ein nichtdeterministischer, endlicher Automat, wobei

- Q eine Menge von Zuständen,

- $\Sigma \subseteq E \times R(V)$ ein Alphabet bestehend aus Paaren von CFA-Kanten und Zustandsprädikaten,
- $\Delta \subseteq Q \times \Sigma \times Q$ eine Transitionsrelation,
- $q_0 \in Q$ der Initialzustand und
- $F \subseteq Q$ eine Menge von Endzuständen

ist.

Wir schreiben $q \xrightarrow{e} q'$ für $(q, e, q') \in \Delta$ und beschriften eine Transition mit *, wenn die Transition jede CFA-Kante akzeptiert. Wir fordern, dass jeder TGA-Zustand eine Selbsttransition hat, die mit * beschriftet ist. Wir sagen *A akzeptiert* eine (partielle) Programmausführung $(\ell_0, c_0) \xrightarrow{e_0} (\ell_1, c_1) \cdots (\ell_{k-1}, c_{k-1}) \xrightarrow{e_{k-1}} (\ell_k, c_k)$, wenn es eine Sequenz $q_0 \xrightarrow{(e_0, r_0)} q_1 \cdots q_{k-1} \xrightarrow{(e_{k-1}, r_{k-1})} q_k$ von TGA-Transitionen gibt, die im initialen Zustand q_0 startet, sodass $c_{i+1} \models r_i$ für jede Transition $q_i \xrightarrow{(e_i, r_i)} q_{i+1}$, $0 \leq i < k$, hält. Im Folgenden werden wir die Begriffe Testziel und TGA synonym verwenden.

Zur Spezifikation von kompletten Programmausführungen anstelle von partiellen Programmausführungen durch einen TGA können wir explizit auf die CFA-Kante verweisen, die das Programm verlässt, d. h. den Terminierungsknoten ℓ_t eines CFAs als Zielknoten hat. Komplette Programmausführungen erlauben es, neben der Testeingabe auch ein Testorakel für einen Testfall abzuleiten.

Zur Spezifikation mehrerer TGAs, die ein komplettes Abdeckungskriterium erfüllen, können Queries verwendet werden, die mit Hilfe der FQL (FShell Query Language) spezifiziert werden [HSTV10, HTSV10]. Die FQL ist eine Spezifikationssprache, die es mit Hilfe von Schlüsselwörtern erlaubt Testziele sowohl generisch für alle Programme als auch spezifisch für ein Programm zu spezifizieren. Aus einer FQL-Query können automatisiert TGAs für einen CFA zur Testfallgenerierung abgeleitet werden [Hol13].

Beispiel 3.12 (Testzielautomaten (TGA))**.** Abbildung 3.5 zeigt Beispiele für TGAs zur Spezifikation verschiedener Abdeckungskriterien für die C-Funktion GaussianSum. An den Transitionen wurden die Zustandsprädikate sowie die Start- und Endzustände von CFA-Kanten für eine bessere Lesbarkeit weggelassen. Die Zustandsprädikate sind in diesen (und allen folgenden) Beispielen implizit *true*.

Die Testziele tga_1 und tga_2 sind Beispiele für Basic-Block-Abdeckung. Beispielsweise beschreibt tga_1 alle CFA-Pfade, welche die Anweisungen $i = x; n = y$ traversieren. Die Zustände q_1^0 und q_1^1 haben Selbsttransitionen, die mit $*$ beschriftet sind und jede Anweisung erlauben. Für eine vollständige Basic-Block-Abdeckung muss für jeden Basic-Block eines Programms ein TGA erstellt werden, der ähnlich zu tga_1 und tga_2 ist. Die TGAs für Basic-Block-Abdeckung können aus der generischen FQL-Query

$$cover\ "\_*".EDGES(@BASICBLOCKENTRY)."\_*"$$

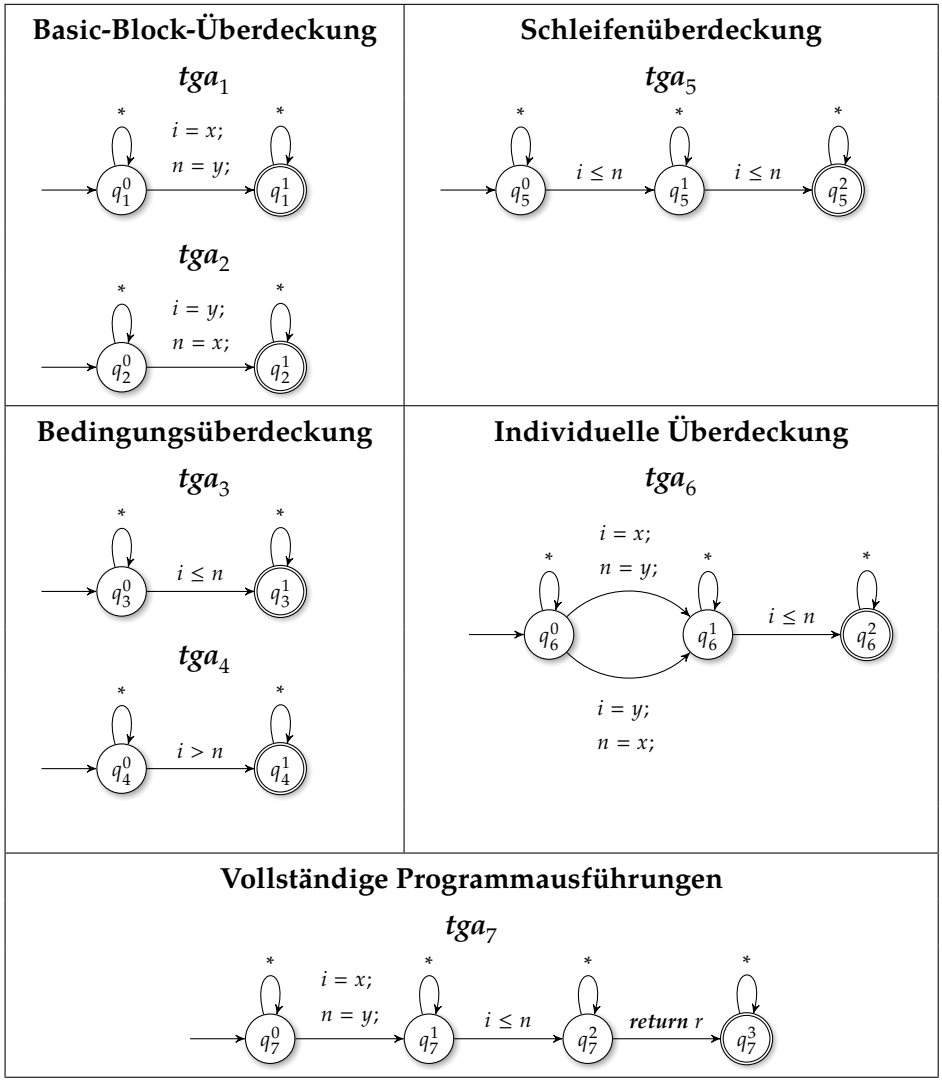


Abbildung 3.5: Testzielautomaten für die Funktion GaussianSum

automatisiert für jedes Programm abgeleitet werden. Das Schlüsselwort *BASIC-BLOCKENTRY* referenziert dabei alle Basic-Blocks eines Programms [HSTV10]. Ähnlich wird bei der Bedingungsabdeckung vorgegangen, bei der für jede Bedingung der entsprechende If- und der entsprechende Else-Branch abgedeckt werden müssen. Dies wird für die Bedingung im Kopf der While-Schleife in Zeile 10 der GaussianSum-Funktion durch die TGAs tga_3 und tga_4 erreicht. Der TGA tga_5 zeigt ein Beispiel dafür, wie Schleifenabdeckung spezifiziert werden kann. Der TGA fordert, dass die While-Schleife der GaussianSum-Funktion mindestens zwei Mal durchlaufen wird. Auch komplexere Testziele können mit TGAs spezifiziert werden, wie beispielsweise mit dem TGA tga_6 demonstriert wird. Zur Abdeckung des Testziels tga_6 müssen entweder die Anweisungen $i = x; n = y$ gefolgt von einem Schleifendurchlauf traversiert werden oder die Anweisungen $i = y; n = x$ gefolgt von einem Schleifendurchlauf. Der TGA tga_7 spezifiziert ein Testziel, dass eine vollständige Programmausführung zur Ab-

deckung fordert, da explizites Beenden des Programms durch Traversierung der Return-Anweisung gemäß der Transition $q_7^2 \xrightarrow{\textbf{\textit{return}}\ r} q_7^3$ gefordert wird.

Wie anhand von Beispiel 3.12 zu sehen ist, können mit Hilfe von TGAs viele verschiedene Arten von Testzielen und Abdeckungskriterien spezifiziert werden. Im folgenden Abschnitt wird beschrieben, wie Testfälle automatisiert für Testziele abgeleitet werden können, die durch TGAs spezifiziert sind.

3.1.4 *Testfallgenerierung*

Das Ziel der automatisierten Testfallgenerierung mit Software-Model-Checking ist die Ableitung eines Testfalls (d. h. eines Eingabevektors und eines optionalen Ausgabevektors) für eine IUT, der ein gegebenes Testziel *tga* abgedeckt. Die Ableitung eines Testfalls zum Abdecken des Testziels *tga* erfolgt in zwei Schritten, wie es von Beyer et al. in [BCH+04] vorgeschlagen und in [BHTV13] um TGAs zur Testziel-getriebenen Testfallgenerierung erweitert wurde:

1. Das Finden eines Pfades durch den CFA einer IUT, der durch das Testziel *tga* akzeptiert wird und das Testziel somit abdeckt.
2. Die Ableitung von Eingabewerten und ggf. Ausgabewerten, die der *Pfadbedingung* des gefundenen CFA-Pfades entsprechen.

Für Schritt 2 konstruiert der Model-Checker einen ARG (vgl. Definition 3.9), der den erreichbaren abstrakten Programmzustandsraum der IUT repräsentiert [BCH+04]. Die Konstruktion des ARG erfolgt Testziel-getrieben für den TGA, der das Testziel spezifiziert. Testziel-getrieben bedeutet, dass möglichst nur die Teile des ARG konstruiert werden, die für die Abdeckung des Testziels relevant sein könnten. Um den aktuellen Zustand eines TGAs während der Testziel-getriebenen ARG-Konstruktion für die spätere Testfallableitung zu verfolgen, werden abstrakte Programmzustände im ARG um Informationen über den aktuellen Zustand des TGA ergänzt. Ein ARG, der für jeden Programmzustand speichert, welche Programm-Location mit welchen Variablenbelegungen erreichbar ist und in welchem Zustand sich ein TGA befindet, wird im Folgenden ARG_T genannt.

Definition 3.13 (ARG mit Testzielinformationen (ARG_T))**.** Ein ARG_T zu einem CFA (L, ℓ_0, ℓ_t, E) für einen TGA $(Q, \Sigma, \Delta, q_0, F)$ ist ein Tripel (S_T, s_{0_T}, T_T), wobei

- $S_T \subseteq L \times Q \times R(V)$ eine Menge von abstrakten Programmzuständen,
- $s_{0_T} = (\ell_0, q_0, \mathit{true}) \in S_T$ der initiale abstrakte Programmzustand und
- $T_T \subseteq S_T \times E \times S_T$ eine beschriftete Transitionsrelation mit $(\ell, q, r) \xrightarrow{e} (\ell', q', r') \in T_T$, wenn $e = (\ell, a, \ell') \in E$, $(q, e, q') \in \Delta$ und $r' \models sp_V(r, a)$ gelten,

ist.

Ein Testziel $tga = (Q, \Sigma, \Delta, q_0, F)$ wird von einem ARG_T-Pfad

$$\omega = (l_0, q_0, r_0) \xrightarrow{e_0} (l_1, q_1, r_1) \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} (l_k, q_k, r_k)$$

abgedeckt, wenn $q_k \in F$ gilt, d. h. der entsprechende TGA im letzten Zustand von ω einen Zielzustand erreicht hat. Entsprechend akzeptiert ein Testziel *tga* einen ARG_T-Pfad ω, wenn ω einer konkreten Ausführung σ entspricht, die von dem TGA *tga* akzeptiert wird. Gibt es im vollständigen ARG_T keinen Pfad, der ein Testziel abdeckt, nennen wir das Testziel *unerfüllbar* oder *unbekannt*.

Beispiel 3.14 (ARG_T)**.** Das Testziel tga_7 aus Abbildung 3.5 für die C-Funktion GaussianSum fordert die Traversierung der Anweisungen $i = x; n = y$, gefolgt von mindestens einem Schleifendurchlauf sowie der Return-Anweisung.

Abbildung 3.6 zeigt einen Ausschnitt des ARG_T der Funktion GaussianSum, der für das Testziel tga_7 konstruiert wurde. Jeder Programmzustand besteht aus einer Programm-Location, dem aktuellen Zustand des TGAs tga_7 und einer Pfadbedingung, d. h. einer Region, die beschreibt mit welchen konkreten Variablenbelegungen der Programmzustand erreichbar ist. Programmzustände mit einem gestrichelten Rahmen haben noch weitere mögliche nachfolgende Programmzustände, die jedoch noch nicht konstruiert wurden. Beispielsweise könnte gemäß des CFAs der GaussianSum-Funktion auf den Programmzustand 10′ noch Programmzustand mit der Programm-Location 11 folgen.

Auf Basis des CFAs und des ARG_T einer IUT kann, wie oben beschrieben, ein Testfall $tc = (I, O)$ (vgl. Definition 2.1) für ein Testziel in einem Zweischrittverfahren abgeleitet werden. Zuerst wird ein Pfad im CFA gesucht, der das Testziel abdeckt. Danach werden anhand der Pfadbedingung des letzten CFA-Zustands im CFA-Pfad, die dem entsprechenden Programmzustand aus dem ARG_T entstammt, Eingabewerte und ggf. Ausgabewerte abgeleitet, die als Testfall dienen.

Beispiel 3.15 (Testfall aus einem ARG_T ableiten)**.** Es soll ein konkreter Testfall abgeleitet werden, der das Testziel tga_7 auf der Funktion GaussianSum abdeckt. Die Return-Anweisung im TGA tga_7 fordert die Abdeckung des Testziels tga_7 durch eine vollständige Programmausführung, die es erlaubt, neben dem Eingabevektor ein Testorakel abzuleiten.

Als erstes wird ein CFA-Pfad gesucht, der tga_7 abdeckt, z. B. der CFA-Pfad $2 \to 4 \to 5 \to 10 \to 11 \to 12 \to 10 \to 15 \to 16$. Danach wird aus dem zum CFA-Pfad gehörigen ARG_T-Pfad ($2 - 4 - 5 - 10 - 11 - 12 - 10'' - 15' - 16'$) des ARG_T aus Abbildung 3.6 ein konkreter Testfall abgeleitet. Zur Berechnung konkreter Ein- und Ausgabewerte für den Testfall wird eine Lösung für die Pfadbedingung des ARG_T-Zustands 16′ berechnet, in dem der TGA tga_7 in einem Endzustand ist, d. h. von der Pfadbedingung

$$r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0 \wedge i_0 \leq n_0$$
$$\wedge\, r_1 = r_0 + i_0 \wedge\ i_1 = i_0 + 1 \wedge i_1 > n_0.$$

Für die Ableitung der Werte der Eingabevariablen werden die Werte der zugehörigen Variablen mit dem kleinsten SSA-Index und für Ausgabevariablen die Werte der Variablen mit dem größten SSA-Index verwendet. In diesem Beispiel bedeutet das, dass für die Eingabevariablen x und y die Werte von x_0 und y_0

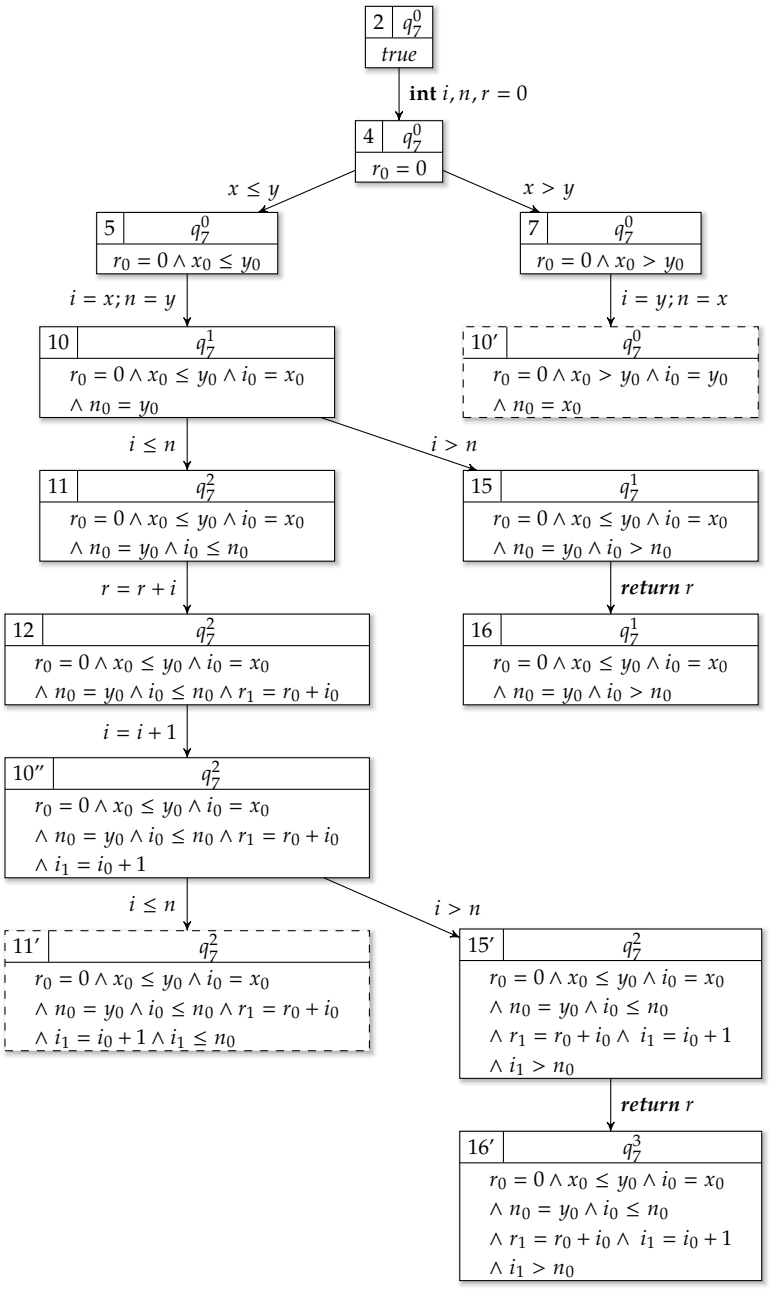


Abbildung 3.6: Ausschnitt des ARG_T der GaussianSum-Funktion für das Testziel tga_7

und für die Ausgabevariable r der Wert von r_1 zur Ableitung des konkreten Testfalls verwendet werden. Ein konkreter Testfall ist somit $([x = 2, y = 2], [r = 2])$. Die Berechnung der konkreten Werte kann dabei mit Hilfe eines SMT-Solvers durchgeführt werden.

Zur Automatisierung der Testfallgenerierung wird ein Algorithmus zur Konstruktion eines ARG_T für eine IUT benötigt, aus dem Testfälle abgeleitet werden können. Im folgenden Abschnitt werden wir einen solchen Algorithmus beschreiben.

3.1.5 *Algorithmus zur ARG_T-Konstruktion*

In diesem Abschnitt beschreiben wir die Umsetzung eines Algorithmus zur Konstruktion eines ARG_T basierend auf einem CFA. Diesen Algorithmus verwenden wir für eine Erreichbarkeitsanalyse, d. h. um zu prüfen, ob ein Testziel im Programmzustandsraum einer IUT (un-)erfüllbar ist. Für die Umsetzung der Erreichbarkeitsanalyse verwenden wir das Model-Checking-Framework *Configurable Program Analysis* (CPA) [BHT07]. In diesem Framework spezifiziert eine CPA die *abstrakte Domäne* der Programmanalyse und somit die Komponenten eines Programmzustands. Weiterhin spezifiziert eine CPA eine Menge von *Operationen* auf der abstrakten Domäne, welche die Programmanalyse steuern und somit auch die Konstruktion des Programmzustandsraums (eines ARGs).

Das Framework beinhaltet eine Reihe von kombinierbaren Standard-CPAs, z. B. die Location-CPA [BHT08]. Die Location-CPA arbeitet auf der Domäne der Programm-Locations und ist für die Traversierung eines CFAs verantwortlich. Die Traversierung eines CFAs wird durch die Operationen der Location-CPA gesteuert. Ein anderes Beispiel für eine Standard-CPA ist die Prädikaten-CPA [BHT08]. Die Prädikaten-CPA speichert die Werte von Programmvariablen symbolisch durch Prädikate. Prädikate werden als Regionen dargestellt und bilden die abstrakte Domäne der Prädikaten-CPA. Die Operationen der Prädikaten-CPA sorgen für einen korrekten Umgang mit diesen Regionen bei einem Übergang von einem Programmzustand zu einem anderen Programmzustand. Die Komposition der Location-CPA und der Prädikaten-CPA kann zur Konstruktion eines ARGs für einen CFA verwendet werden. Für die Konstruktion eines ARGs traversiert die Location-CPA einen CFA und währenddessen werden durch die Prädikaten-CPA die gültigen Variablenbelegungen berechnet, mit denen eine Programm-Location erreicht werden kann. Das Ergebnis der ARG-Konstruktion für den CFA der GaussianSum-Funktion aus Abbildung 3.2, geleitet durch die Location-CPA in Kombination mit der Prädikaten-CPA, entspricht dem ARG aus Abbildung 3.4.

Eine CPA besteht aus vier Komponenten. Die erste Komponenten ist die abstrakte Domäne $D = (C, \Psi, [\![\cdot]\!])$, welche die Programmzustände beschreibt auf der die CPA arbeitet. Dabei ist C eine Menge von konkreten Programmzuständen, Ψ die Menge von abstrakten Programmzuständen und $[\![\cdot]\!] : \Psi \to 2^C$ eine Konkretisierungsfunktion, die jedes Element $\psi \in \Psi$ auf eine Menge von konkreten Zuständen $[\![\psi]\!]$ abbildet. Die zweite Komponente ist eine Transferrelation $\rightsquigarrow$, welche die abstrakten Nachfolgeprogrammzustände eines Programmzustands berechnet. Die dritte Komponente ist ein Merge-Operator *merge*, der spezifiziert, ob und wie zwei abstrakte Programmzustände vereinigt werden müssen, wenn der Kontrollfluss zusammenläuft. Die vierte Komponente ist ein Stopp-Operator *stop*, der spezifiziert, ob ein abstrakter Programmzustand durch einen anderen abgedeckt wird und somit nicht weiter analysiert werden muss. Eine CPA ist wie folgt definiert.

Definition 3.16 (Configurable Program Analysis (CPA) [BHT08])**.** Eine CPA $(D, \rightsquigarrow, \mathit{merge}, \mathit{stop})$ besteht aus den folgenden vier Komponenten:

1. Abstrakte Domäne $D = (C, \Psi, [\![\cdot]\!])$

2. Abstrakte Transferrelation $\rightsquigarrow \subseteq \Psi \times \Psi$
3. Merge-Operator $merge : \Psi \times \Psi \rightarrow \Psi$
4. Stopp-Operator $stop : \Psi \times 2^{\Psi} \rightarrow \mathbb{B}$

Für die Konstruktion eines ARG_T zur Testfallgenerierung, wie er beispielsweise in Abbildung 3.6 zu sehen ist, verwenden wir in dieser Arbeit eine komponierte CPA $CPA_{\mathbb{T}}$ bestehend aus drei CPAs, die während der Traversierung eines gegebenen CFAs den zugehörigen ARG_T für einen TGA aufbauen:

- Location-CPA $CPA_{\mathbb{L}}$: Steuert die Traversierung eines CFAs, d. h. die CPA analysiert den Kontrollfluss.
- Prädikaten-CPA $CPA_{\mathbb{P}}$: Prädikatenabstraktionsbasierte Analyse zur Berechnung von Pfadbedingungen, d. h. die Prädikate sind in diesem Fall Pfadbedingungen. Somit analysiert die CPA den Datenfluss.
- Automaten-CPA $CPA_{\mathbb{A}}$: Automatengerichtete Analyse zur Behandlung von TGAs, d. h. die CPA analysiert die Testabdeckung.

Die drei CPAs berechnen jeweils einen Teil des Programmzustands eines ARG_T. Für ein vollständiges Programm, das aus mehreren Funktionen und Datentypen wie Pointern bestehen kann, würden die drei CPAs mit weiteren CPAs kombiniert werden, z. B. einer Callstack-CPA, die Funktionsaufrufe behandelt, oder einer Function-Pointer-CPA, die Function-Pointer analysiert [BHT08]. Im Folgenden werden die Funktionsweisen der drei CPAs $CPA_{\mathbb{L}}$, $CPA_{\mathbb{P}}$ und $CPA_{\mathbb{A}}$ kurz beschrieben. Darauf aufbauend wird ein Algorithmus beschrieben, der mit Hilfe der drei CPAs einen ARG_T konstruiert.

LOCATION-CPA. Die Location-CPA $CPA_{\mathbb{L}} = (D_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}}, merge_{\mathbb{L}}, stop_{\mathbb{L}})$ arbeitet auf der Domäne der Programm-Locations L. Sowohl konkrete als auch abstrakte Programmzustände sind einzelne Programm-Locations, d. h. es wird nicht abstrahiert. Die Transferrelation $\rightsquigarrow_{\mathbb{L}}$ berechnet die Nachfolger einer Programm-Location. Somit enthält $\rightsquigarrow_{\mathbb{L}}$ den Programmzustandsübergang $\ell \rightsquigarrow_{\mathbb{L}} \ell'$, wenn es eine entsprechende CFA-Kante $(\ell, a, \ell') \in E$ gibt. Der Merge-Operator $merge_{\mathbb{L}}$ vereinigt zwei Programmzustände immer dann, wenn die Programmzustände die gleiche Programm-Location enthalten. Dabei prüft der Merge-Operator, ob eine neue Programm-Location ℓ' einer bereits traversierten Programm-Location ℓ entspricht. Sind die beiden Programm-Locations ℓ' und ℓ gleich, wird die bereits traversierte Programm-Location ℓ zurückgegeben und sonst die neue Programm-Location ℓ'. Der Stopp-Operator $stop_{\mathbb{L}}$ prüft für jeden Programmzustand einzeln, ob die Programm-Location des Programmszustands während der Analyse bereits erreicht wurde, d. h. ein Programmzustand mit der Programm-Location bereits in der Menge der erreichten Programmzustande *reached* enthalten ist. Ist dies der Fall, wird die Analyse dieses Programmszustands abgebrochen und ein anderer Programmzustand wird analysiert.

Definition 3.17 (Location-CPA $\text{CPA}_{\mathbb{L}}$ (basierend auf [BHT08]))**.** Sei (L, ℓ_0, ℓ_t, E) ein CFA. Die zugehörige Location-CPA $\text{CPA}_{\mathbb{L}} = (D_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}}, \mathit{merge}_{\mathbb{L}}, \mathit{stop}_{\mathbb{L}})$ besteht aus den folgenden vier Komponenten:

1. $D_{\mathbb{L}} = (C_{\mathbb{L}}, \Psi_{\mathbb{L}}, [\![\cdot]\!]_{\mathbb{L}})$ besteht aus der Menge der konkreten Zustände $C_{\mathbb{L}} = L$, der Menge der abstrakten Zustände $\Psi_{\mathbb{L}} = L$ und einer Konkretisierungsfunktion $[\![\cdot]\!]_{\mathbb{L}} : C_{\mathbb{L}} \to \Psi_{\mathbb{L}}$, die der Identitätsfunktion entspricht.
2. Die Transferrelation $\rightsquigarrow_{\mathbb{L}}$ enthält den Programmzustandsübergang $\ell \rightsquigarrow_{\mathbb{L}} \ell'$, wenn es eine entsprechende CFA-Kante $(\ell, a, \ell') \in E$ gibt.
3. $\mathit{merge}_{\mathbb{L}}(\ell, \ell') = \begin{cases} \ell & \text{, wenn } \ell = \ell' \\ \ell' & \text{sonst} \end{cases}$
4. $\mathit{stop}_{\mathbb{L}}(\ell, \mathit{reached}) \Leftrightarrow (\exists \ell' \in \mathit{reached} : \ell = \ell')$

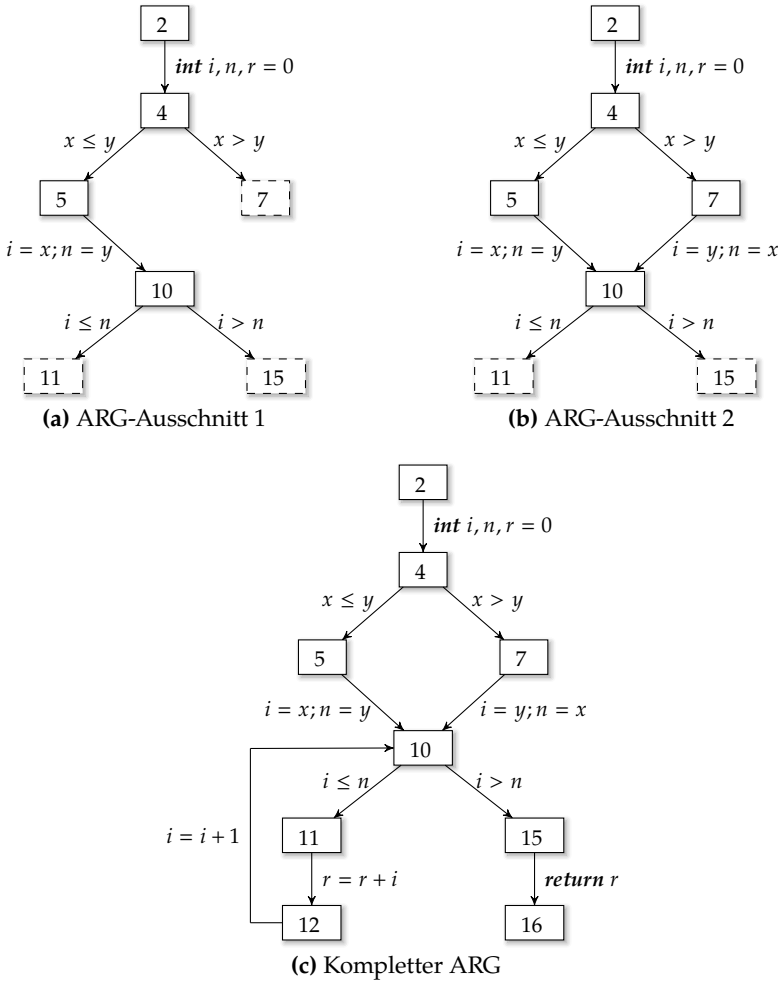
Beispiel 3.18 (Location-CPA)**.** Abbildung 3.7a zeigt einen Ausschnitt des ARGs, der gemäß der Location-CPA $\text{CPA}_{\mathbb{L}}$ für den CFA (L, l_0, l_t, E) der GaussianSum-Funktion aus Abbildung 3.2 konstruiert wurde. Gemäß der Domäne $D_{\mathbb{L}}$ der Location-CPA ist jeder Programmzustand mit einer Programm-Location beschriftet. Die Nachfolgezustände der Programmzustände werden gemäß der Transitionen E des CFAs berechnet, d. h. wenn es eine Transition $(l, a, l') \in E$ gibt, dann gibt es eine entsprechende Transition $l \rightsquigarrow_{\mathbb{L}} l'$ im ARG. Beispielsweise hat der CFA-Knoten 4 zwei ausgehenden Transitionen $4 \xrightarrow{x \leq y} 5$ und $4 \xrightarrow{x > y} 7$. Dementsprechend werden für den Programmzustand 4 zwei nachfolgende Programmzustände im ARG mit den Programm-Locations 5 bzw. 7 berechnet.

Abbildung 3.7b demonstriert eine weitere Stufe der ARG-Konstruktion. Der CFA-Knoten 7 hat eine ausgehende Transition $7 \xrightarrow{i = n; n = x} 10$ zum CFA-Knoten 10. Ein Programmzustand mit der Programm-Location 10 wurde bereits als Nachfolger des Programmzustands 5 berechnet. Dies führt zu zwei Aktionen:

1. Gemäß des Merge-Operators $\mathit{merge}_{\mathbb{L}}$ können die zwei Programmzustände mit der Programm-Location 10, die als Nachfolger der Programmzustände 5 und 7 berechnet wurden, vereinigt werden, da die Programmzustände die gleiche Programm-Location referenzieren.
2. Da sich bereits ein Programmzustand mit der Programm-Location 10 in der Menge der erreichten Programmzustände *reached* befindet, muss der neu berechnete Programmzustand mit der Programm-Location 10 gemäß des Stopp-Operators $\mathit{stop}_{\mathbb{L}}$ nicht weiter analysiert werden.

Der komplette ARG ist in Abbildung 3.7c dargestellt.

Prädikaten-CPA. Zur Berechnung der Pfadbedingung von Programmzuständen wird die Prädikaten-CPA verwendet. Die Prädikaten-CPA $\text{CPA}_{\mathbb{P}} = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}},$



(a) ARG-Ausschnitt 1 **(b)** ARG-Ausschnitt 2

(c) Kompletter ARG

Abbildung 3.7: ARGs mit Programm-Locations für die C-Funktion GaussianSum

$merge_{\mathbb{P}}, stop_{\mathbb{P}}$) arbeitet auf der Domäne der Menge von Regionen $R(V)$ über den Programmvariablen V. Die Transferrelation $\rightsquigarrow_{\mathbb{P}}$ zur Berechnung von abstrakten Nachfolgern wird durch die Strongest-Postcondition sp_V (vgl. Definition 3.7) angegeben. Der Merge-Operator $merge_{\mathbb{P}}$ vereinigt zwei Programmzustände durch die Disjunktion der Prädikate der zugehörigen Regionen. Der Stopp-Operator $stop_{\mathbb{P}}$ prüft für jeden Programmzustand, ob seine Region bereits in der Region eines anderen bereits erreichten Programmzustands aus der Menge der erreichten Programmzustände *reached* enthalten ist. Ist das der Fall, wird der Programmzustand nicht weiter analysiert.

Definition 3.19 (Prädikaten-CPA $\text{CPA}_{\mathbb{P}}$ (basierend auf [Wen17]))**.** Sei (L, ℓ_0, ℓ_t, E) ein CFA. Die zugehörige Prädikaten-CPA $\text{CPA}_{\mathbb{P}} = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, merge_{\mathbb{P}}, stop_{\mathbb{P}})$ besteht aus den folgenden vier Komponenten:

1. $D_{\mathbb{P}} = (C_{\mathbb{P}}, \Psi_{\mathbb{P}}, [\![\cdot]\!]_{\mathbb{P}})$ besteht aus der Menge der konkreten Zustände $C_{\mathbb{P}}$, die konkrete Programmzustände $C : V \to \mathcal{D}$ repräsentieren, der Menge der abstrakten Programmzustände $\Psi_{\mathbb{P}} = R(V)$, die Mengen von

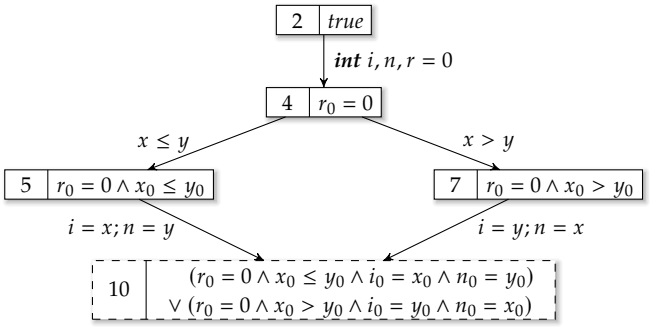


Abbildung 3.8: Ausschnitt des ARGs der C-Funktion GaussianSum mit der Vereinigung von Programmzuständen

konkreten Variablenbelegungen charakterisieren und einer Konkretisierungsfunktion $[\![\cdot]\!]_{\mathbb{P}} : C_{\mathbb{P}} \to \Psi_{\mathbb{P}}$, die einen abstrakten Zustand ψ auf eine Menge von konkreten Zuständen $[\![\psi]\!]_{\mathbb{P}}$ abbildet.

2. Sei $e = (\ell, a, \ell') \in E$ eine CFA-Kante, dann enthält die Transferrelation $\rightsquigarrow_{\mathbb{P}}$ den Programmzustandsübergang $\psi \rightsquigarrow_{\mathbb{P}} \psi'$, wenn $\psi' \Leftrightarrow sp_V(\psi, a)$ gilt.

3. $merge_{\mathbb{P}}(\psi, \psi') = \psi \vee \psi'$

4. $stop_{\mathbb{P}}(\psi, reached) \Leftrightarrow (\exists \psi' \in reached : \psi' \models \psi)$

Zur Berechnung von Pfadbedingungen entlang von CFA-Pfaden wird die Prädikaten-CPA mit der Location-CPA komponiert. Die Location-CPA traversiert einen CFA und die Prädikaten-CPA berechnet die Pfadbedingungen, welche die Variablenbelegungen charakterisieren mit denen die Programm-Locations jeweils erreichbar sind. Die Komposition der CPAs arbeitet auf der komponierten Domäne $D_{\mathbb{L}} \times D_{\mathbb{P}}$. Programmzustände werden immer dann vereinigt, wenn die beiden Merge-Operatoren $merge_{\mathbb{L}}$ und $merge_{\mathbb{P}}$ eine Vereinigung auf den jeweiligen Komponenten der Programmzustände erlauben und gestoppt wird. Die Analyse wird für einen Programmzustand gestoppt, wenn die beiden Stopp-Operatoren $stop_{\mathbb{L}}$ und $stop_{\mathbb{P}}$ *true* zurückliefern.

Beispiel 3.20 (Prädikaten-CPA)**.** Abbildung 3.8 zeigt einen Ausschnitt eines ARGs, der gemäß der Komposition der Location-CPA und der Prädikaten-CPA für den CFA der GaussianSum-Funktion aus Abbildung 3.2 konstruiert wurde. Die Programmzustände enthalten Paare, bestehend aus Programm-Locations und Pfadbedingungen.

Die komponierte Transferrelation berechnet die Nachfolger eines Programmzustands gemäß der Komposition der Transferrelationen $\rightsquigarrow_{\mathbb{L}}$ und $\rightsquigarrow_{\mathbb{P}}$. Ausgehend von Programmzustand $(4, r_0 = 0)$ berechnet die Transferrelation $\rightsquigarrow_{\mathbb{L}}$ die zwei nachfolgendende Programm-Locations 5 und 7 im CFA. Die Transferrelation $\rightsquigarrow_{\mathbb{P}}$ der Prädikaten-CPA verfeinert die Pfadbedingung $r_0 = 0$ des Programmzustand $(4, r_0 = 0)$ gemäß der Operationen an den CFA-Kanten $4 \xrightarrow{x \leq y} 5$ und $4 \xrightarrow{x > y} 7$ zu $r_0 = 0 \wedge x_0 \leq y_0$ bzw. $r_0 = 0 \wedge x_0 > y_0$. Zusammengenom-

men ergibt das für den Programmzustand $(4, r_0 = 0)$ die zwei nachfolgenden Programmzustände $(5, r_0 = 0 \wedge x_0 \leq y_0)$ und $(7, r_0 = 0 \wedge x_0 > y_0)$.

Der komponierte Merge-Operator vereinigt immer dann, wenn der Kontrollfluss zusammenfließt, d. h. beispielsweise in CFA-Location 10. Die Pfadbedingung des vereinigten Programmzustands resultiert aus der Disjunktion der zu vereinigenden Programmzustände. Der nachfolgende Programmzustand von Programmzustand $(5, r_0 = 0 \wedge x_0 \leq y_0)$ ist $(10, r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0)$ und der Nachfolger von Programmzustand $(7, r_0 = 0 \wedge x_0 > y_0)$ ist $(10, r_0 = 0 \wedge x_0 > y_0 \wedge i_0 = y_0 \wedge n_0 = x_0)$. Da in den nachfolgenden Programmzuständen von $(5, r_0 = 0 \wedge x_0 \leq y_0)$ und $(7, r_0 = 0 \wedge x_0 > y_0)$ der Kontrollfluss zusammenfließt, können sie vereinigt werden. Das Ergebnis ist der abstrakte Programmzustand $(10, (r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0) \vee (r_0 = 0 \wedge x_0 > y_0 \wedge i_0 = y_0 \wedge n_0 = x_0))$.

Automaten-CPA. Die CPA $\text{CPA}_{\mathbb{A}} = (D_{\mathbb{A}}, \rightsquigarrow_{\mathbb{A}}, \mathit{merge}_{\mathbb{A}}, \mathit{stop}_{\mathbb{A}})$, definiert von Holzer [Hol13], berechnet die TGA-Zustände in welchen sich ein TGA $A = (Q, \Sigma, \Delta, q_0, F)$ beim Erreichen eines Programmzustands befindet. Dabei arbeitet die Automaten-CPA auf der Domäne $D_{\mathbb{A}}$ der Zustände Q eines TGAs und schaltet immer dann, wenn eine CFA-Kante traversiert wird, die einen Zustandsübergang im TGA erlaubt. Da wir fordern, dass jeder TGA-Zustand eine mit * beschriftete Selbsttransition hat, kann die Automaten-CPA immer schalten. Eine Vereinigung von Programmzuständen ist gemäß des Merge-Operators $\mathit{merge}_{\mathbb{A}}$ immer dann erlaubt, wenn die Programmzustände den gleichen TGA-Zustand referenzieren. Der Stopp-Operator $\mathit{stop}_{\mathbb{A}}$ überprüft für jeden Programmzustand, ob es einen anderen Programmzustand gibt, der bereits den gleichen TGA-Zustand referenziert. Ist dies der Fall gibt er *true* zurück und die Analyse wird gestoppt.

Definition 3.21 (Automaten-CPA $\text{CPA}_{\mathbb{A}}$ (basierend auf [Hol13]))**.** Seien (L, ℓ_0, ℓ_t, E) ein CFA und $A = (Q, \Sigma, \Delta, q_0, F)$ ein TGA mit dem Alphabet $\Sigma \subseteq E \times R$. Die zugehörige Automaten-CPA $\text{CPA}_{\mathbb{A}} = (D_{\mathbb{A}}, \rightsquigarrow_{\mathbb{A}}, \mathit{merge}_{\mathbb{A}}, \mathit{stop}_{\mathbb{A}})$ besteht aus den folgenden vier Komponenten:

- $D_{\mathbb{A}} = (C_{\mathbb{A}}, \Psi_{\mathbb{A}}, [\![\cdot]\!]_{\mathbb{A}})$ besteht aus einer Menge von konkreten Zuständen $C_{\mathbb{A}} = Q$, einer Menge von abstrakten Zuständen $\Psi_{\mathbb{A}} = Q$ und einer Konkretisierungsfunktion $[\![\cdot]\!]_{\mathbb{A}} : C_{\mathbb{Q}} \to \Psi_{\mathbb{Q}}$, die der Identitätsfunktion entspricht.
- Sei $e \in E$ eine CFA-Kante, dann enthält die Transferrelation $\rightsquigarrow_{\mathbb{A}}$ den Programmzustandsübergang $q \rightsquigarrow_{\mathbb{A}} q'$, wenn der TGA A eine Transition $(q, e, q') \in \Delta$ enthält.
- $\mathit{merge}_{\mathbb{A}}(q, q') = \begin{cases} q & \text{, wenn } q = q' \\ q' & \text{sonst} \end{cases}$
- $\mathit{stop}_{\mathbb{A}}(q, \mathit{reached}) = (\exists q' \in \mathit{reached} : q = q')$

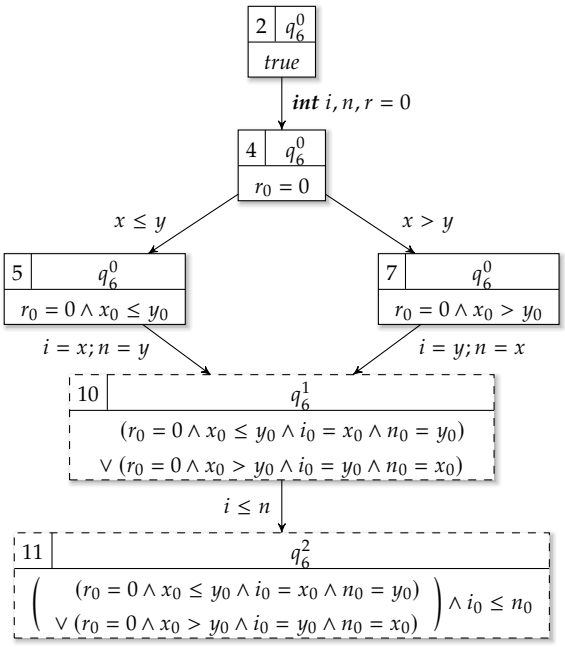


Abbildung 3.9: Ausschnitt des ARG_T der C-Funktion GaussianSum mit der Vereinigung von Programmzuständen für das Testziel tga_6

Die aus der Location-CPA $CPA_{\mathbb{L}}$, der Prädikaten-CPA $CPA_{\mathbb{P}}$ und der Automaten-CPA $CPA_{\mathbb{A}}$ komponierte CPA $CPA_{\mathbb{T}}$ führt eine automatengetriebene Programm-Analyse zur Testfallgenerierung für die Abdeckung eines Testziels *tga* durch, die bei der ARG_T-Konstruktion

- die Pfade traversiert, die für den TGA *tga* relevant sein können und
- terminiert, wenn die Analyse einen ARG_T-Pfad findet, der durch den TGA *tga* akzeptiert wird, d. h. das Testziel *tga* abdeckt.

Die CPA $CPA_{\mathbb{T}}$ arbeitet auf der komponierten Domäne $D_{\mathbb{L}} \times D_{\mathbb{A}} \times D_{\mathbb{P}}$. Programmzustände werden immer dann vereinigt, wenn alle drei Merge-Operatoren $merge_{\mathbb{L}}$, $merge_{\mathbb{A}}$ und $merge_{\mathbb{P}}$ eine Vereinigung auf den jeweiligen Komponenten der Programmzustände erlauben. Die CPA $CPA_{\mathbb{T}}$ stoppt die Analyse für einen Programmzustand, wenn alle drei Stopp-Operatoren $stop_{\mathbb{L}}$, $stop_{\mathbb{A}}$ und $stop_{\mathbb{P}}$ *true* zurückliefern.

Beispiel 3.22 ($CPA_{\mathbb{T}}$)**.** Ein ARG_T für die GaussianSum-Funktion, der gemäß der CPA $CPA_{\mathbb{T}}$ für den TGA tga_6 aus Abbildung 3.5 konstruiert wurde, ist in Abbildung 3.9 dargestellt. Die Konstruktion des ARG_T erfolgte ähnlich zur Konstruktion des ARG aus Abbildung 3.8 aus Beispiel 3.20. Der Unterschied ist, dass jeder Programmzustand gemäß der Domäne $D_{\mathbb{T}} = D_{\mathbb{L}} \times D_{\mathbb{A}} \times D_{\mathbb{P}}$ zusätzlich zur Programm-Location und der Pfadbedingung den aktuellen TGA-Zustand von tga_6 enthält. In Programmzustand 10 läuft der Kontrollfluss gemäß des CFA der GaussianSum-Funktion zusammen und Programmzustände werden vereinigt. Die Vereinigung ist gemäß der Definition des Merge-Operators der Automaten-CPA $merge_{\mathbb{A}}$ nur erlaubt, da sich der TGA tga_6 in Programmzustand

10 für beide zu vereinigende Kontrollflüsse im TGA-Zustand q_6^1 befindet, d. h. im gleichen TGA-Zustand.

In dem ARG_T aus Abbildung 3.6, der für den TGA tga_7 für die GAUSSIAN-SUM-Funktion erstellt wurde, wurden die beiden Programmzustände 10 und 10′ nicht vereinigt. Sowohl der Merge-Operator $merge_{\mathbb{L}}$ der Location-CPA als auch der Merge-Operator $merge_{\mathbb{P}}$ der Prädikaten-CPA würden das Vereinigen der Programmzustände 10 und 10′ erlauben. Da sich der TGA tga_7 jedoch in den beiden Programmzuständen in unterschiedlichen TGA-Zuständen befinden (q_7^1 in Programmzustand 10 und q_7^0 in Programmzustand 10′), dürfen die Programmzustände 10 und 10′ gemäß des Merge-Operators $merge_{\mathbb{A}}$ der Automaten-CPA nicht vereinigt werden.

Die komponierte CPA $\text{CPA}_\mathbb{T}$ kann als Eingabe für einen generischen Algorithmus zur Konstruktion eines ARG_T verwendet werden. Der Algorithmus von Beyer et al. ist Teil des Model-Checking-Frameworks *CPA* und konstruiert, basierend auf einer CPA und einem CFA, einen ARG_T [BHT08]. Die CPA bestimmt die Domäne der Programmzustände des ARG_T und der Algorithmus steuert die Konstruktion des ARG_T anhand der Operatoren der übergebenen (komponierten) CPA. Wir werden den Algorithmus im Folgenden zur Testziel-getriebenen Konstruktion eines ARG_T verwenden, um dann aus dem ARG_T Pfade abzuleiten, die als abstrakte Testfälle verwendet werden können. Dieses Vorgehen zur Testfallableitung wurde bereits von Holzer in [Hol13] beschrieben.

Algorithmus 1 zeigt den Pseudo-Code zur ARG_T-Konstruktion. Der Algorithmus verwendet zwei Datenstrukturen, die Mengen bestehend aus abstrakten Programmzuständen Ψ enthalten. Die Menge *reached* enthält alle während der ARG_T-Konstruktion bereits erreichten Programmzustände und die Menge *waitlist* enthält alle noch abzuarbeitenden Programmzustände. Als Eingabe erhält der Algorithmus eine CPA sowie eine Menge $reached_0$ zur Initialisierung der Menge *reached* und eine Menge $waitlist_0 \subseteq reached_0$ zur Initialisierung der Menge *waitlist*. Durch die Übergabe der Mengen $reached_0$ und $waitlist_0$ als Eingabe an den Algorithmus, kann eine ARG_T-Konstruktion an einer beliebigen Stelle eines ARG_T begonnen oder fortgesetzt werden. Der Algorithmus konstruiert einen ARG_T solange, bis die Menge *waitlist* leer ist und es keine weiteren Programmzustände mehr gibt, die weitere noch nicht konstruierte nachfolgende Programmzustände haben (für die Testfallgenerierung heißt das, dass ein Testziel unerfüllbar ist) oder bis ein TGA in einem Endzustand ist und somit das entsprechende Testziel abgedeckt wurde.

Für die Konstruktion des Programmzustandsraums initialisiert der Algorithmus zuerst die Mengen *reached* und *waitlist* mit den Mengen $reached_0$ und $waitlist_0$, die bei einer komplett neuen ARG_T-Konstruktion nur den initialen abstrakten Programmzustand $\psi_0 = (\ell_0, q_0, r_0)$ enthalten (Zeile 1–2). Solange die Menge *waitlist* nicht leer ist (Zeile 3), wählt der Algorithmus den nächsten zu betrachtenden abstrakten Programmzustand ψ aus der Menge *waitlist* aus und entfernt ψ aus der Menge *waitlist* (Zeile 4).

Für jeden abstrakten Nachfolgeprogrammzustand ψ' von ψ (Zeile 5) wird überprüft, ob es einen abstrakten Programmzustand ψ'' aus der Menge der erreichbaren Programmzustände *reached* gibt, mit dem ψ' vereinigt werden kann (Zeile 6–7). Ist dies der Fall, wird ein neuer, vereinigter, abstrakter Programmzustand ψ_{new} erstellt,

Algorithmus 1 Erreichbarkeitsanalyse-Algorithmus (basierend auf [BHT08])

Input: Eine CPA $(D, \rightsquigarrow, merge, stop)$,
eine initiale Menge erreichbarer abstrakter Zustände $reached_0$,
eine initiale Menge von nicht analysierten Zuständen $waitlist_0 \subseteq reached_0$
Output: Eine Menge erreichbarer abstrakter Zustände *reached*,
eine Menge von nicht analysierten abstrakten Zuständen *waitlist*

```
1:  reached := reached_0 // Menge erreichter abstrakter Zustände
2:  waitlist := waitlist_0 // Menge noch abzuarbeitender Zustände
3:  while waitlist ≠ ∅ do
4:    pop(ψ) from waitlist
5:    for each ψ' with ψ ⇝ ψ' do
6:      for each ψ'' ∈ reached do
7:        ψ_new := merge(ψ', ψ'') // Vereinigung mit existierenden Zuständen
8:        if ψ_new ≠ ψ'' then
9:          waitlist := (waitlist ∪ {ψ_new}) \ {ψ''}
10:         reached := (reached ∪ {ψ_new}) \ {ψ''}
11:     // Füge neuen abstrakten Zustand hinzu
12:     if ¬stop(ψ', reached) then
13:       waitlist := waitlist ∪ {ψ'}
14:       reached := reached ∪ {ψ'}
15:     if tgaIsInFinalState(ψ') then
16:       waitlist := waitlist ∪ {ψ}
17:       return (reached, waitlist)
18: return (reached, ∅)
```

der die beiden Zustände ψ' und ψ'' enthält, und den Programmzustand ψ'' in den Mengen *reached* und *waitlist* ersetzt (Zeile 8–10).

Danach wird durch den Stopp-Operator sichergestellt, dass der Programmzustand ψ' nur zu den Mengen *reached* und *waitlist* hinzugefügt wird, wenn es wirklich ein neuer Programmzustand ist, d. h. das ψ' nicht bereits in einem anderen, bereits erreichten Programmzustand aus *reached* enthalten ist (Zeile 12–14). Weiterhin wird überprüft, ob der in der CPA enthaltene TGA in einem Endzustand ist und die ARG_T-Konstruktion unterbrochen werden muss (Zeile 15). Wird die ARG_T-Konstruktion unterbrochen, wird der Vorgängerzustand ψ von ψ' wieder zu der Menge *waitlist* hinzugefügt, um später ggf. weitere Nachfolger von ψ zu berechnen (Zeile 16). Weiterhin werden die Mengen *reached* und *waitlist* zurückgegeben (Zeile 17), wobei *reached* den abstrakten Testfall (einen ARG_T-Pfad) enthält, der das Testziel abdeckt. Wird die ARG_T-Konstruktion nicht unterbrochen und es gibt keinen Testfall, der das Testziel abdeckt, wird die Menge *reached* und eine leere Menge zurückgegeben (Zeile 18). Es wird eine leere Menge zurückgegeben, da es keinen Programmzustand in *waitlist* mehr gibt, der abgearbeitet werden muss. In diesem Fall ist das Testziel für die Abstraktion mit welcher der ARG_T konstruiert wurde, unerfüllbar.

Durch die Verwendung weiterer CPAs, zusätzlich zu den CPAs $\text{CPA}_\mathbb{L}$, $\text{CPA}_\mathbb{P}$ und $\text{CPA}_\mathbb{A}$, kann die Analyse um weitere Domänen erweitert werden. Enthält die IUT

beispielsweise Function-Pointer, kann die CPA $\text{CPA}_{\mathbb{T}}$ mit einer Function-Pointer-CPA [BHT08] komponiert werden, welche die Function-Pointer analysiert. Dadurch, dass Algorithmus 1 generisch ist, kann der Algorithmus auch eine komponierte CPA bestehend aus der CPA $\text{CPA}_{\mathbb{T}}$ und einer Function-Pointer-CPA behandeln.

Auf Basis der in diesem und den vorherigen Abschnitten beschrieben Ansätze werden wir im folgenden Abschnitt einen Algorithmus beschreiben, der nicht nur einen Testfall für ein einzelnes Testziel generiert, sondern eine Test-Suite für eine Menge von Testzielen, die beispielsweise durch ein Abdeckungskriterium definiert wird.

3.1.6 *Test-Suite-Generierung*

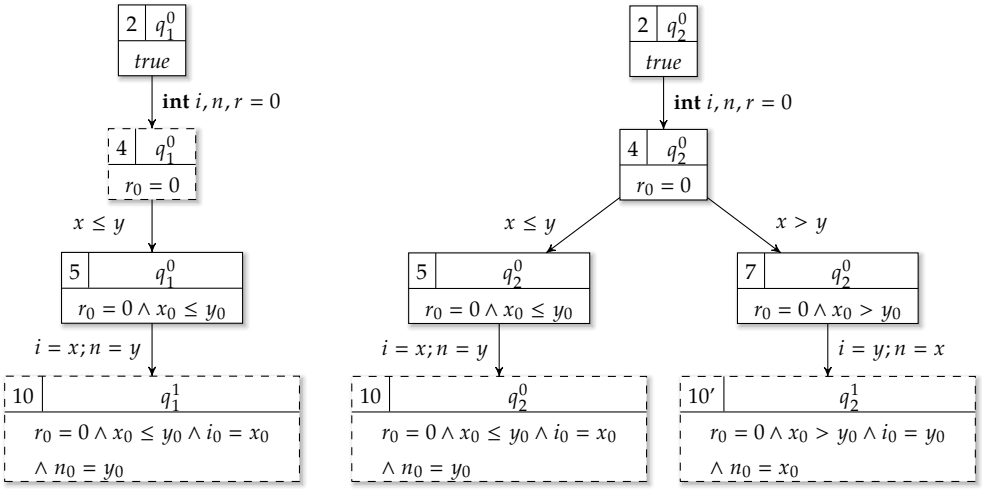
Zur Generierung einer Test-Suite für eine Menge von Testzielen für eine IUT, die z. B. durch ein Abdeckungskriterium definiert sein können, iteriert ein Testfallgenerator über die Menge von Testzielen und leitet für jedes der Testziele einen Testfall gemäß der Verfahren aus den Abschnitten 3.1.4 und 3.1.5 ab. Die zugrundeliegende Idee dafür wurde bereits von Holzer in [Hol13] beschrieben.

Beispiel 3.23 (White-Box Testgenerierung)**.** Es soll eine Test-Suite zur Abdeckung der Testzielmenge $\{tga_1, tga_2, tga_3\}$ aus Abbildung 3.5 für die C-Funktion GaussianSum erzeugt werden. Der Testfallgenerator iteriert über die Menge von Testzielen und generiert für jedes Testziel einen Testfall, der dann einer Test-Suite hinzugefügt wird. Die Generierung eines Testfalls erfolgt in einem Zweischrittverfahren, wie es in Abschnitt 3.1.4 beschrieben wurde. Im ersten Schritt wird für jedes Testziel ein ARG_{T} konstruiert aus dem ein abstrakter Testfall (ein ARG_{T}-Pfad) abgeleitet werden kann, der das Testziel abdeckt. Aus dem abstrakten Testfall kann in einem zweiten Schritt ein konkreter Testfall abgeleitet werden (vgl. Beispiel 3.15).

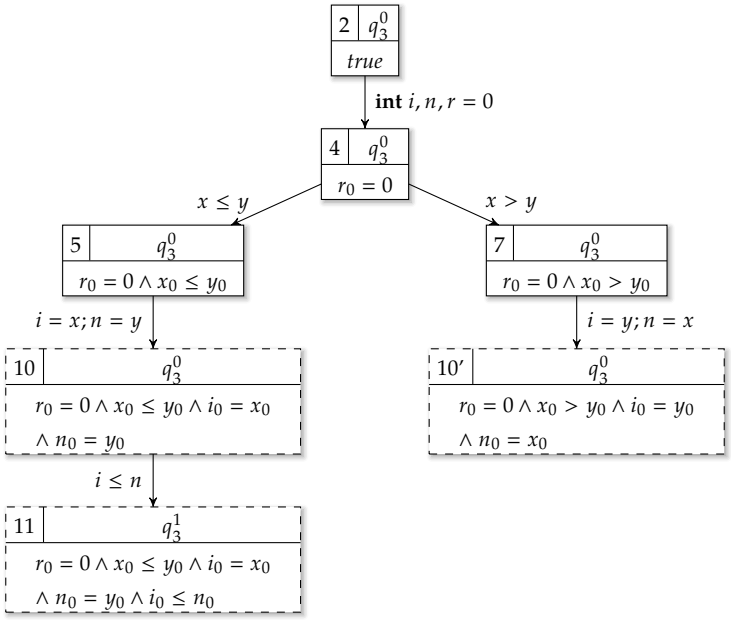
Begonnen wird beispielsweise mit der Generierung eines Testfalls für das Testziel tga_1. Im ersten Schritt wird der ARG_{T} für den TGA tga_1 konstruiert, der in Abbildung 3.10a dargestellt ist. In Programmzustand 10 des ARG_{T}-Pfades $2 - 4 - 5 - 10$ ist der TGA tga_1 in einem Endzustand und das Testziel tga_1 ist somit abgedeckt. Im zweiten Schritt kann aus der Pfadbedingung

$$r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0$$

von Programmzustand 10 ein konkreter Testfall $tc_1 = ([x = 1, y = 2])$ abgeleitet werden. Für den Testfall tc_1 kann kein Ausgabevektor aus der Pfadbedingung von Programmzustand 10 abgeleitet werden, da die C-Funktion GaussianSum in Programmzustand 10 noch nicht terminiert, d. h. es sich um eine partielle Programmausführung handelt. Danach generiert der Testfallgenerator Testfälle für die Testziele tga_2 und tga_3 auf die gleiche Art und Weise. Für das Testziel tga_2 wird der ARG_{T} aus Abbildung 3.10b konstruiert und aus der Pfadbedingung von Programmzustand 10', in dem der TGA tga_2 einen Endzustand erreicht hat, der konkrete Testfall $tc_2 = ([x = 2, y = 1])$ abgeleitet. Für das Testziel tga_3 wird der ARG_{T} aus Abbildung 3.10c konstruiert und aus der Pfadbedin-



(a) ARG_T für das Testziel tga_1 **(b)** ARG_T für das Testziel tga_2



(c) ARG_T für das Testziel tga_3

Abbildung 3.10: Ausschnitte der ARG_T für die C-Funktion GaussianSum getrieben durch die Testziele tga_1, tga_2 und tga_3

gung von Programmzustand 11 der konkrete Testfall $tc_3 = ([x = 1, y = 2])$ abgeleitet. Die generierte Test-Suite $TS_1 = \{tc_1, tc_2, tc_3\}$ deckt die Testzielmenge $\{tga_1, tga_2, tga_3\}$ für die C-Funktion GaussianSum vollständig ab.

Algorithmus 2 beschreibt das Vorgehen zur Test-Suite-Generierung, wie es in Beispiel 3.23 demonstriert wurde. Die Eingaben des Algorithmus sind ein Kontrollflussautomat *CFA* und eine Menge von n Testzielen *TG*, die als TGAs spezifiziert sind und automatisiert aus einer FQL-Query abgeleitet werden können. Die Ausgabe ist eine abstrakte Test-Suite *TS*, die alle erreichbaren Testziele abdeckt. Der Algorithmus beginnt mit der Initialisierung der Test-Suite *TS* als leere Menge (Zeile 1) und der Menge der noch nicht abgearbeiteten Testziele *UC* mit der Menge aller

Algorithmus 2 Test-Suite-Generierung

Input: Kontrollflussautomat $CFA = (L, \ell_0, \ell_t, E)$,
Menge von Testzielautomaten $TG = \{tga_1, \ldots, tga_n\}$
Output: Abstrakte Test-Suite TS

1: $TS := \{\}$
2: $UC := TG$
3: **while** $UC \neq \emptyset$ **do**
4: $tga := (Q, \Sigma, \Delta, q_0, F) \in UC$
5: **if** $\exists\ \omega := \psi_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{k-1}} \psi_k$ mit $\psi_i = (\ell_i, q_i, r_i)$ für $i \leq 0 \leq k$ und $q_k \in F$ **then**
6: $TS := TS \cup (tga, \omega)$
7: $UC := UC \setminus tga$

Testziele TG (Zeile 2). Danach werden in einer Schleife solange Testfälle generiert (Zeile 3–7), bis alle Testziele abgearbeitet wurden, d. h. jedes Testziel entweder von einem Testfall abgedeckt wird oder entweder als unerfüllbar oder als unbekannt identifiziert wurde. In der Schleife wird zuerst ein zu betrachtendes, noch nicht abgearbeitetes Testziel tga aus der Menge UC ausgewählt (Zeile 4). Für das Testziel tga wird eine Erreichbarkeitsanalyse (vgl. Algorithmus 1) durchgeführt, die überprüft, ob das Testziel tga durch einen ARG_T-Pfad ω abgedeckt werden kann, d. h. ob sich der TGA tga im letzten Programmzustand von ω in einem Endzustand befindet (Zeile 5). Wird das Testziel tga von ω abgedeckt, wird die Test-Suite um ein Paar bestehend aus ω (d. h. einem abstrakten Testfall) und dem Testziel tga erweitert (Zeile 6). Aus dem abstrakten Testfall kann später ein konkreter Testfall abgeleitet werden, wie es in Beispiel 3.15 bereits demonstriert wurde. Gibt es keinen Pfad im ARG_T, der das Testziel tga abdeckt, ist tga unerfüllbar oder unbekannt und es wird entsprechend kein Testfall zur Test-Suite TS hinzugefügt. Sobald das Testziel tga bearbeitet wurde, wird das Testziel aus der Menge UC entfernt (Zeile 7).

Algorithmus 2 konstruiert für jedes Testziel einen ARG_T und generiert dann für jedes Testziel einen neuen Testfall. Dies kann zum einen dazu führen, dass viele Teile der ARG_T für die verschiedenen Testziele mehrfach konstruiert werden und zum anderen kann es vorkommen, dass für verschiedene Testziele redundante Testfälle werden, wodurch die generierte Test-Suite unnötig groß werden kann. Das Beispiel 3.23 zeigt, dass selbst für kleine Programm viele Teile der ARG_T während der Test-Suite-Generierung mehrfach generiert werden können (vgl. Abbildung 3.10). Weiterhin sind beispielsweise die Testfälle tc_1 und tc_3 in Beispiel 3.23 gleiche Testfälle, die für unterschiedliche Testziele generiert wurden. Für größere IUTs werden die genannten Probleme tendenziell noch verstärkt. Als Gegenmaßnahme kann für Testfälle gezielt analysiert werden, ob die Testfälle zwischen Testzielen wiederverwendet werden können. Dies wird im folgenden Abschnitt beschrieben wird.

3.1.7 *Wiederverwendung von Testfällen zwischen Testzielen*

Testfälle können während der Test-Suite-Generierung zwischen Testzielen wiederverwendet werden, indem für einen generierten Testfall überprüft wird, ob er weitere, noch nicht abgedeckte Testziele zusätzlich abdeckt, wie es beispielsweise von

Holzer in [Hol13] vorgeschlagen wurde. Dies reduziert die Anzahl der ARG_T-Konstruktionen, die zur Test-Suite-Generierung notwendig sind, da für Testziele keine ARG_T-Konstruktion durchgeführt werden muss, die bereits durch einen Testfall abgedeckt werden, der für ein anderes Testziel generiert wurde. Weiterhin reduziert die Wiederverwendung von Testfällen die Anzahl der generierten Testfälle, da Testfälle zwischen Testzielen wiederverwendet werden können und somit ein Testfall mehrere Testziele abdecken kann.

Beispiel 3.24 (Test-Suite-Generierung mit Wiederverwendung von Testfällen)**.** Wie in Beispiel 3.23 soll eine Test-Suite zur Abdeckung der Testzielmenge $\{tga_1, tga_2, tga_3\}$ aus Abbildung 3.5 für die C-Funktion GaussianSum erzeugt werden. In diesem Beispiel soll die Test-Suite jedoch unter Wiederverwendung von Testfällen zwischen Testzielen erzeugt werden.

Begonnen wird beispielsweise mit der Generierung eines Testfalls für das Testziel tga_3. Zuerst wird der ARG_T aus Abbildung 3.10c konstruiert. Im Programmzustand 11 auf dem ARG_T-Pfad $\omega = 2 - 4 - 5 - 10 - 11$ ist der TGA tga_3 in einem Endzustand und das Testziel tga_3 ist somit abgedeckt. Anhand der Pfadbedingung

$$r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0 \wedge i_0 \leq n_0$$

von Programmzustand 11 kann ein konkreter Testfall $tc_1 = ([x = 1, y = 2])$ abgeleitet werden. Für den abstrakten Testfall ω wird nun für alle noch nicht abgedeckten Testziele geprüft, ob die Testziele ebenfalls durch diesen abstrakten Testfall und somit auch durch den konkreten Testfall tc_1 abgedeckt werden. Hierfür wird geprüft, ob der TGA, der ein Testziel spezifiziert, den abstrakten Testfall ω akzeptiert. TGA tga_2 akzeptiert den abstrakten Testfall ω nicht und wird somit nicht durch ω abgedeckt. Im Gegensatz dazu akzeptiert der TGA tga_1 den abstrakten Testfall ω. Somit kann der Testfall tc_1 zur Abdeckung von tga_1 wiederverwendet werden. Da das Testziel tga_1 durch den Testfall tc_1 bereits mit abgedeckt ist, muss für das Testziel tga_1 kein ARG_T zur Testfallgenerierung mehr konstruiert werden. Da das Testziel tga_2 noch nicht abgedeckt ist, wird im nächsten Schritt für das Testziel tga_2 der ARG_T aus Abbildung 3.10b konstruiert und aus der Pfadbedingung von Programmzustand 10′, in dem der TGA tga_2 einen Endzustand erreicht hat, der konkrete Testfall $tc_2 = ([x = 2, y = 1])$ abgeleitet.

Die generierte Test-Suite $TS_2 = \{tc_1, tc_2\}$ deckt die Testzielmenge $\{tga_1, tga_2, tga_3\}$ für die C-Funktion GaussianSum vollständig ab. Dabei konnte durch die Wiederverwendung von Testfällen zwischen Testzielen ein Testfall eingespart werden und eine ARG_T-Konstruktion weniger ausgeführt werden als in Beispiel 3.23.

Der Grad der Wiederverwendung von Testfällen zwischen Testzielen kann von der Reihenfolge abhängen in der die Testziele während der Test-Suite-Generierung betrachtet werden. Würde beispielsweise zuerst für das Testziel tga_1 ein abstrakter Testfall (ARG_T-Pfad $2 - 4 - 5 - 10$) konstruiert werden, könnte dieser nicht zur Abdeckung des Testziels tga_3 wiederverwendet werden, da

Algorithmus 3 Test-Suite-Generierung mit Wiederverwendung von Testfällen zwischen Testzielen

Input: Kontrollflussautomat $CFA = (L, \ell_0, \ell_t, E)$,
Menge von Testzielautomaten $TG = \{tga_1, \ldots, tga_n\}$
Output: Abstrakte Test-Suite TS

```
1:  TS := {}
2:  UC := TG
3:  while UC ≠ ∅ do
4:    tga := (Q, Σ, Δ, q0, F) ∈ UC
5:    if ∃ ω := ψ0 -e0-> ··· -ek-1-> ψk mit ψi = (ℓi, qi, ri) für i ≤ 0 ≤ k und qk ∈ F then
6:      for each tga' ∈ UC do
7:        if tga' akzeptiert ω then
8:          TS := TS ∪ (tga', ω)
9:          UC := UC \ tga'
10:   else
11:     UC := UC \ tga
```

tga_3 durch diesen Testfall nicht abgedeckt wird. Somit müssten sowohl für tga_3 als auch für tga_2, wie in Beispiel 3.23, weitere Testfälle separat generiert werden. Der Einfluss der Reihenfolge, in der Testziele optimalerweise betrachtet werden sollten, ist nicht Gegenstand dieser Arbeit.

Experimentelle Evaluationen zeigen, dass durch die Wiederverwendung von Testfällen zwischen Testzielen weniger ARG_T-Konstruktionen für die Test-Suite-Generierung benötigt und kleinere Test-Suiten generiert werden können [Hol13]. Dies gilt insbesondere für größere IUTs.

Algorithmus 3 beschreibt das Vorgehen zur Test-Suite-Generierung mit Wiederverwendung von Testfällen zwischen Testzielen, wie es in Beispiel 3.24 beschrieben wurde. Der Algorithmus baut auf Algorithmus 2 auf und erweitert Algorithmus 2 um die Zeilen 6, 7 und 9. Sobald ein ARG_T-Pfad ω (d. h. ein abstrakter Testfall) gefunden wurde, der das aktuell betrachtete Testziel tga abdeckt, wird für jedes noch nicht abgearbeitete Testziel $tga' \in UC$ überprüft, ob es ebenfalls durch den ARG_T-Pfad ω abgedeckt wird (Zeile 6-7). Ein ARG_T-Pfad ω deckt tga' ab, wenn ω einer (partiellen) Programmausführung entspricht, die tga' akzeptiert (vgl. Abschnitt 3.1.3). Deckt ω das Testziel tga' ab, wird die Test-Suite um ein Tupel, bestehend aus dem abstrakten Testfall ω und dem Testziel tga' erweitert (Zeile 8). Weiterhin wird das Testziel tga' aus der Menge UC entfernt (Zeile 9).

Die Wiederverwendung von Testfällen zwischen Testzielen kann die Anzahl von ARG_T-Konstruktionen und von generierten Testfällen reduzieren [Hol13]. Um die Test-Suite-Generierung weiter zu verbessern, kann die Anzahl der zu konstruierenden Programmzustände und somit die Größe der konstruierten ARG_T verringert werden, indem häufiger geprüft wird, ob Programmzustände im ARG_T vereinigt werden können. In Beispiel 3.22 wird gezeigt, dass Programmzustände, welche die Programm-Location 10 des CFAs der GaussianSum-Funktion referenzieren, im ARG_T aus Abbildung 3.9 vereinigt werden können, da sich der TGA tga_6 in den vereinigten Programmzuständen im gleichen TGA-Zustand befindet (q_6^1). Für den

ARG_T aus Abbildung 3.6 ist das Vereinigen der Programmzustände 10 und 10' hingegen nicht möglich, da sich der TGA tga_7 in den beiden Programmzuständen in verschiedenen TGA-Zuständen befindet. Somit müssen für die Programm-Location 10 im ARG_T aus Abbildung 3.6 alle nachfolgenden Programmzustände doppelt berechnet werden (einmal als Nachfolger von Programmzustand 5 und einmal für Programmzustand 7), wohingegen die nachfolgenden Programmzustände für den ARG_T aus Abbildung 3.9 durch das Vereinigen der nachfolgenden Programmzustände von Programmzustand 5 und Programmzustand 7 nur einmal berechnet werden müssen. Im nächsten Abschnitt wird beschrieben wie Programmzustände unabhängig davon, ob sich die TGA-Zustände in zu vereinigenden Programmzuständen unterscheiden, vereinigt werden können. Dadurch kann die Anzahl der zu berechnenden Programmzustände eines ARG_T verringert werden.

3.2 ARG_T mit Programmzustandsvereinigungen

Der üblicherweise sehr schnell wachsende (abstrakte) Programmzustandsraum ist eines der inhärenten Probleme beim Model-Checking [BCM+90] und somit auch bei der Testfallgenerierung mit Hilfe von Model-Checking. Zum Beispiel führt jede Bedingung in einem Programm (z. B. der CFA-Knoten 4 des CFAs aus Abbildung 3.2) zur Berechnung von zwei (abstrakten) nachfolgenden Programmzuständen (z. B. die ARG_T-Zustände 5 und 7 des ARG_T aus Abbildung 3.6). Um die Größe des ARG_T dennoch kontrollierbar zu halten, führen Model-Checker üblicherweise Kontrollflussvereinigungen im ARG_T durch (z. B. in Programmzustand 10 aus Abbildung 3.9). Bei Kontrollflussvereinigungen können zwei abstrakte Zustände zusammengefasst werden, welche die gleiche Programm-Location referenzieren, wenn dadurch kein Verlust von Datenfluss- und Kontrollflussinformationen (Programmpfadinformationen) entsteht, der die Validität der Analyse beeinträchtigt. Bei der Testfallgenerierung können allerdings Kontrollflussvereinigungen zum Verlust von Programmpfadinformationen führen, wenn ein Testziel auf einem zu vereinigenden Kontrollfluss ganz oder teilweise abgedeckt ist und auf dem anderen zu vereinigenden Kontrollfluss nicht. In diesem Abschnitt werden wir die Probleme beim Vereinigen von Kontrollflüssen während der Testfallgenerierung an einem Beispiel darstellen. Weiterhin werden wir ein Verfahren einführen, um den Verlust von Programmpfadinformationen zu verhindern. Dadurch können bei der ARG_T-Konstruktion während der Testfallgenerierung Kontrollflüsse präziser vereinigt werden als es beispielsweise der Merge-Operator $merge_{\mathbb{A}}$ der Automaten-CPA $CPA_{\mathbb{A}}$ zulässt.

3.2.1 *Verlust von Programmpfadinformationen in ARG_T*

Während der Testfallgenerierung führen Kontrollflussvereinigungen zu Informationsverlust, wenn sich der TGA, für den ein ARG_T konstruiert wird, in den vereinigten Programmzuständen in verschiedenen TGA-Zuständen befindet. Dies kann zur Ableitung von konkreten Testfällen für ein Testziel führen, die das Testziel gar nicht abdecken, wodurch Algorithmus 2 eine inkorrekte Test-Suite zurückliefern würde.

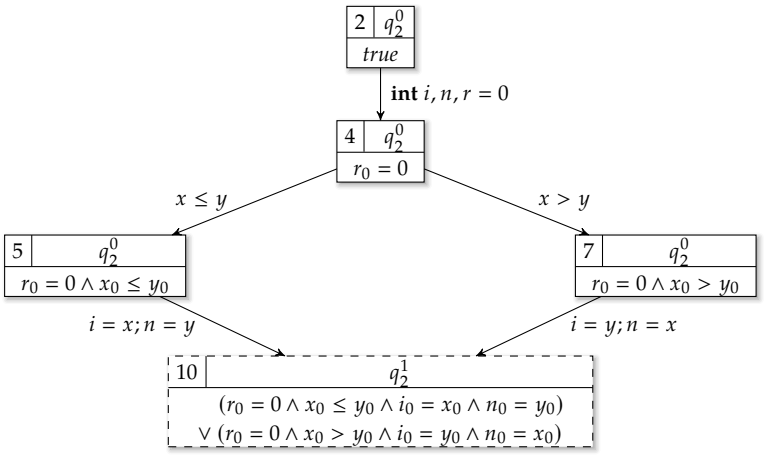


Abbildung 3.11: Ausschnitt des ARG_T der C-Funktion GaussianSum für das Testziel tga_2 mit der Vereinigung von Programmzuständen

Beispiel 3.25 (Informationsverlust durch Vereinigungen von Programmzuständen in ARG_T)**.** Abbildung 3.10b zeigt einen Ausschnitt des konstruierten ARG_T zur Testfallgenerierung für das Testziel tga_2, wobei im ARG_T keine Programmzustände vereinigt werden. Die Vereinigung der Programmzustände 10 und 10′ wird dadurch verhindert, dass sich der TGA tga_2 in den zwei Programmzuständen 10 und 10′ in unterschiedlichen TGA-Zuständen befindet (vgl. $merge_A$-Operator in Definition 3.21). Durch die Verhinderung der Vereinigung der Programmzustände 10 und 10′, müssen alle weiteren nachfolgenden Programmzustände doppelt berechnet werden (einmal als Nachfolger von Programmzustand 10 und einmal als Nachfolger von Programmzustand 10′).

Die Mehrfachberechnung von Programmzuständen kann durch das Vereinigen von Programmzuständen vermieden werden. Abbildung 3.11 zeigt den ARG_T, der für das Testziel tga_2 konstruiert wurde, wobei die Programmzustände 10 und 10′ des ARG_T aus Abbildung 3.10b hier zu einem Programmzustand 10 vereinigt wurden. Durch das Vereinigen müssen alle Folgezustände von Programmzustand 10 nur einmal berechnet werden. Allerdings führt die Vereinigungen zu einem Verlust von Programmpfadinformationen, der eine korrekte Testfallgenerierung für das Testziel tga_2 verhindern kann. Das Testziel tga_2 fordert, dass ein ARG_T-Pfad die Anweisungen $i = y; n = x$ traversiert. Dies ist in Programmzustand 10 durch den ARG_T-Pfad $2 - 4 - 7 - 10$ erfüllt und der TGA tga_2 erreicht in Programmzustand 10 einen Endzustand. Wenn ein Testeingabevektor aus der Pfadbedingung von Programmzustand 10

$$(r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0)$$
$$\vee (r_0 = 0 \wedge x_0 > y_0 \wedge i_0 = y_0 \wedge n_0 = x_0)$$

abgeleitet wird, könnte jedoch ein Eingabevektor abgeleitet werden, welcher zwar der Pfadbedingung von Programmzustand 10 entspricht, aber bei der Ausführung den Programmpfad $2 - 4 - 5 - 10$ traversiert und somit das Testziel tga_2 nicht abdeckt. Ein Beispiel für einen solchen Testfall ist $([x = 3, y = 1])$.

Der mögliche Verlust an Datenfluss- und Kontrollflussinformationen nach der Vereinigung von Programmzuständen wird im Folgenden als Verlust an *Pfadsensitivität* bezeichnet. Um dem Verlust an Pfadsensitivität entgegenzuwirken hat Holzer in [Hol13] den Merge-Operator $merge_{\mathbb{A}}$ der Automaten-CPA $\text{CPA}_{\mathbb{A}}$ so definiert, dass er keine Vereinigungen von Programmzuständen zulässt, in denen sich ein TGA in verschiedenen TGA-Zuständen befindet. Dies kann schon sehr früh während der ARG_T-Konstruktion dazu führen, dass keine Programmzustände vereinigt werden und somit viele Programmzustände unter Umständen mehrfach auf verschiedenen Programmpfaden berechnet werden müssen, wie in Beispiel 3.22 zu sehen ist. Damit auch Programmzustände vereinigt werden können, in denen sich ein TGA in unterschiedlichen TGA-Zuständen befindet, werden wir im Folgenden die Definition des Merge-Operators $merge_{\mathbb{A}}$ anpassen und gleichzeitig eine Technik zur Sicherstellung einer hinreichenden Pfadsensitivität bei solchen Vereinigungen von Programmzuständen einführen.

3.2.2 *Konstruktion eines ARG_T unter Erhaltung der Pfadsensitivität*

Zur Erhaltung der Pfadsensitivität trotz der Vereinigung von Programmzuständen während der ARG_T-Konstruktion zur Testfallgenerierung führen wir das Konzept der *symbolischen Pfadsensitivität (SPS)* ein, welches in einer gemeinsamen Forschungsarbeit zwischen der Technischen Universität Darmstadt, der Universität Passau und der Ludwig-Maximilians-Universität München entwickelt wurde [ABB+ar]. SPS erlaubt es zu ermitteln, welche Programmpfade ein Testziel erfüllen, auch wenn im ARG_T Programmzustände vereinigt werden, die auf verschiedene TGA-Zustände verweisen. Somit können mit Hilfe von SPS trotz der Vereinigung von Programmzuständen korrekte Testfälle aus einer Pfadbedingung eines Programmzustands abgeleitet werden, selbst wenn die Pfadbedingung unter anderem ARG_T-Pfade beschreibt, die das Testziel nicht abdecken. Hierzu sind zwei Schritte notwendig:

1. In einem Programmzustand wird nicht mehr nur ein nicht-deterministisch berechneter TGA-Zustand gespeichert, sondern eine Menge von allen TGA-Zuständen, die in diesem Programmzustand erreicht werden können. Dies ist die Grundlage für die Veränderung des Merge-Operators der Automaten-CPA dahingehend, dass die Automaten-CPA die Vereinigung von Programmzuständen immer erlaubt.

2. Zur Erhaltung der Pfadsensitivität nach der Vereinigung von Programmzuständen werden dynamisch Informationen zu Ausführungspfaden von Programmen hinzugefügt, sodass entschieden werden kann, ob die Ausführungspfade relevant zur Abdeckung eines Testziels sind.

Zur Änderung der Definition des Merge-Operators $merge_{\mathbb{A}}$ dahingehend, dass das Vereinigen von Programmzuständen immer erlaubt ist, müssen alle TGA-Zustände in den Programmzuständen gespeichert werden, die in den jeweiligen Programmzuständen erreichbar sind. Im Gegensatz dazu wurde bisher nur genau ein nicht-deterministischer TGA-Zustand gespeichert, der in einem Programmzustand erreichbar ist.

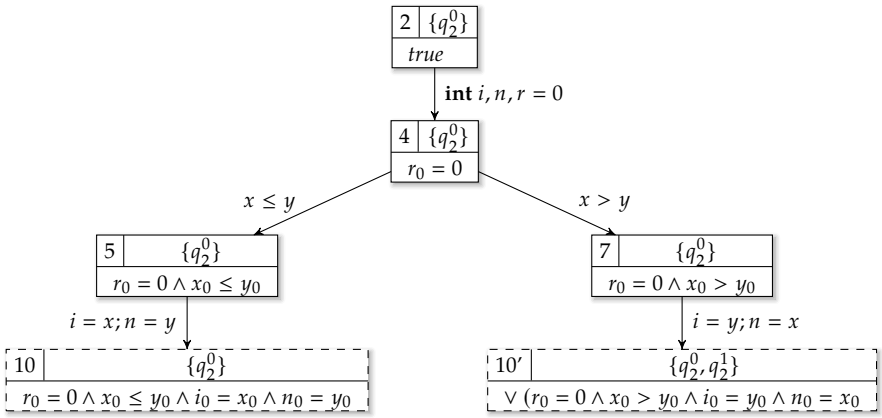


Abbildung 3.12: Ausschnitt des ARG_T der Funktion GaussianSum für das Testziel tga_2 mit Mengen von TGA-Zuständen in Programmzuständen

Beispiel 3.26 (ARG_T mit TGA-Zustandsmengen)**.** Bisher wurde für jeden Programmzustand nicht-deterministisch genau ein TGA-Zustand berechnet, in dem sich der TGA in dem Programmzustand befinden kann. Im Gegensatz dazu zeigt Abbildung 3.12 einen ARG_T, der für das Testziel tga_2 konstruiert wurde, bei dem für jeden Programmzustand konkret in einer Menge aufgelistet wurde, in welchem TGA-Zustand sich tga_2 in diesem Programmzustand befinden kann. Zum Beispiel ist der TGA tga_2 in Programmzustand 2 in seinem initialen Zustand q_2^0 und kann auch keinen anderen Zustand erreichen. In Programmzustand 10′ kann sich der TGA jedoch in mehreren Zuständen befinden. Zum einen weiterhin in TGA-Zustand q_2^0, da beim Traversieren der CFA-Kante $e = (5, i = y; n = x, 10)$ die Selbsttransition $q_2^0 \xrightarrow{*} q_2^0$ des TGA-Zustand q_2^0 ausgeführt wurde. Zum anderen in TGA-Zustand q_2^1, da beim Traversieren der CFA-Kante e die TGA-Transition $q_2^0 \xrightarrow{i = y; n = x} q_2^1$ ausgeführt wurde.

Durch das Speichern von TGA-Zustandsmengen in Programmzuständen, kann der Merge-Operator $merge_{\mathbb{A}}$ so definiert werden, dass der Merge-Operator Programmzustände nicht mehr nur dann vereinigt, wenn der aktuelle TGA-Zustand der Programmzustände gleich ist, sondern dass der Merge-Operator Programmzustände immer vereinigt. Aus diesen Überlegungen ergibt sich die neue Automaten-CPA $CPA_{\mathbb{A}'}$. Die CPA $CPA_{\mathbb{A}'}$ arbeitet auf der Domäne $D_{\mathbb{A}'}$ bestehend aus Mengen von TGA-Zuständen. Bei einem Programmzustandsübergang $\Psi \rightsquigarrow_{\mathbb{A}'} \Psi'$ für eine CFA-Kante e gemäß der Transferrelation $\rightsquigarrow_{\mathbb{A}'}$ werden als TGA-Zustände in Ψ' alle TGA-Zustände verwendet, die als Nachfolger der erreichten TGA-Zustände in Ψ gemäß der CFA-Kante möglich sind. Beim Vereinigen von Programmzuständen durch den Merge-Operator $merge_{\mathbb{A}'}$ werden die TGA-Zustandsmengen der zu vereinigenden Programmzustände vereinigt. Dadurch wird in einem ARG_T immer dann vereinigt, wenn der Kontrollfluss eines Programms zusammenfließt.

Definition 3.27 (Automaten-CPA $CPA_{\mathbb{A}'}$)**.** Seien (L, ℓ_0, ℓ_t, E) ein CFA und $A = (Q, \Sigma, \Delta, q_0, F)$ ein TGA mit dem Alphabet $\Sigma \subseteq E \times R$. Die zugehörige Automa-

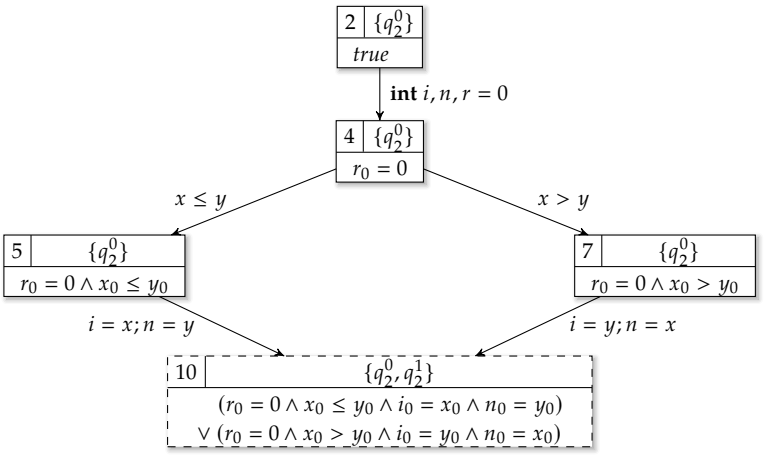


Abbildung 3.13: Ausschnitt des ARG_T der C-Funktion GaussianSum für den TGA tga_2 mit der Vereinigung von Programmzuständen und TGA-Zustandsmengen

ten-CPA $CPA_{\mathbb{A}'} = (D_{\mathbb{A}'}, \rightsquigarrow_{\mathbb{A}'}, merge_{\mathbb{A}'}, stop_{\mathbb{A}'})$ besteht aus den folgenden vier Komponenten:

- $D_{\mathbb{A}'} = (C_{\mathbb{A}'}, \Psi_{\mathbb{A}'}, [\![\cdot]\!]_{\mathbb{A}'})$ besteht aus einer Menge von konkreten Zuständen $C_{\mathbb{A}'} = 2^Q$, einer Menge von abstrakten Zuständen $\Psi_{\mathbb{A}'} = 2^Q$ und einer Konkretisierungsfunktion $[\![\cdot]\!]_{\mathbb{A}'} : C_{\mathbb{Q}} \rightarrow \Psi_{\mathbb{Q}}$, die der Identitätsfunktion entspricht.
- Sei $e \in E$ eine CFA-Kante, dann enthält die Transferrelation $\rightsquigarrow_{\mathbb{A}'}$ den Programmzustandsübergang

$$Q' \rightsquigarrow_{\mathbb{A}'} Q'' \text{ gdw. } \forall (q', e, q'') \in \Delta : q' \in Q' \Leftrightarrow q'' \in Q''$$

- $merge_{\mathbb{A}'}(Q', Q'') = Q' \cup Q''$
- $stop_{\mathbb{A}'}(Q', reached) = \exists Q'' \in reached : Q' = Q''$

Die komponierte CPA $CPA_{\mathbb{R}'}$, die sich aus der Komposition der Location-CPA $CPA_{\mathbb{L}}$, der Prädikaten-CPA $CPA_{\mathbb{P}}$ und der Automaten-CPA $CPA_{\mathbb{A}'}$ ergibt, unterscheidet sich zur CPA $CPA_{\mathbb{R}}$ darin, dass immer dann vereinigt wird, wenn Kontrollflüsse im CFA zusammenfließen, unabhängig davon in welchen TGA-Zuständen sich der TGA in den zu vereinigenden Programmzuständen befindet.

Beispiel 3.28 ($CPA_{\mathbb{R}'}$)**.** Abbildung 3.13 zeigt einen ARG_T, der durch die CPA $CPA_{\mathbb{R}'}$ für das Testziel tga_2 konstruiert wurde. Jeder Programmzustand besteht aus einer Programm-Location, einer Menge von TGA-Zuständen und einer Pfadbedingung. Bei dem Programmzustandsübergang von Programmzustand 7 zu Programmzustand 10 werden alle nachfolgeden TGA-Zustände von TGA-Zustand q_2^0 berechnet. Dies ist zum einen der TGA-Zustand q_2^0 selbst, aufgrund der Selbsttransition $q_2^0 \xrightarrow{*} q_2^0$ in TGA tga_2 und zum anderen der TGA-Zustand q_2^1, da der Zustandsübergang $q_2^0 \xrightarrow{i = y; n = x} q_2^1$ gemäß der traversierten

CFA-Kante $(7, i = y; n = x; 10)$ im TGA tga_2 erlaubt ist. Bei dem Programmzustandsübergang von Programmzustand 5 zu Programmzustand 10 ist im TGA tga_2 lediglich die Ausführung der Selbsttransition $q_2^0 \xrightarrow{*} q_2^0$ erlaubt. Beim Vereinigen der jeweiligen Nachfolgerprogrammzustände der Programmzustände 5 und 7 ergibt sich somit die TGA-Zustandsmenge $\{q_2^0, q_2^1\} = \{q_2^0\} \cup \{q_2^0, q_2^1\}$ für Programmzustand 10 gemäß des Merge-Operators $merge_{\mathbb{A}'}$.

Wie in Beispiel 3.25 beschrieben wurde, geht durch die Vereinigung von Programmzuständen, die verschiedene Mengen von TGA-Zuständen enthalten, die Pfadsensitivität verloren. Somit gehen Informationen über den Daten- und den Kontrollfluss verloren, die es erlauben zu ermitteln, welcher Programmpfad, der zu einem Programmzustand führt, ein Testziel tatsächlich abdeckt. Zur Sicherstellung einer hinreichenden Pfadsensitivität fügen wir deshalb dynamisch Informationen zu Ausführungspfaden von Programmen hinzu, um zu entscheiden, ob die Ausführungspfade relevant zur Abdeckung eines Testziels sind [ABB+ar]. Hierzu annotieren wir TGA-Transitionen mit Testpfadmarkierungen, die einer TGA-Kante entsprechen. Eine *Testpfadmarkierung* ist dabei eine Boole'sche Variable, welche die Transitionen in TGAs eindeutig identifizieren. Transitionen, die mit $*$ beschriftet sind, werden nicht mit Testpfadmarkierungen annotiert. Für jede andere Transition eines TGA wird eine Testpfadmarkierungsvariable eingeführt, deren Name dem Muster pmX,Y entspricht, wobei X den TGA identifiziert und Y ein eindeutiger Identifizierer für die Transition des TGA ist.

Beispiel 3.29 (Testpfadmarkierungen)**.** Abbildung 3.14 zeigt die TGAs aus Abbildung 3.5 angereichert um Testpfadmarkierungen. Dabei wird jede Transition eines TGA um eine Programmoperation erweitert, die der zugehörigen Testpfadmarkierungsvariable den Wert 1 (wahr) zuweist. Ausgenommen davon sind Transitionen, die mit * beschriftet sind. Beispielsweise wird die Beschriftung der Transition $q_1^0 \xrightarrow{i = x; n = y} q_1^1$ von tga_1 um die Anweisung ***pm1,0 = 1*** erweitert. Da tga_5 zwei Transitionen enthält, die keine $*$-Transitionen sind, werden diesem TGA zwei Testpfadmarkierungsvariablen ***pm5,0 = 1*** und ***pm5,1 = 1*** zugeordnet, mit denen die Transitionsbeschriftungen der zwei Transitionen $q_5^0 \xrightarrow{i \leq n} q_5^1$ und $q_5^1 \xrightarrow{i \leq n} q_5^2$ angereichert werden.

Die Testpfadmarkierungsvariablen werden im Folgenden verwendet, um Pfadinformationen während der TGA-getriebenen Testgenerierung zu speichern. Dafür werden die Testpfadmarkierungsvariablen am Anfang der ARG_T-Konstruktion mit 0 (d. h. *false*) initialisiert. Jedes Mal, wenn eine TGA-Transition einer ARG_T-Transition während der Konstruktion des Programmzustandsraums entspricht, wird die zur TGA-Transition gehörige Testpfadmarkierungsvariablen auf 1 (d. h. *true*) gesetzt, indem die entsprechende Programmoperation ***pmX,Y = 1*** zur Pfadbedingung des nachfolgenden Programmzustands hinzugefügt wird.

Beispiel 3.30 (ARG_T mit Testpfadmarkierungen)**.** Abbildung 3.15 zeigt einen Ausschnitt aus dem ARG_T für das Testziel tga_2 mit einer Testpfadmarkierungsvariablen, wobei die Testpfadmarkierungsvariable *pm2,0* fett hervorgehoben

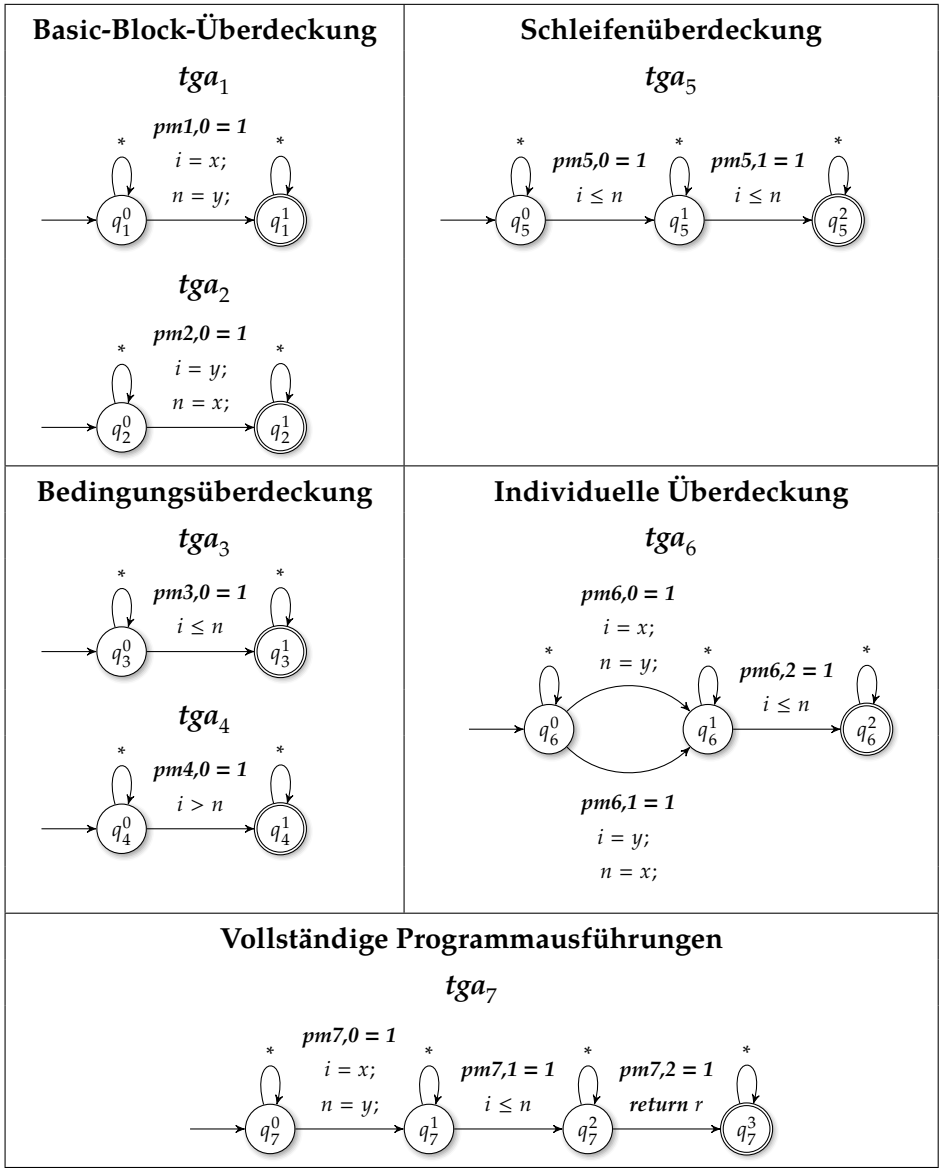


Abbildung 3.14: Testzielautomaten für die C-Funktion GaussianSum mit Testpfadmarkern

ist. Die Testpfadmarkierungsvariable *pm2,0* wird in der Pfadbedingung des initialen Programmzustands 2 mit dem Wert 0 initialisiert. Sobald eine zu einer Testpfadmarkierungsvariable gehörige CFA-Transition einer Transition im ARG_T entspricht, wird der Testpfadmarkierungsvariablen in der Pfadbedingung des nachfolgenden Programmzustands der Wert 1 zugewiesen. Dies ist beispielsweise in Programmzustand 10 für die Testpfadmarkierungsvariable *pm2,0* der Fall. Dadurch wird die Analyse pfadsensitiv und es kann nachvollzogen werden, dass beispielsweise ein Testfall, der das Testziel tga_2 erfüllt, den Else-Zweig der If-Anweisung in Zeile 4 der GaussianSum-Funktion ausführen muss (d. h. Programmzustand 7), da nur bei der Traversierung des Else-Zweiges die Testpfadmarkierungsvariable *pm2,0* auf 1 gesetzt wird.

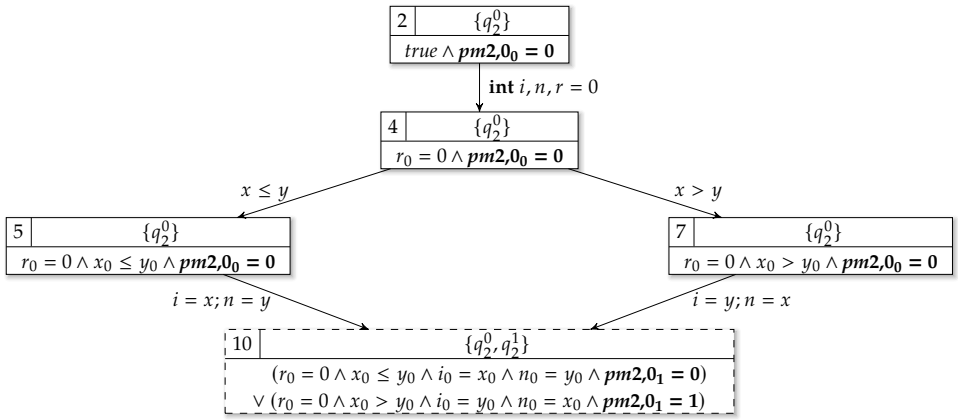


Abbildung 3.15: Ausschnitt des ARG_T der C-Funktion GaussianSum für das Testziel tga_2 mit Testpfadmarkern

Der TGA tga_6 zeigt, dass mehrere (sich gegenseitig ausschließende) Pfade durch einen TGA ein Testziel erfüllen können. Deswegen muss zur Ableitung eines konkreten Testfalls von der Pfadbedingung eines ARG_T-Zustands für einen TGA tga_X eine Teilmenge von Testpfadmarkierungsvariablen *pmX,Y*, die einem akzeptierenden Pfad im TGA entsprechen, auf den Wert 1 (*true*) gesetzt werden. Betrachtet wird dabei immer die letzte Zuweisung zu einer Testpfadmarkierungsvariable, d. h. die Testpfadmarkierungsvariable mit dem höchsten SSA-Index. Wie bei Programmvariablen, werden die SSA-Indizes von Testpfadmarkierungsvariable beim Vereinigen von Programmzuständen synchronisiert (vgl. Beispiel 3.8).

Beispiel 3.31 (Ableiten von konkreten Testfällen mit Testpfadmarkierungen)**.** Es soll ein konkreter Testfall abgeleitet werden, der das Testziel tga_2 abdeckt. Das Testziel tga_2 ist in Programmzustand 10 des ARG_T aus Abbildung 3.15 abgedeckt. Um einen konkreten Testfall abzuleiten, konjugieren wir die Pfadbedingung von Programmzustand 10 mit den auf den Wert 1 gesetzten Testpfadmarkierungsvariablen von tga_2, d. h. mit $\boldsymbol{pm2{,}0_1 = 1}$. Die Pfadbedingung muss dabei mit der *letzten* Zuweisung von *pm2,0* konjugiert werden, d. h. in diesem Fall mit $\boldsymbol{pm2{,}0_1 = 1}$. Jede Lösung der daraus resultierenden Formel

$$\begin{pmatrix} (r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0 \wedge \boldsymbol{pm2{,}0_1 = 0}) \\ \vee\, (r_0 = 0 \wedge x_0 > y_0 \wedge i_0 = y_0 \wedge n_0 = x_0 \wedge \boldsymbol{pm2{,}0_1 = 1}) \end{pmatrix} \wedge \boldsymbol{pm2{,}0_1 = 1}$$

entspricht einem validen, konkreten Testfall zur Abdeckung von tga_2, da Aufgrund der Bedingung $\boldsymbol{pm2{,}0_1 = 1}$ nur der Teil der Formel erfüllt sein kann, der dem Else-Zweig in Zustand 7 entspricht. Dies funktioniert, wenn die SSA-Indizes der Testpfadmarkierungsvariablen *pm2,0* nach der Vereinigung in Programmzustand 10 für die eingehenden ARG_T-Pfade synchronisiert werden. Andernfalls wäre die Pfadbedingung in Programmzustand 10

$$\begin{pmatrix} (r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0 \wedge \boldsymbol{pm2{,}0_0 = 0}) \\ \vee\, (r_0 = 0 \wedge x_0 > y_0 \wedge i_0 = y_0 \wedge n_0 = x_0 \wedge \boldsymbol{pm2{,}0_1 = 1}) \end{pmatrix}.$$

> In der Pfadbedingung für den If-Zweig hätte die Variable *pm2,0* den Index 0, da auf diesem Pfad keine Zuweisung zur Variablen *pm2,0* ausgeführt wurde. Konjugiert mit $\boldsymbol{pm2,0_1 = 1}$ zur Ableitung einer konkreten Variablenbelegung könnte dann auch der Teil der Pfadbedingung für den If-Zweig eine valide Lösung haben, da *pm2,0* mit dem Index 1 hier keine Rolle spielt. Durch die Synchronisation des Index beim Vereinigen von Programmzuständen wird jedoch $\boldsymbol{pm2,0_0 = 0}$ durch $\boldsymbol{pm2,0_1 = 0}$ ersetzt und der Pfadbedingungsteil für den If-Zweig konjugiert mit $\boldsymbol{pm1,0_1 = 1}$ ist unerfüllbar.

Wie in Beispiel 3.30 illustriert wurde, verwenden wir Testpfadmarkierungen, um TGAs in der IUT zu kodieren. Anstatt direkt den Programm-Code der IUT vor der Testgenerierung mit Testpfadmarkierungsinformationen zu instrumentieren, verwenden wir eine Instrumentierung zur Laufzeit mit Hilfe der *LOOM*-Analyse, einer Technik aus dem Bereich des Multi-Property-Checkings [ABM+16]. Die LOOM-Analyse erlaubt es, zusätzliche Kontrollflusskanten in den ARG_T während der ARG_T-Konstruktion hinzuzufügen, d. h. in unserem Fall die CFA-Kanten mit den Testpfadmarkierungen, mit denen die TGA-Transitionen beschriftet sind. Somit wird immer dann, wenn ein TGA eine Transition ausführt, die mit einer Zuweisung zu einer Testpfadmarkierungsvariable beschriftet ist, automatisch der nachfolgende ARG_T-Zustand um diese Zuweisung ergänzt. Die LOOM-Analyse unterstützt dabei auch SSA-Indizes, sodass beispielsweise beim Vereinigen von Programmzuständen, die SSA-Indizes von Testpfadmarkierungen korrekt synchronisiert werden (vgl. Beispiel 3.31). Die LOOM-Analyse ist in einer CPA implementiert, die parallel zu den bereits beschriebenen CPAs geschaltet werden kann, sodass die LOOM-Analyse automatisch den ARG_T während der ARG_T-Konstruktion um Testpfadmarkierungsinformationen anreichert. Eine Definition der LOOM-CPA kann in [ABM+16] gefunden werden.

Trotz Techniken wie der Wiederverwendung von Testfällen zwischen Testzielen (vgl. Abschnitt 3.1.7) und dem Vereinigen von Programmzuständen während der ARG_T-Konstruktion kann die Generierung einer Test-Suite aufgrund der immer noch zahlreichen ARG_T-Konstruktionen sehr aufwändig sein. Eine Neuberechnung des ARG_T (im schlimmsten Fall) für jedes Testziel kann sehr ineffizient sein und sollte vermieden werden. Zur Vermeidung der Neuberechnung von ARG_T für jedes Testziel ist es möglich, mehrere Testziele gleichzeitig bei einer einzelnen ARG_T-Konstruktion zu berücksichtigen, sodass die direkte Wiederverwendung von Erreichbarkeitsinformationen zwischen Testzielen weiter verbessert werden kann. Dies gilt gerade für Programme mit tausenden Testzielen, die dicht beieinander die gesamte Programmstruktur abdecken sollen. Deswegen erweitern wir den Testgenerierungsansatz im nächsten Abschnitt um die Möglichkeit, mehrere Testziele während einer einzelnen ARG_T-Konstruktion zu berücksichtigen.

3.3 Testgenerierung mit Multi-Property-Checking

Für jedes Testziel einzeln einen ARG_T zur Ableitung einer Test-Suite zu konstruieren kann sehr ineffizient sein (vgl. Beispiel 3.23). Deswegen kann es sinnvoll sein, aus einem konstruierten ARG_T mehrere Testfälle abzuleiten, die mehrere Testziele überdecken. Um dies zu erreichen, darf zum einen die ARG_T-Konstruktion nicht

abgebrochen werden, sobald ein Testziel abgedeckt wurde, damit nach weiteren Testfällen für andere Testziele im ARG_T gesucht werden kann und die ARG_T-Konstruktion muss durch mehrere TGAs gleichzeitig getrieben werden, welche die unterschiedlichen Testziele spezifizieren.

Multi-Property-Checking erlaubt es, während einer ARG-Konstruktion mehrere Programmeigenschaften gleichzeitig zu verifizieren [ABM+16]. In diesem Abschnitt werden wir diese Idee aus dem Bereich Multi-Property-Checking auf die automatisierte Generierung von Testfällen für mehrere Testziele zugleich mit Hilfe von Model-Checking übertragen. Dadurch können mehrere Testziele während einer einzigen ARG_T-Konstruktion analysiert werden und somit Erreichbarkeitsinformationen zwischen den Testzielen wiederverwendet werden. Die in diesem Abschnitt präsentierten Ideen basieren auf einer gemeinsamen Forschungsarbeit zwischen der Technischen Universität Darmstadt, der Universität Passau und der Ludwig-Maximilians-Universität München [ABB+ar].

3.3.1 *Testfallgenerierung mit Multi-Property-Checking*

Zur Testfallgenerierung mit Model-Checking wurde in Abschnitt 3.1.4 ein ARG_T verwendet, der für einen TGA anhand des CFAs einer IUT konstruiert wurde. Aus diesem ARG_T kann ein abstrakter Testfall abgeleitet werden, der den TGA abdeckt, d. h. ein ARG_T-Pfad, der einer Programmausführung entspricht, die vom TGA akzeptiert wird. Durch Lösen der Pfadbedingung des ARG_T-Pfades kann ein konkreter Testfall aus dem abstrakten Testfall abgeleitet werden.

Wie in Abschnitt 3.1.5 gezeigt wurde, schränkt die Automaten-CPA die Konstruktion eines ARG_T für einen TGA möglichst auf die Teile des ARG_T ein, die für den TGA relevant sind. Bei der Erweiterung der Testfallgenerierung um Techniken aus dem Multi-Property-Checking, um mehrere TGAs gleichzeitig zu betrachten, muss der ARG_T einer IUT somit nicht nur für einen TGA, sondern für eine Menge von TGAs konstruiert werden. Durch die Betrachtung mehrerer TGAs bei einer ARG_T-Konstruktion wird sichergestellt, dass alle Programmzustände berücksichtigt werden, die für mindestens einen TGA relevant sind. Wird ein ARG_T beispielsweise nur für einen TGA tga_1 konstruiert, kann es vorkommen, dass Teile des ARG_T die für tga_1 nicht relevant sind, aber für einen anderen TGA tga_2 schon, stärker abstrahiert werden und nicht ausreichend Informationen zur Ableitung eines Testfalls zur Abdeckung von tga_2 zur Verfügung stehen. In Abbildung 3.10a wurde ein ARG_T für tga_1 aus Abbildung 3.5 konstruiert. Dieser enthält relevante Programmzustände zur Abdeckung von tga_2 nicht, da diese Programmzustände für die Abdeckung des Testziels tga_1 nicht relevant sind, d. h. insbesondere die Programmzustände 7 und 10' der ARG_T aus Abbildung 3.10b. In diesem und den folgenden Abschnitten werden wir die Testgenerierung aus Abschnitt 3.1 dahingehend erweitern, dass bei der Konstruktion eines ARG_T mehrere TGAs berücksichtigt werden. Dadurch kann die Anzahl an ARG_T-Konstruktionen für die Generierung von Test-Suiten für mehrere Testziele reduziert werden.

Zur Berücksichtigung mehrerer TGAs erweitern wir die Definition von ARG_T dahingehend, dass in einem Programmzustand nicht mehr nur der aktuelle TGA-

Zustand von einem einzelnen TGA gespeichert wird, sondern die aktuellen Zustände einer Menge von TGAs.

Definition 3.32 (ARG_M)**.** Ein ARG_M zu einem CFA (L, ℓ_0, ℓ_t, E), der für eine Menge von TGAs $\{tga_1, \ldots, tga_n\}$ mit $tga_i = (Q_i, \Sigma_i, \Delta_i, q_i^0, F_i)$, $i = 1, \ldots, n$, konstruiert wird, ist ein Tripel (S_M, s_{0_M}, T_M), wobei

- $S_M \subseteq L \times Q_1 \times \ldots \times Q_n \times R(V)$ eine Menge von abstrakten Programmzuständen,
- $s_{0_M} = (\ell_0, q_{0_1}, \ldots, q_{0_n}, true) \in S_M$ der initiale abstrakte Programmzustand und
- $T_M \subseteq S_M \times E \times S_M$ eine beschriftete Transitionsrelation mit

  $$(\ell, q_0, \ldots, q_n, r) \xrightarrow{e} (\ell', q_0', \ldots, q_n', r') \in T_M,$$

  wenn $e = (\ell, a, \ell') \in E$, $(q_i, e, q_i') \in \Delta_i$ für $i = 1, \ldots, n$ und $r' \models sp_V(r, a)$ gelten,

ist.

Ein Testziel $tga_i = (Q_i, \Sigma_i, \Delta_i, q_i^0, F_i)$, $i = 1, \ldots, n$, wird von einem ARG_M-Pfad

$$\omega = (l_0, q_1^0, \ldots, q_n^0, r_0) \xrightarrow{e_0} \cdots \xrightarrow{e_{k-1}} (l_k, q_1^k, \ldots, q_n^k, r_k)$$

abgedeckt, wenn $q_i^k \in F_i$ gilt, d. h. der entsprechende TGA im letzten Zustand von ω einen Zielzustand erreicht hat. Gibt es im vollständigen ARG_M keinen Pfad, der ein Testziel abdeckt, nennen wir das Testziel *unerfüllbar* oder *unbekannt*.

Beispiel 3.33 (ARG_M)**.** Abbildung 3.10 zeigt die verschiedenen ARG_T, die zur Test-Suite-Generierung für die Testziele tga_1, tga_2 und tga_3 aus Abbildung 3.5 konstruiert werden. Aus jedem ARG_T wurde in Beispiel 3.23 ein Testfall abgeleitet, der eines der Testziele abdeckt. Dabei wurden viele Programmzustände mehrfach berechnet, obwohl die Programmzustände sich nur im TGA-Zustand des aktuell betrachteten TGAs unterscheiden, z. B. die Programmzustände 2, 4, 5 und 10 in den verschiedenen ARG_T.

Abbildung 3.16 zeigt den entsprechenden ARG_M, der für die Testziele tga_1, tga_2 und tga_3 konstruiert wurde. Im Gegensatz zu den ARG_T aus Abbildung 3.16 wurden in dem ARG_M jedoch alle Testziele parallel betrachtet. Jeder Programmzustand im ARG_M besteht aus einer Programm-Location, den aktuellen Zuständen der TGAs tga_1, tga_2 und tga_3 sowie einer Pfadbedingung. Der ARG_M enthält sowohl die Teile des Programmzustandsraumes, die für tga_1 relevant sind, als auch die Programmzustände, die für tga_2 und tga_3 relevant sind. Beispielsweise ist der Programmzustand 11 weder zur Abdeckung des Testziels tga_1 relevant, noch zur Abdeckung des Testziels tga_2. Der Programmzustand 11 wurde dennoch berechnet um das Testziel tga_3 abzudecken.

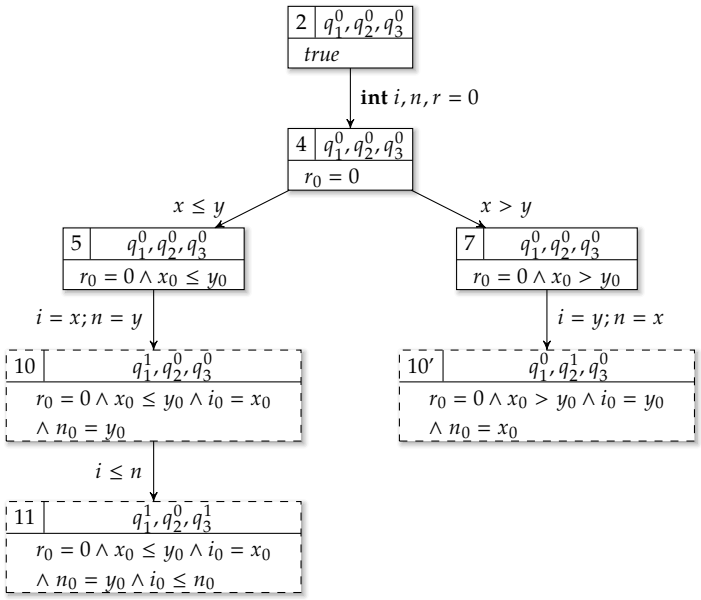


Abbildung 3.16: Ausschnitt des ARG_M der C-Funktion GaussianSum für die Testzielmenge $\{tga_1, tga_2, tga_3\}$

Die Ableitung konkreter Testfälle erfolgt wie in Beispiel 3.15. Der TGA tga_1 hat im Programmzustand 10 einen Endzustand erreicht und das Testziel tga_1 ist somit abgedeckt. Durch das Lösen der Pfadbedingung

$$r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0$$

des Programmzustands 10 kann ein konkreter Testfall zur Abdeckung des Testziels tga_1, bestehend aus einem Eingabevektor, abgeleitet werden (z. B. $tc_1 = [x = 1, y = 2]$). Der TGA tga_2 erreicht in Programmzustand 10′ einen Endzustand. Durch das Lösen der Pfadbedingung

$$r_0 = 0 \wedge x_0 > y_0 \wedge i_0 = y_0 \wedge n_0 = x_0$$

des Programmzustands 10′ kann ein konkreter Testfall zur Abdeckung des Testziels tga_2 abgeleitet werden (z. B. $tc_2 = [x = 2, y = 1]$). Weiterhin erreicht TGA tga_3 in Programmzustand 11 einen Endzustand. Durch das Lösen der Pfadbedingung

$$r_0 = 0 \wedge x_0 \leq y_0 \wedge i_0 = x_0 \wedge n_0 = y_0 \wedge i_0 \leq n_0$$

des Programmzustands 11 kann ein konkreter Testfall zur Abdeckung des Testziels tga_3 abgeleitet werden, z. B. $tc_3 = [x = 1, y = 2]$. Die Test-Suite $\{tc_1, tc_2, tc_3\}$ deckt die Testzielmenge $\{tga_1, tga_2, tga_3\}$ vollständig ab.

Das in Abschnitt 3.1.5 beschriebene Vorgehen zur ARG_T-Konstruktion berücksichtigt nur einen einzigen TGA. Um einen ARG_M für mehrere TGAs zu konstruieren, muss das Vorgehen zur ARG_T-Konstruktion entsprechend angepasst werden.

3.3.2 *ARG_M-Konstruktion für mehrere TGAs*

Die ARG_T-Konstruktion aus Abschnitt 3.1.5 besteht aus zwei Teilen:

1. Einer komponierten CPA $CPA_{\mathbb{T}}$, welche die Programmzustände eines ARG_T beschreibt sowie Operationen, die auf den Programmzuständen spezifiziert sind.

2. Einem generischen Algorithmus (Algorithmus 1), der für einen CFA mit Hilfe der Operationen der CPA $CPA_{\mathbb{T}}$ einen ARG_T konstruiert.

Zum Erweitern der ARG_T-Konstruktion für einen TGA hin zu einer ARG_M-Konstruktion für mehrere TGAs ist es ausreichend, die CPA $CPA_{\mathbb{T}}$ anzupassen. Algorithmus 1 muss nicht verändert werden. Die Anpassung der komponierten CPA $CPA_{\mathbb{T}}$ beschränkt sich auf die Erweiterung der Automaten-CPA $CPA_{\mathbb{A}}$, die einen TGA während der ARG_T-Konstruktion behandelt. Die Location-CPA $CPA_{\mathbb{L}}$ und die Prädikaten-CPA $CPA_{\mathbb{P}}$ können unverändert weiterverwendet werden.

Die Erweiterung der Automaten-CPA $CPA_{\mathbb{A}}$ umfassen zum einen das Anpassen der Domäne der Automaten-CPA, die nun die aktuellen TGA-Zustände mehrerer TGAs speichert, und zum anderen die Anpassung der Operationen der Automaten-CPA, sodass beim Berechnen eines Nachfolgeprogrammzustands, beim Vereinigen und beim Terminierungs-Check alle von der Automaten-CPA betrachteten TGAs berücksichtigt werden. Somit erweitern wir die Definition 3.21 der Automaten-CPA $CPA_{\mathbb{A}}$ wie folgt:

- Die Domäne der Automaten-CPA speichert ein Tupel von Automatenzuständen mit je einem Element pro TGA.

- Die Transferrelation sowie die Operatoren *merge* und *stop* der Automaten-CPA berücksichtigen immer alle TGAs komponentenweise.

Das Ergebnis dieser Anpassungen ist die Automaten-CPA $CPA_{\mathbb{A}_M} = (D_{\mathbb{A}_M}, \rightsquigarrow_{\mathbb{A}_M}, merge_{\mathbb{A}_M}, stop_{\mathbb{A}_M})$, welche die ARG_M-Konstruktion für mehrere TGAs durchführt. Die Automaten-CPA $CPA_{\mathbb{A}_M}$ berechnet nicht-deterministisch für jeden Programmzustand, in welchen TGA-Zuständen sich die TGAs $A_i = (Q_i, \Sigma_i, \Delta_i, q_{i,0}, F_i)$ mit $i = 1, \ldots, n$ beim Erreichen des Programmzustands befinden. Dabei arbeit die CPA $CPA_{\mathbb{A}_M}$ auf der Domäne $D_{\mathbb{A}_M} = Q_1 \times \ldots \times Q_n$ des Kreuzproduktes der TGA-Zustände der TGAs und schaltet immer dann, wenn eine CFA-Kante traversiert wird, die Zustandsübergänge in allen TGAs erlaubt. Durch die *-Transitionen an jedem TGA-Zustand von jedem TGA gibt es für jede CFA-Kante immer mindestens eine Möglichkeit für jeden TGA zu schalten. Dies verhindert das Steckenbleiben einer ARG_M-Konstruktion, da alle TGAs immer in jedem Schritt simultan schalten können. Eine Vereinigung von Programmzuständen ist gemäß des Merge-Operators $merge_{\mathbb{A}_M}$ immer dann erlaubt, wenn die zu vereinigenden Programmzustände für alle TGAs jeweils die gleichen TGA-Zustände referenzieren. Der Stopp-Operator $stop_{\mathbb{A}_M}$ überprüft für jeden Programmzustand, ob es einen anderen Programmzustand in der Menge der bereits erreichten Programmzustände *reached* gibt, der die gleichen TGA-Zustände referenziert. Ist dies der Fall, wird *true* zurückgegeben und die Analyse für den überprüften Programmzustand gestoppt.

Definition 3.34 (Automaten-CPA für mehrere Automaten $\text{CPA}_{\mathbb{A}_M}$)**.** Seien (L, ℓ_0, ℓ_t, E) ein CFA und $TGA = \{tga_1, \ldots, tga_n\}$ mit $tga_i = (Q_i, \Sigma_i, \Delta_i, q_{i_0}, F_i)$, $i = 1, \ldots, n$, eine Menge von TGAs mit dem Alphabet $\Sigma \subseteq E \times R$. Die Automaten-CPA für mehrere Automaten $\text{CPA}_{\mathbb{A}_M} = (D_{\mathbb{A}_M}, \rightsquigarrow_{\mathbb{A}_M}, merge_{\mathbb{A}_M}, stop_{\mathbb{A}_M})$ besteht aus den folgenden vier Komponenten:

- $D_{\mathbb{A}_M} = (C_{\mathbb{A}_M}, \Psi_{\mathbb{A}_M}, [\![\cdot]\!]_{\mathbb{A}_M})$ besteht aus einer Menge von konkreten Zustände $C_{\mathbb{A}_M} = Q_1 \times \ldots \times Q_n$, einer Menge von abstrakten Zuständen $\Psi_{\mathbb{A}_M} = Q_1 \times \ldots \times Q_n$ und einer Konkretisierungsfunktion $[\![\cdot]\!]_{\mathbb{A}_M} : C_{\mathbb{Q}} \to \Psi_{\mathbb{Q}}$, die der Identitätsfunktion entspricht.
- Sei $e \in E$ eine CFA-Kante, dann enthält die Transferrelation $\rightsquigarrow_{\mathbb{A}_M}$ den Programmzustandsübergang $(q_1, \ldots, q_n) \rightsquigarrow^e_{\mathbb{A}_M} (q'_1, \ldots, q'_n)$, wenn jeder TGA A_i eine Transition $(q_i, e, q'_i) \in \Delta_i$, $i = 1, \ldots, n$, enthält.
- $merge_{\mathbb{A}_M}((q_1, \ldots, q_n), (q'_1, \ldots, q'_n)) = \begin{cases} (q_1, \ldots, q_n) & \text{, mit } \forall q_i = q'_i \\ & \text{für } i = 1, \ldots, n \\ (q'_1, \ldots, q'_n) & \text{sonst} \end{cases}$
- $stop_{\mathbb{A}_M}((q_1, \ldots, q_n), reached) = (\exists (q'_1, \ldots, q'_n) \in reached : (q_1, \ldots, q_n) = (q'_1, \ldots, q'_n))$

Die aus der Location-CPA $\text{CPA}_{\mathbb{L}}$, der Prädikaten-CPA $\text{CPA}_{\mathbb{P}}$ und der Automaten-CPA $\text{CPA}_{\mathbb{A}_M}$ komponierte CPA $\text{CPA}_{\mathbb{R}_M}$ führt eine automatengetriebene ARG_M-Konstruktion für einen CFA durch, die

- alle Pfade traversiert, die für mindestens einen TGA relevant sein können und
- terminiert, sobald die Analyse einen ARG_M-Pfad findet, der durch mindestens einen TGA akzeptiert wird.

Beispiel 3.35 ($\text{CPA}_{\mathbb{R}_M}$)**.** Abbildung 3.16 zeigt einen ARG_M, der mit der CPA $CPA_{\mathbb{R}_M}$ für die Testzielmenge $\{tga_1, tga_2, tga_3\}$ konstruiert wurde. Die Location-CPA $\text{CPA}_{\mathbb{L}}$ und die Prädikaten-CPA $\text{CPA}_{\mathbb{P}}$ arbeiten dabei wie zuvor. Die Automaten-CPA $\text{CPA}_{\mathbb{A}_M}$ verfolgt die TGA-Zustände der TGAs tga_1, tga_2 und tga_3. Die Transferrelation $\rightsquigarrow_{\mathbb{A}_M}$ berechnet für jeden TGA einen neuen TGA-Zustand auf die gleiche Art und Weise wie die Transferrelation $\rightsquigarrow_{\mathbb{A}}$ (vgl. Definition 3.21). Zwei Programmzustände werden vereinigt, wenn die Programm-Locations der Programmzustände gleich sind und alle TGAs in den beiden Programmzuständen in den gleichen TGA-Zuständen sind. Zum Beispiel enthalten Programmzustand 10 und 10′ die gleiche Programm-Location (10) und der TGA tga_3 befindet sich in beiden Programmzuständen im TGA-Zustand q^0_3. Die Programmzustände 10 und 10′ dürfen dennoch nicht vereinigt werden, da sich die TGAs tga_1 und tga_2 in den Programmzuständen 10 und 10′ jeweils in unterschiedlichen TGA-Zuständen befinden.

Die CPA $\text{CPA}_{\mathbb{A}}$ ist ein Spezialfall der CPA $\text{CPA}_{\mathbb{A}_M}$, da sich die CPA $\text{CPA}_{\mathbb{A}}$ genauso wie die CPA $\text{CPA}_{\mathbb{A}_M}$ verhält, wenn die CPA $\text{CPA}_{\mathbb{A}_M}$ eine Menge von TGAs analysiert,

die genau einen TGA enthält. Das gilt auch für die CPAs $CPA_{\mathbb{R}}$ und $CPA_{\mathbb{R}_M}$. Die CPA $CPA_{\mathbb{R}}$ verhält sich genauso wie die CPA $CPA_{\mathbb{R}_M}$, wenn die CPA $CPA_{\mathbb{R}_M}$ genau einen TGA betrachtet.

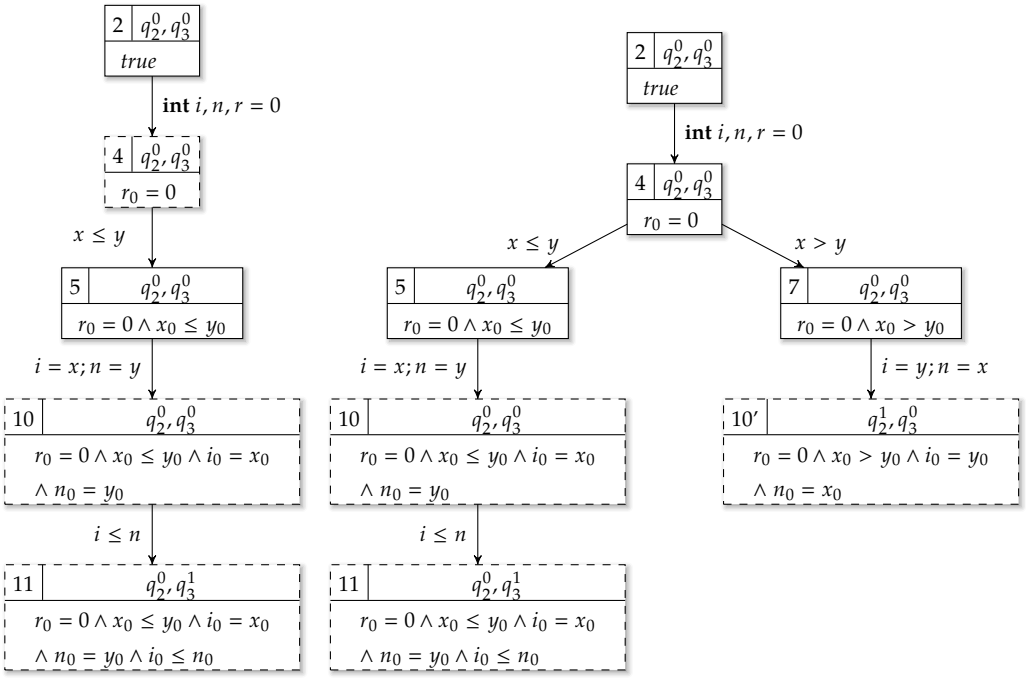
Die ARG_M-Konstruktion für mehrere TGAs wird unterbrochen, sobald ein TGA einen ARG_M-Pfad akzeptiert, um dann einen Testfall aus dem ARG_M-Pfad ableiten zu können. Damit die ARG_M-Konstruktion danach wieder fortgesetzt werden kann, um weitere Testfälle zur Abdeckung weiterer Testziele abzuleiten, wird ein Algorithmus ähnlich Algorithmus 2 benötigt. Im folgenden Abschnitt werden wir einen solchen Algorithmus beschreiben, der die ARG-Konstruktion solange fortsetzt, bis alle betrachteten TGAs abgedeckt wurden oder alle noch nicht abgedeckten TGAs als unerfüllbar oder unbekannt identifiziert wurden.

3.3.3 *Test-Suite-Generierung mit Multi-Property-Checking*

In diesem Abschnitt werden wir einen Algorithmus zur Generierung von Test-Suiten basierend auf der ARG_M-Konstruktion aus Abschnitt 3.3.2 beschreiben. Dabei können alle abzudeckenden Testziele während einer ARG_M-Konstruktion parallel betrachtet werden und somit können durch eine einzige ARG_M-Konstruktion alle erfüllbaren Testziele durch eine generierte Test-Suite abgedeckt werden. Die Betrachtung aller Testziele parallel ist jedoch aus zwei Gründen nicht immer sinnvoll. Zum einen werden die Abstraktionsmöglichkeiten bei der ARG_M-Konstruktion verringert, da tendenziell sämtliche Teile des ARG_M für mindestens eines der über das Programm verteilten Testziele relevant sind. Dies ist beispielsweise bei Anweisungsüberdeckung der Fall und führt zu einem hohen Berechnungsaufwand, da der ARG_M sehr detailliert berechnet werden muss. Zum anderen müssen bei der Testfallgenerierung Erreichbarkeitsinformationen für eine sehr große Anzahl von Testzielen verfolgt werden. Dies führt ebenfalls zu einem hohen Berechnungsaufwand. Beide Gründe führen somit zu einem Berechnungsaufwand, der mit zunehmender Testzielanzahl steigt und ab einem gewissen Grad die Vorteile von Multi-Property-Checking für Testfallgenerierung überwiegen kann. Deswegen sollte der Algorithmus für die Test-Suite-Generierung mit Multi-Property-Checking eine Möglichkeit enthalten, Testziele auf verschiedene Partitionen zu verteilen, um so den Trade-off zwischen Berechnungsaufwand und der Wiederverwendung von Erreichbarkeitsinformationen möglichst zu verbessern.

Beispiel 3.36 (Test-Suite-Generierung mit Multi-Property-Checking)**.** Wie in Beispiel 3.23 soll eine Test-Suite zur Abdeckung der Testziele tga_1, tga_2 und tga_3 aus Abbildung 3.5 für die C-Funktion GaussianSum generiert werden. Im Gegensatz zur Test-Suite-Generierung in Beispiel 3.23 soll in diesem Beispiel Multi-Property-Checking zur Test-Suite-Generierung eingesetzt werden. Hierfür wird die Testzielmenge in die zwei Partitionen $TG_1 = \{tga_2, tga_3\}$ und $TG_2 = \{tga_1\}$ aufgeteilt, die nacheinander abgearbeitet werden.

Begonnen wird mit Testfallgenerierung für die Testziele der Partition TG_1. Es wird der in Abbildung 3.17a dargestellte ARG_M konstruiert. In Programmzustand 11 ist der TGA tga_3 in einem Endzustand, das Testziel tga_3 ist somit



(a) Konstruktionsschritt 1 **(b)** Konstruktionsschritt 2

Abbildung 3.17: Schrittweise Konstruktion des ARG_M der C-Funktion GaussianSum für die Testzielmenge $\{tga_2, tga_3\}$

in diesem Programmzustand erfüllt und es kann aus der Pfadbedingung von Programmzustand 11 ein Testfall abgeleitet werden, der das Testziel tga_3 abdeckt. Zum Beispiel der Testfall $tc_1 = ([x = 1, y = 2])$. Da noch nicht alle Testziele aus der Partition TG_1 abgedeckt sind, wird der ARG_M aus Abbildung 3.17a weiter konstruiert, um das noch nicht abgedeckte Testziel tga_2 ebenfalls abzudecken. Abbildung 3.17b zeigt den weiter konstruierten ARG_M. In Programmzustand 10′ ist das Testziel tga_2 abgedeckt und es kann ein Testfall $tc_2 = ([x = 2, y = 1])$ aus der Pfadbedingung von Programmzustand 10′ abgeleitet werden, der das Testziel tga_2 abdeckt. Die Partition TG_1 wird durch die Testfälle tc_1 und tc_2 vollständig abgedeckt. Somit kann die Konstruktion des ARG_M beendet werden und es können Testfälle für die nächste Partition TG_2 generiert werden.

Zur Generierung eines Testfalls zur Abdeckung des Testziels tga_1 aus der Partition TG_2 wird der ARG_M aus Abbildung 3.10a konstruiert. Da die Partition TG_2 nur das eine Testziel tga_1 enthält, entspricht der ARG_M dem ARG_T für das Testziel tga_1. Das Testziel tga_1 ist in Programmzustand 10 erfüllt und es kann der Testfall $tc_3 = ([x = 1, y = 2])$ wie bisher abgeleitet werden.

Die generierte Test-Suite $TS_1 = \{tc_1, tc_2, tc_3\}$ deckt die Testzielmenge $\{tga_1, tga_2, tga_3\}$ vollständig ab. Im Gegensatz zur Test-Suite-Generierung ohne Multi-Property-Checking aus Beispiel 3.23 wurden nur zwei statt drei ARGs konstruiert, da durch die gleichzeitige Betrachtung der Testziele tga_2 und tga_3 Erreichbarkeitsinformationen während der ARG_M-Konstruktion zwischen den Testzielen wiederverwendet werden konnten.

Algorithmus 4 Test-Suite-Generierung mit Multi-Property-Checking

Input: Kontrollflussautomat $CFA = (L, \ell_0, \ell_t, E)$,
Menge von Testzielautomaten $TG = \{tga_1, \ldots, tga_n\}$
Output: Abstrakte Test-Suite TS

1: $TS := \{\}$
2: $UC := TG$
3: **while** $UC \neq \emptyset$ **do**
4: $TG' := \{tga'_1, \ldots, tga'_l\} \subseteq UC$ mit $tga'_i = (Q'_i, \Sigma'_i, \Delta'_i, q'_{0_i}, F'_i), 1 \leq i \leq l$
5: **while** $\exists\, \omega := \psi_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{k-1}} \psi_k$ mit $\psi_i = (\ell_i, q'_{i_1}, \ldots, q'_{i_l}, r_i), 0 \leq i \leq k$, und $\exists q'_{k_j} \in F'_j, 1 \leq j \leq l$ **do**
6: **for each** $tga'_m \in TG'$ mit $q'_{k,m} \in F'_m, 1 \leq m \leq l$ **do**
7: $TS := TS \cup (tga'_m, \omega)$
8: $TG' := TG' \setminus tga'_m$
9: $UC := UC \setminus tga'_m$
10: $UC := UC \setminus TG'$

Algorithmus 4 beschreibt das Vorgehen zur Test-Suite-Generierung mit Multi-Property-Checking. Der Algorithmus baut auf Algorithmus 2 auf. Allerdings unterscheidet er sich zu Algorithmus 2 darin, dass nicht nur ein abstrakter Testfall für ein Testziel pro Durchlauf der While-Schleife in Zeile 5 generiert wird, sondern eine Menge von abstrakten Testfällen für eine Menge von Testzielen. Die Ein- und Ausgaben sowie die Initialisierungsphasen der beiden Algorithmen unterscheiden sich nicht.

Zur Generierung von Test-Suiten für eine Menge von TGAs mit Hilfe von Multi-Property-Checking geht Algorithmus 4 wie folgt vor. Als Eingabe erhält Algorithmus 4 einen CFA und eine Menge von Testzielen TG, d. h. eine Menge von TGAs, und als Ausgabe liefert der Algorithmus eine abstrakte Test-Suite TS. Die Test-Suite TS wird mit der leeren Menge initialisiert (Zeile 1) und die Menge der nicht-behandelten Testziele UC mit der Menge aller Testziele TG (Zeile 2). Solange bis alle Testziele verarbeitet wurden (Zeile 3) wird eine Teilmenge von nicht-bearbeiteten Testzielen TG' aus UC ausgewählt (Zeile 4) für die eine Menge von Testfällen generiert werden soll. Für die Menge TG' werden solange abstrakte Testfälle generiert, d. h. ARG_M-Pfade aus dem ARG_M abgeleitet, bis jedes Testziel aus TG' durch mindestens einen Testfall abgedeckt wurde oder es keinen neuen ARG_M-Pfad mehr gibt, der ein Testziel aus TG' abdeckt (Zeile 5). Die dafür notwendige ARG_M-Konstruktion wird durch die komponierte CPA $CPA_{\mathbb{R}_M}$ aus Abschnitt 3.3.2 geleitet. Wurde ein ARG_M-Pfad ω gefunden, der ein Testziel aus TG' abdeckt, d. h. in dessen letzten Programmzustand mindestens ein TGA aus der Menge TG' im Endzustand ist (Zeile 5), wird ω direkt als abstrakter Testfall verarbeitet. Hierfür wird für jeden TGA $tga'_m \in TC'$ überprüft, ob der TGA tga'_m im letzten Zustand des ARG_M-Pfades ω in einem Endzustand ist (Zeile 6). Ist dies der Fall, wird ein Tupel bestehend aus dem TGA tga'_m und dem abstrakten Testfall ω zur Test-Suite hinzugefügt (Zeile 7). Weiterhin wird der TGA tga'_m sowohl aus der Menge TG' als auch aus der Menge der noch nicht verarbeiteten Testziele UC entfernt (Zeile 8–9). Danach wird die Konstruktion des ARG_M für die verbliebenen Testziele in

TG' fortgesetzt. Gibt es keinen neuen ARG_M-Pfad mehr, der ein Testziel aus TG' abdeckt, gelten alle verbliebenen TGA $tga \in TG'$ als unerfüllbar und werden aus der Menge UC entfernt, d. h. für diese TGAs kann kein Testfall abgeleitet werden (Zeile 10). Aus den abstrakten Testfällen in TS können konkrete Testfälle abgeleitet werden wie es in Beispiel 3.33 beschrieben wurde.

Zur Zerlegung der Testzielmenge in verschiedene Partitionen in Zeile 4 von Algorithmus 4 können verschiedene Strategien implementiert werden. Die einfachste Strategie zerlegt die Testzielmenge zufällig in Partitionen der Größe l. Dadurch kann die Größe der Testzielpartitionen auf ein handhabbares Maß reduziert werden. Da die Auswahl der Testziele, die gemeinsam in einer Partition der Größe l landen zufällig ist, muss die Zerlegung nicht optimal für die Test-Suite-Generierung sein. Beispielsweise können zufällig Testziele ausgewählt werden, die im CFA und somit auch im ARG_M weit auseinander liegen, wodurch wieder viele Teile des ARG_M detailliert konstruiert werden müssen und dadurch weniger abstrahiert werden kann. Eine andere Strategie zur Zerlegung der Testzielmenge könnte deshalb Informationen über den CFA berücksichtigen, d. h. beispielsweise Testziele in einer Partition zusammenzufassen, die im CFA nahe beieinanderliegen. Die Entwicklung von Strategien zur Zerlegung von Testzielmengen ist jedoch nicht Gegenstand dieser Arbeit.

Zur Verbesserung der Effizienz der Test-Suite-Generierung mit Multi-Property-Checking werden wir die Test-Suite-Generierung im folgenden Abschnitt um Wiederverwendung von Testfällen zwischen Testzielen anreichern, wie dies bereits in Abschnitt 3.1.7 für Test-Suite-Generierung ohne Multi-Property-Checking beschrieben wurde.

3.3.4 *Test-Suite-Generierung mit Multi-Property-Checking und Wiederverwendung von Testfällen zwischen Testzielen*

Nachdem in Algorithmus 4 ein abstrakter Testfall für ein Testziel generiert wurde, wird für alle Testziele der aktuellen Partition geprüft, ob die Testziele durch den Testfall ebenfalls abgedeckt werden. Diese Prüfung kann dahingehend erweitert werden, dass für alle noch nicht abgedeckten Testziele überprüft wird, ob die noch nicht abgedeckten Testziele durch einen generierten Testfall abgedeckt werden (vgl. Algorithmus 3). Dies hat den Vorteil, dass die zusätzlich abgedeckten Testziele aus anderen Partitionen bei späteren Erreichbarkeitsanalysen nicht mehr berücksichtigt werden müssen. Dadurch können später betrachtete Testziele besser auf weitere Partitionen verteilt werden oder die später betrachteten Partitionen werden kleiner, wodurch tendenziell eine höherer Abstraktion bei der ARG_M-Konstruktion möglich wird.

Beispiel 3.37 (Test-Suite-Generierung mit Multi-Property-Checking und Wiederverwendung von Testfällen zwischen Testzielen)**.** Wie in Beispiel 3.36 soll eine Test-Suite zur Abdeckung der Testzielmenge $\{tga_1, tga_2, tga_3\}$ aus Abbildung 3.5 für die Funktion GaussianSum generiert werden. Die Testzielmenge wird wieder in die zwei Partitionen $TG_1 = \{tga_2, tga_3\}$ und $TG_2 = \{tga_1\}$ eingeteilt, die nacheinander abgearbeitet werden.

Algorithmus 5 Test-Suite-Generierung mit Multi-Property-Checking und Wiederverwendung von Testfällen zwischen Testzielen

Input: Kontrollflussautomat $CFA = (L, \ell_0, \ell_t, E)$,
Menge von Testzielautomaten $TG = \{tga_1, \ldots, tga_n\}$
Output: Abstrakte Test-Suite TS

1: $TS := \{\}$
2: $UC := TG$
3: **while** $UC \neq \emptyset$ **do**
4: $TG' := \{tga'_1, \ldots, tga'_l\} \subseteq UC$ mit $tga'_i = (Q'_i, \Sigma'_i, \Delta'_i, q'_{0_i}, F'_i), 1 \leq i \leq l$
5: **while** $\exists\, \omega := \psi_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{k-1}} \psi_k$ mit $\psi_i = (\ell_i, q'_{i_1}, \ldots, q'_{i_l}, r_i), 0 \leq i \leq k$, und $\exists q'_{k_j} \in F'_j, 1 \leq j \leq l$ **do**
6: **for each** $tga' \in UC$ **do**
7: **if** tga' akzeptiert ω **then**
8: $TS := TS \cup (tga', \omega)$
9: $TG' := TG' \setminus tga'$
10: $UC := UC \setminus tga'$
11: $UC := UC \setminus TG'$

Begonnen wird mit Testfallgenerierung für die Testziele der Partition TG_1. Es wird der in Abbildung 3.17a dargestellte ARG_M konstruiert. In Programmzustand 11 ist der TGA tga_3 in einem Endzustand, das Testziel tga_3 ist somit in diesem Programmzustand erfüllt und es kann aus der Pfadbedingung von Programmzustand 11 ein Testfall abgeleitet werden, der das Testziel tga_3 abdeckt. Zum Beispiel der Testfall $tc_1 = ([x = 1, y = 2])$. Für den generierten Testfall tc_1 wird im nächsten Schritt geprüft, ob er noch weitere, nicht abgedeckte Testziele abdeckt. Der Testfall tc_1 deckt ebenfalls das Testziel tga_1 ab. Somit kann der Testfall tc_1 zur Abdeckung des Testziels tga_1 wiederverwendet werden.

Für das Testziel tga_2 wird wie in Beispiel 3.36 der Testfall $tc_2 = ([x = 2, y = 1])$ generiert. Die Partition TG_1 wird durch die Testfälle tc_1 und tc_2 vollständig abgedeckt. Da alle Testziele der Partition TG_2 durch die Wiederverwendung von Testfällen zwischen Testzielen bereits abgedeckt sind (d. h. tga_1), muss kein weiterer ARG_M für die Partition TG_2 mehr konstruiert werden.

Die Test-Suite $TS_2 = \{tc_1, tc_2\}$ deckt die Testzielmenge $\{tga_1, tga_2, tga_3\}$ vollständig ab. Im Gegensatz zur Test-Suite-Generierung in Beispiel 3.36 ohne Wiederverwendung von Testfällen zwischen Testzielen musste nur ein ARG_M anstatt zwei ARG_M konstruiert werden und die generierte Test-Suite TS_2 enthält nur zwei anstatt drei Testfälle wie die Test-Suite TS_1 aus Beispiel 3.36.

Algorithmus 5 zeigt einen Algorithmus basierend auf Algorithmus 4, der für jeden generierten Testfall prüft, ob dieser noch weitere, nicht abgedeckte Testziele abdeckt. Hierfür wurden die Zeilen 6 und 7 angepasst, in denen im neuen Algorithmus für jedes Testziel tga' aus der Menge UC überprüft wird, ob es durch den generierten Testfall ω abgedeckt wird. Dies umfasst insbesondere auch Testziele aus anderen Partitionen. Wird ein Testziel tga' durch den Testfall ω abgedeckt, wird das Paar (tga', ω) zur Test-Suite TS hinzugefügt und tga' aus den Mengen TG'

und *UC* entfernt, und somit nicht weiter bei der Test-Suite-Generierung betrachtet (Zeile 8–10).

Die Verwendung von Multi-Property-Checking für die Test-Suite-Generierung und die Wiederverwendung von Testfällen zwischen Testzielen kann zu einer Effizienzsteigerung bei der Test-Suite-Generierung führen. Eine weitere Verbesserung der Test-Suite-Generierung kann durch die Kombination von Multi-Property-Checking mit der verbesserten ARG_T-Vereinigung aus Abschnitt 3.2 erzielt werden. Dies wird im folgenden Abschnitt beschrieben.

3.3.5 *ARG_M mit Programmzustandsvereinigungen ohne Verlust von Pfadinformationen*

Während der durch einen einzelnen TGA getriebenen ARG_T-Konstruktion aus Abschnitt 3.1.5 werden im ARG_T nur Programmzustände vereinigt, wenn die TGA-Zustände von den zwei zu vereinigenden Programmzuständen gleich sind. In Abschnitt 3.2 wurde beschrieben, dass dadurch während der ARG_T-Konstruktion unter Umständen viele Programmzustände mehrfach für unterschiedliche TGA-Zustände berechnet werden müssen. Zur Vermeidung der Mehrfachberechnung von Programmzuständen wurde ebenfalls in Abschnitt 3.2 eine Technik vorgestellt, um Programmzustände häufig zu vereinigen und dadurch weniger Mehrfachberechnungen von Programmzuständen während einer ARG_T-Konstruktion durchführen zu müssen.

Das Problem des schnell wachsenden Programmzustandsraums durch wenige Programmzustandsvereinigungen ist bei der ARG_M-Konstruktion noch dramatischer. Im ARG_M werden Programmzustände nur vereinigt, wenn sich alle TGAs in den zu vereinenden Programmzuständen in den gleichen TGA-Zuständen befinden.

> **Beispiel 3.38** (Verhinderung der Vereinigung von Programmzuständen in ARG_M)**.** Abbildung 3.17b zeigt einen Ausschnitt des ARG_M für die TGAs tga_2 und tga_3 aus Abbildung 3.5. Die Programmzustände 10 und 10′ können nicht vereinigt werden, da sich zwar der TGA tga_3 in beiden Programmzuständen im TGA-Zustand q_3^0 befindet, aber der TGA tga_2 sich in Programmzustand 10 in TGA-Zustand q_2^0 und in Programmzustand 10′ in TGA-Zustand q_2^1 befindet.

Beispiel 3.38 zeigt, dass eine Vereinigung von Programmzuständen während einer ARG_M-Konstruktion durch einen einzelnen TGA aus der Menge der TGAs verhindert werden kann, für die ein ARG_M konstruiert wird. Tendenziell gilt, je mehr TGAs während einer ARG_M-Konstruktion berücksichtigt werden, desto höher ist die Wahrscheinlichkeit, dass sich mindestens ein TGA in zwei Programmzuständen in verschiedenen TGA-Zuständen befindet. Dadurch würden viele Vereinigungen von Programmzuständen während der ARG_M-Konstruktion verhindert und somit viele Programmzustände für mehrere TGA-Zustandskombinationen mehrfach berechnet.

Um auch während der Konstruktion von ARG_M häufiger Vereinigungen von Programmzuständen zu erlauben, werden wir in diesem Abschnitt Multi-Property-Checking mit der Vereinigung von Programmzuständen aus Abschnitt 3.2 kombinieren. Hierzu sind drei Schritte notwendig:

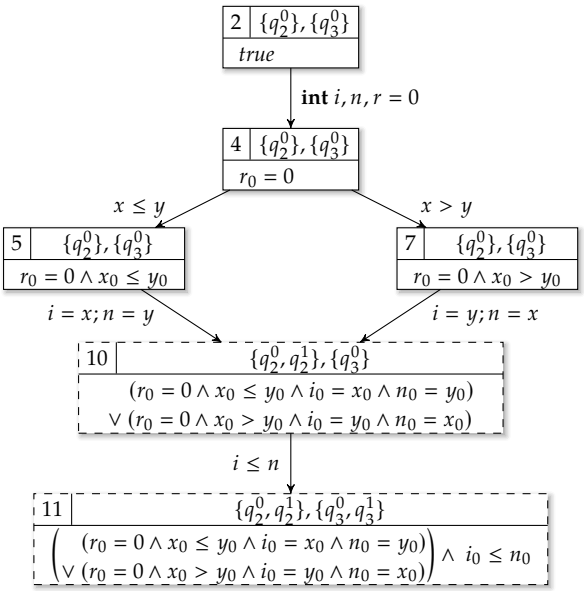


Abbildung 3.18: Ausschnitt des ARG_M der GaussianSum-Funktion für die Testziele tga_2 und tga_3 mit der Vereinigung von Programmzuständen

1. Die Automaten-CPA $\text{CPA}_{\mathbb{A}_M}$ muss für jeden TGA eine Menge von TGA-Zuständen speichern, in denen sich der TGA in einem Programmzustand befinden kann.

2. Anpassung des Merge-Operators $merge_{\mathbb{A}_M}$, sodass beim Vereinigen von zwei Programmzuständen, die TGA-Zustandsmengen der einzelnen TGAs vereinigt werden.

3. Die ARG_M-Konstruktion muss durch Testpfadmarkierungen pfadsensitiv gemacht werden, um dem Verlust von Pfadinformationen (Daten- und Kontrollflussinformationen) nach dem Vereinigen von Programmzuständen entgegenzuwirken.

Beispiel 3.39 (ARG_M mit Vereinigung von Programmzuständen)**.** Abbildung 3.18 zeigt den ARG_M aus Abbildung 3.17b mit Vereinigung von Programmzuständen unabhängig von TGA-Zuständen. Jeder Programmzustand enthält für jeden TGA eine Menge von TGA-Zuständen, die in diesem Programmzustand erreichbar sind (vgl. Beispiel 3.30 für einen TGA). Beispielsweise sind in Programmzustand 10 die TGA-Zustände q_2^0 und q_2^1 in TGA tga_2 erreichbar und der TGA-Zustand q_3^0 in TGA tga_3. Beim Vereinigen von Programmzuständen werden die einzelnen TGA-Zustandsmenge für die verschiedenen TGAs der Programmzustände vereinigt. Beispielsweise werden die TGA-Zustandsmengen $\{q_2^0\}$ und $\{q_3^0\}$ des Nachfolgers von Programmzustand 5 mit den TGA-Zustandsmengen $\{q_2^0, q_2^1\}$, $\{q_3^0\}$ des Nachfolgers von Programmzustand 7 in Programmzustand 10 zu den TGA-Zustandsmengen $\{q_2^0, q_2^1\}$, $\{q_3^0\}$ vereinigt.

Um wie in Beispiel 3.39 einen ARG_M zu konstruieren, der für jeden TGA TGA-Zustandsmengen speichert und diese beim Vereinigen von Programmzuständen

zusammenführt, erweitern wir im Folgenden die Definition der Automaten-CPA $\text{CPA}_{\mathbb{A}_M}$ für mehrere TGAs entsprechend. Die daraus resultierende Automaten-CPA $\text{CPA}_{\mathbb{A}'_M} = (D_{\mathbb{A}'_M}, \rightsquigarrow_{\mathbb{A}'_M}, merge_{\mathbb{A}'_M}, stop_{\mathbb{A}'_M})$ arbeitet auf der Domäne $D_{\mathbb{A}'_M}$ bestehend aus Mengen von TGA-Zustandsmengen. Die Transferrelation $\rightsquigarrow_{\mathbb{A}'_M}$ berechnet die nachfolgende TGA-Zustandsmenge für jeden TGA gemäß der Transferrelation $\rightsquigarrow_{\mathbb{A}'}$ (vgl. Definition 3.27). Beim Vereinigen von Programmzuständen durch den Merge-Operator $merge_{\mathbb{A}'_M}$ werden jeweils die TGA-Zustandsmengen aller TGAs der zu vereinigenden Programmzustände vereinigt. Dadurch wird in einem ARG_M immer dann vereinigt, wenn der Kontrollfluss eines Programms zusammenfließt.

Definition 3.40 (Automaten-CPA für mehrere Automaten $\text{CPA}_{\mathbb{A}'_M}$)**.** Seien (L, ℓ_0, ℓ_t, E) ein CFA und $TGA = \{tga_1, \ldots, tga_n\}$ mit $tga_i = (Q_i, \Sigma_i, \Delta_i, q_{i_0}, F_i)$, $i = 1, \ldots, n$, eine Menge von TGAs mit dem Alphabet $\Sigma \subseteq E \times R$. Die Automaten-CPA für mehrere Automaten $\text{CPA}_{\mathbb{A}'_M} = (D_{\mathbb{A}'_M}, \rightsquigarrow_{\mathbb{A}'_M}, merge_{\mathbb{A}'_M}, stop_{\mathbb{A}'_M})$ besteht aus den folgenden vier Komponenten:

- $D_{\mathbb{A}'_M} = (C_{\mathbb{A}'_M}, \Psi_{\mathbb{A}'_M}, [\![\cdot]\!]_{\mathbb{A}'_M})$ besteht aus einer Menge von konkreten Zuständen $C_{\mathbb{A}'_M} = Q_{1_0} \times \ldots \times Q_{1_j} \times \ldots \times Q_{n_0} \times \ldots \times Q_{n_k}$, einer Menge von abstrakten Zuständen $\Psi_{\mathbb{A}'_M} = C_{\mathbb{A}'_M}$ und einer Konkretisierungsfunktion $[\![\cdot]\!]_\mathbb{Q} : C_\mathbb{Q} \to \Psi_\mathbb{Q}$, die der Identitätsfunktion entspricht.
- Sei $e \in E$ eine CFA-Kante, dann enthält die Transferrelation $\rightsquigarrow_{\mathbb{A}'_M}$ den Programmzustandsübergang

$$(Q'_1, \ldots, Q'_n) \rightsquigarrow^e_{\mathbb{A}'_M} (Q''_1, \ldots, Q''_n)$$
$$\text{gdw.}$$
$$\exists (q', e, q'') \in \Delta_i : q' \in Q'_i \Leftrightarrow q'' \in Q''_i \text{ für } i = 1, \ldots, n$$

- $merge_{\mathbb{A}'_M}((Q_1, \ldots, Q_n), (Q'_1, \ldots, Q'_n)) = (Q_1 \cup Q'_1, \ldots, Q_n \cup Q'_n)$ mit $i = 1, \ldots, n$
- $stop_{\mathbb{A}'_M}((Q_1, \ldots, Q_n), reached) = (\exists (Q'_1, \ldots, Q'_n)' \in reached : Q_i = Q'_i$ mit $i = 1, \ldots, n)$

Die komponierte CPA $\text{CPA}_{\mathbb{R}'_M}$, die sich aus der Komposition der Location-CPA $\text{CPA}_\mathbb{L}$, der Prädikaten-CPA $\text{CPA}_\mathbb{P}$ und der Automaten-CPA $\text{CPA}_{\mathbb{A}'_M}$ ergibt, unterscheidet sich zur CPA $\text{CPA}_{\mathbb{R}_M}$ darin, dass immer dann vereinigt wird, wenn Kontrollflüsse im CFA zusammenfließen, unabhängig davon in welchen TGA-Zuständen sich die betrachteten TGAs in den zu vereinigenden Programmzuständen befinden, wie beispielsweise in Abbildung 3.18 zu sehen ist.

Beispiel 3.41 (Fehlende Pfadsensitivität der CPA $\text{CPA}_{R'_M}$)**.** Abbildung 3.18 zeigt den ARG_M für die TGAs tga_2 und tga_3 der GaussianSum-Funktion mit Vereinigung von Programmzuständen. Aufgrund der fehlenden Pfadsensitivität kann es bei der Ableitung von Testfällen dazu kommen, dass ein Testfall abgeleitet wird, der ein Testziel nicht abdeckt (vgl. Abschnitt 3.25). Wie in Abschnitt 3.2

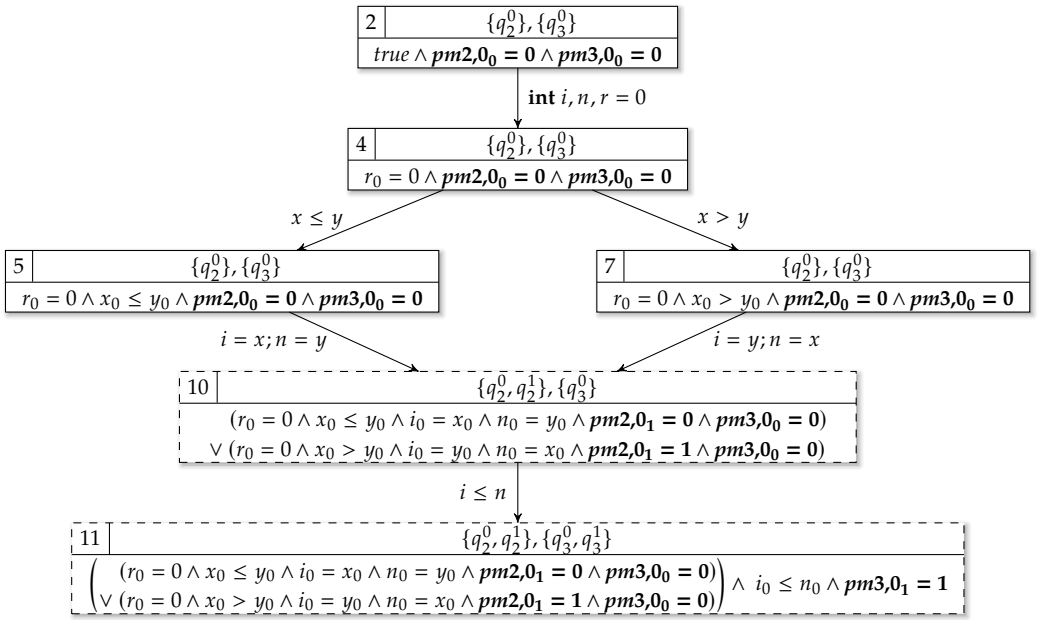


Abbildung 3.19: Ausschnitt des ARG_M der GaussianSum-Funktion für die Testziele tga_2 und tga_3 mit Vereinigung von Programmzuständen und Testpfadmarkern

ist die Ursache dafür, dass aufgrund der Vereinigung von Programmzuständen im ARG_M Daten- und Kontrollflussinformationen verloren gehen können und dadurch nicht mehr identifizierbar ist, auf welchem ARG_M-Pfad ein TGA einen Endzustand erreicht hat. Beispielsweise ist der TGA tga_2 in Programmzustand 10 in einem Endzustand. Beim Ableiten eines Testfalls aus der Pfadbedingung von Programmzustand 10 kann es vorkommen, dass ein Testfall abgeleitet wird, der den ARG_M-Pfad $2 - 4 - 5 - 10$ traversiert, der das Testziel tga_2 nicht abdeckt. Zum Beispiel der Testfall $([x = 1, y = 2])$.

Zum Erhalt der Pfadsensitivität und somit zur korrekten Testfallgenerierung in ARG_M mit verbesserter Vereinigung von Programmzuständen können wie in Abschnitt 3.2.2 Testpfadmarkierungen verwendet werden. Dabei werden TGA-Transitionen mit Testpfadmarkierungsvariablen versehen, die dynamisch während der ARG_M-Konstruktion in den ARG_M eingepflegt werden. Das Einpflegen der Testpfadmarkierungsvariablen wird von der LOOM-CPA erledigt, die parallel zu der CPA $\text{CPA}_{R'_M}$ ausgeführt wird (vgl. Abschnitt 3.2.2).

Beispiel 3.42 (Ableiten von konkreten Testfällen aus einem ARG_M mit Testpfadmarkierungen)**.** Abbildung 3.19 zeigt den ARG_M, der für die Testziele tga_2 und tga_3 für die GaussianSum-Funktion mit Hilfe der CPA $\text{CPA}_{\mathbb{R}'_M}$ konstruiert wurde. Aus diesem ARG_M sollen Testfälle zur Abdeckung der Testziele tga_2 und tga_3 abgeleitet werden. Das Testziel tga_3 ist in Programmzustand 11 sowohl über den ARG_M-Pfad $2 - 4 - 5 - 10 - 11$ als auch über den ARG_M-Pfad $2 - 4 - 7 - 10 - 11$ erfüllt. Zur Ableitung eines konkreten Testfalls zur Abdeckung des Testziels tga_3 wird die Pfadbedingung des Programmzustands 11 mit der Testpfadmarkierungsvariablen $\boldsymbol{pm3,0_1 = 1}$ konjugiert, die der CFA-

Kante 10 $\xrightarrow{i \leq n}$ 11 entspricht (vgl. Beispiel 3.31). Ein möglicher Testfall ist somit ($[x = 1, y = 1]$).

Das Testziel tga_2 ist in Programmzustand 10 durch den ARG_M-Pfad 2 – 4 – 7 – 10 erfüllt. Zur Ableitung eines Testfalls, der tatsächlich den ARG_M 2 – 4 – 7 – 10 traversiert wird die Pfadbedingung von Programmzustand 10 mit der Testpfadmarkierungsvariablen $\boldsymbol{pm2{,}0_1 = 1}$ konjugiert. Eine Lösung muss also der zweiten Hälfte der Pfadbedingung von Programmzustand 10 entsprechen und erfüllt somit tatsächlich das Testziel tga_2. Ein möglicher Testfall ist ($[x = 2, y = 1]$).

3.4 Experimentelle Evaluation

In diesem Abschnitt evaluieren wir die Test-Suite-Generierung mit Multi-Property-Checking (*TM*) und die aus der Test-Suite-Generierung resultierenden Test-Suiten in Bezug auf Effizienz und Effektivität. Die Effizienz der Test-Suite-Generierung beschreibt dabei die CPU-Zeit, die für die Generierung einer Test-Suite benötigt wird. Die Effizienz einer generierten Test-Suite ist hingegen die Größe der Test-Suite (Anzahl der Testfälle), welche als Indikator für den Aufwand zur späteren Ausführung der Test-Suite auf dem zu testenden Programm dient. Die Effektivität der Test-Suite-Generierung wird durch die Abdeckungsraten angegeben, die durch die generierten Test-Suiten erreicht werden. Die Effektivität der generierten Test-Suiten wird hingegen durch ihre Fehlerdetektionsraten bestimmt.

Für die Evaluation wurden die Techniken zur Test-Suite-Generierung aus den Abschnitten 3.1, 3.2 und 3.3 basierend auf dem Model-Checking-Framework CPAchecker [BK11, BHT07] implementiert und es wurde eine Serie von Experimenten für verschiedene Software-Systeme durchgeführt. Wir untersuchen dabei den Einfluss der Verwendung der symbolischen Pfadsensitivität (SPS, vgl. Abschnitt 3.2) und von verschiedenen Partitionsgrößen auf die Test-Suite-Generierung (vgl. Abschnitt 3.3).

- Wir untersuchen den Einfluss von SPS auf die Test-Suite-Generierung, welche den Programmzustandsraum durch das Zusammenfassen von Programmzustände im ARG_T verkleinern kann. Hierfür vergleichen wir die Test-Suite-Generierung mit Multi-Property-Checking mit SPS (TM_S) und ohne SPS (TM_N).

- Wir vergleichen die Auswirkungen der Partitionierung der Menge aller Testziele auf die Test-Suite-Generierung. Hierfür teilen wir die Menge der Testziele auf Partitionen verschiedener Größe auf und untersuchen die Auswirkungen dieser Partitionierungen auf die Effizienz und die Effektivität der Test-Suite-Generierung. Neben verschiedenen Partitionsgrößen werden dabei die Test-Suite-Generierung bei der immer nur ein Testziel betrachtet wird (TS_S) und bei der immer alle Testzielen gleichzeitig betrachtet werden (TA_S) für einen Vergleich herangezogen.

Für eine bessere Verständlichkeit dieses Abschnittes kann eine Übersicht der evaluierten Test-Suite-Generierungsstrategien sowie deren Bezeichnungen in Tabelle 3.1 gefunden werden.

Tabelle 3.1: Übersicht über die evaluierten Test-Suite-Generierungsstrategien

SPS \ # Testziele	1 (*TS*)	X (*TM*)	Alle (*TA*)
Ohne SPS (*N*)	TS_N	$TM_{N,X}$	TA_N
SPS (*S*)	TS_S	$TM_{S,X}$	TA_S

Forschungsfragen

Das Ziel der Evaluation ist die Beantwortung der zwei Forschungsfragen **RQ1** und **RQ2**, die sich auf die Parametrisierung unserer Test-Suite-Generierungstechnik beziehen. Zur Parametrisierung der Test-Suite-Generierungstechnik verwenden wir die zwei unabhängigen Variablen *SPS* (d. h. ein- und ausschalten der symbolischen Pfadsensitivität) und die *Partitionsgröße* (d. h. die Verwendung verschiedener Partitionsgrößen).

- **RQ1 (Effizienz):** Erhöht die Wiederverwendung von *TM* zur Test-Suite-Generierung die Effizienz verglichen mit TS_S und TA_S?
- **RQ2 (Effektivität):** Wie beeinflusst die Verwendung von *TM* die Effektivität der Test-Suite-Generierung verglichen mit TS_S und TA_S?

Die Forschungsfrage **RQ1** zielt auf die Untersuchung des Einflusses verschiedener Parametrisierungen der Test-Suite-Generierung auf die Effizienz der Test-Suite-Generierung und der generierten Test-Suiten ab. Somit verwenden wir die zwei Variablen *durchschnittliche CPU-Zeit pro Zielsystem*, d. h. die CPU-Zeit die durchschnittlich zur Generierung einer Test-Suite für ein Programm benötigt wird und *Test-Suite-Größe* zur Bestimmung der verschiedenen Facetten von Effizienz.

Die Forschungsfrage **RQ2** zielt auf die Untersuchung des Einflusses verschiedener Parametrisierungen der Test-Suite-Generierung auf die Effektivität der Test-Suite-Generierung und der generierten Test-Suiten ab. Somit verwenden wir die zwei Variablen *Abdeckungsrate* und *Fehlerdetektionsrate* zur Bestimmung der Effektivität.

Zum Beantworten dieser Forschungsfragen evaluieren wir drei testbare Hypothesen. Die erste Hypothese untersucht die Auswirkungen der Betrachtung aller Testziele gleichzeitig (TA_S) bei der Test-Suite-Generierung im Gegensatz zur separaten Betrachtung aller Testziele (TS_S). Beide Ansätze verwenden dabei SPS.

H1: Das Verarbeiten aller Testziele gleichzeitig mit TA_S erhöht die Effizienz und Effektivität verglichen mit der separaten Betrachtung aller Testziele mit TS_S.

Die zweite Hypothese vergleicht die Test-Suite-Generierung unter Verwendung von SPS mit der Test-Suite-Generierung ohne SPS. Deshalb wird zusätzlich die Test-Suite-Generierung für alle Testziele gleichzeitig ohne SPS (TA_N) in die Untersuchung mit einbezogen.

H2: Test-Suite-Generierung mit SPS unter Verwendung von TA_S erhöht die Effizienz und die Effektivität verglichen mit der Test-Suite-Generierung ohne SPS unter Verwendung von TA_N.

Die dritte Hypothese untersucht die Auswirkungen verschieden großer Partitionen auf die Test-Suite-Generierung (TM_S). Wie für die Hypothese **H1** werden wieder nur Ansätze mit SPS betrachtet.

H3: TM_S verbessert sowohl die Effizienz als auch die Effektivität der Test-Suite-Generierung. Der Grad der Verbesserung ist dabei abhängig von der Anzahl der Testziele, die zusammen betrachtet werden.

Experimentaufbau

Fallstudien. Die Zielsysteme für die Evaluation entstammen einer Menge von Funktionen und Programmen von der Competition of Software Verification (SV-Comp), welche Linux-Kernel-Treiber und Neuimplementierungen von Unix-Kommandozeilentools für Ressourcen-beschränkte Systeme umfassen [Bey12]. Im Detail betrachten wir 324 C-Programme, welche durchschnittlich 1541 Zeilen Code enthalten. Zur Messung der Fehlerdetektionsrate haben wir Fehler in den Programmen mit Testzielen in Beziehung gesetzt, welche durch eine Test-Suite abgedeckt werden müssen, damit die Fehler während der Testausführung gefunden werden. Zu diesem Zweck stehen uns umfangreiche Informationen über Programmierfehler in den Zielsysteme zur Verfügung [Bey12]. Die betrachteten Zielsysteme enthalten im Schnitt 4 Fehler.

Diese Zielsysteme stellen eine viel genutzte Sammlung von Community-Benchmarks dar, welche sowohl eine große Bandbreite an C-Programmstrukturen enthalten als auch seltene Spezialfälle. Weiterhin werden viele dieser Zielsysteme häufig in der Praxis verwendet (z. B. BusyBox[1]).

Tool-Unterstützung. Die Konzepte zur Test-Suite-Generierung mit Multi-Property-Checking aus den Abschnitten 3.2 und 3.3 wurden basierend auf dem C-Model-Checking-Framework CPAchecker[2] [BK11, BHT07] implementiert. Für die Spezifikation von Abdeckungskriterien verwenden wir FQL[3] [HSTV10, HTSV10]. Als SMT-Solver kam SMTInterpol zum Einsatz.

Abdeckungskriterien. Als Abdeckungskriterien verwenden wir *Anweisungsüberdeckung* (C_0) und *Bedingungsüberdeckung* (C_3), zwei der relevantesten Abdeckungskriterien in der Praxis [SL12]. Auf die Zielsysteme angewendet ergeben sich die folgenden Zahlen: für C_0 enthalten die Zielsysteme 7966 Testziele und für C_3 8254 Testziele, wobei jedes Zielsystem im Schnitt 25 Testziele für jedes Abdeckungskriterium enthält.

Experiment-Design. Als Baseline verwenden wir TS_S und TA_S wie sie oben beschrieben sind. Die Experimentkonfigurationen für TS_S und TA_S enthalten als Parametrisierung die An-/Abwahl der SPS und TM kann über verschiedene Partitionsgrößen parametrisiert werden. Als Partitionsgrößen verwenden wir die fixen Werte 1 (TS_S), 10, 20 und 50 sowie die prozentualen Werte 25%, 50%, 75% und 100%

[1] `https://busybox.net/`
[2] `https://cpachecker.sosy-lab.org/`
[3] `http://forsyte.at/software/fshell/fql/`

(TA_S) aller Testziele. Wir haben die 10 Experimentkonfigurationen (d. h. TS_S und TA_S jeweils mit/ohne SPS kombiniert und TM_S mit verschiedenen Partitionsgrößen), die sich aus diesen Überlegungen ergeben, auf alle Zielsysteme sowohl für C_0- als auch für C_3-Abdeckung angewendet.

Aufbau der Messungen. Die gemessenen CPU-Zeiten enthalten den kompletten Test-Suite-Generierungsprozess einschließlich der TGA-Konstruktion aus den FQL-Queries, der CPA-Initialisierung und der Ausführung des Testgenerators selbst. Für jedes Testziel sichern wir eine Mindest-CPU-Zeit von 60s zu. Somit erhält jede Testzielpartition einen *Partitions-Timeout*, der durch die Formel *Timeout pro Testziel * Anzahl der Testziele pro Partition* berechnet wird. Der globale Timeout für den gesamten Test-Suite-Generierungsprozess wird pro Zielsystem auf 3 Stunden gesetzt. Die Evaluation wurde auf Computern mit einem Intel Xeon E5-2650 (2GHz) mit 30GB RAM ausgeführt. Die Verwendung von 30GB RAM ist für die betrachteten Zielsysteme ausreichend, sodass kein Swapping von Speicherinhalten im Arbeitsspeicher während der Test-Suite-Generierung durchgeführt werden muss.

Datensammlung. Für alle 10 Experimente haben wir die folgenden Daten zum Beantworten der Hypothesen und der Forschungsfragen gesammelt. Zur Evaluation der Effizienz haben wir die benötigte *CPU-Zeit* für die Test-Suite-Generierung und die Anzahl der generierten Testfälle gemessen. Bezüglich der Effektivität haben wir die Abdeckungsraten und die Fehlerdetektionsraten der generierten Test-Suiten für jedes Experiment bestimmt.

Um die Hypothese **H1** zu beantworten, vergleichen wir die für TS_S und TA_S gemessenen Daten. Dabei untersuchen wir die Auswirkungen auf die Test-Suite-Generierung, wenn alle Testziele separat betrachtet werden im Gegensatz zu dem Fall, dass alle Testziele gleichzeitig betrachtet werden. Bezüglich der Hypothese **H2** vergleichen wir die für die Test-Suite-Generierung gemessenen Daten, wenn alle Testziele gleichzeitig mit (TA_S) und ohne (TA_N) SPS betrachtet werden und untersuchen den Einfluss von SPS auf die Test-Suite-Generierung. Zur Beantwortung der Hypothese **H3** vergleichen wir die Effizienz und die Effektivität von TM_S mit den festen Partitionsgrößen 10 ($TM_{S,10}$), 20 ($TM_{S,20}$) ($TM_{S,50}$) sowie mit den prozentualen Partitionsgrößen 25% ($TM_{S,25\%}$), 50% ($TM_{S,50\%}$) und 75% ($TM_{S,75\%}$) mit der Baseline (TS_S und TA_S), um den Einfluss der verschiedenen Partitionsgrößen auf die Test-Suite-Generierung zu untersuchen. Ob die Ergebnisse relevant sind bestimmen wir mit Hilfe des T-Tests.

Ergebnisse

Die gemessenen Daten für die Hypothesen **H1** und **H2** sind in der Tabelle 3.2 zusammengefasst und die gemessenen Daten für Hypothese **H3** in der Tabelle 3.3.

Hypothese **H1**. Im Durchschnitt ist die Test-Suite-Generierung mit TA_S langsamer (77,5s pro Zielsystem) als mit TS_S (70,5s pro Zielsystem), aber TA_S deckt einzelne Testziele schneller ab (1,09s pro Testziel) als TS_S (1,2s pro Testziel). Abbildung 3.20 zeigt, dass TA_S im allgemeinen schneller ist als TS_S. Dabei nehmen die

Tabelle 3.2: Vergleich von TS_S und TA mit (TA_S) und ohne SPS (TA_N)

	TS_S	TA_S	TA_N
∅ CPU-Zeit pro Fallstudie (s)	70,5	77,5	243,6
∅ CPU-Zeit pro Testziel (s)	1,2	1,09	2,15
∅ # Testfälle pro Fallstudie	24,9	21,4	20,0
∅ Abdeckungsrate (%)	99,7	99,8	99,2
∅ Fehlerdetektionsrate (%)	89,9	86,8	85,1

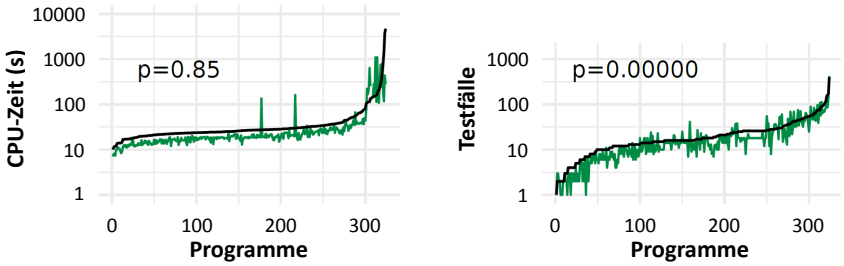


Abbildung 3.20: Vergleich zwischen der Verarbeitung aller Testziele gleichzeitig (TA_S, grüne Linie) und separat (TS_S, schwarze Linie)

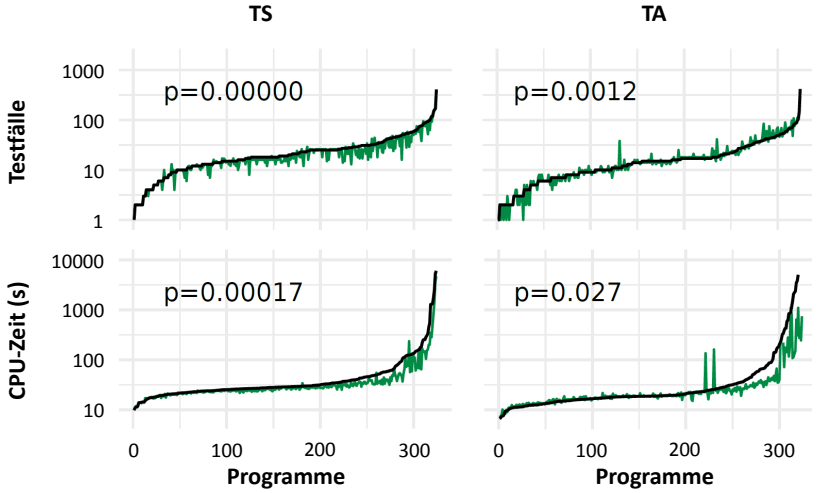


Abbildung 3.21: Auswirkungen von SPS auf *TS* und *TA* (grün: SPS eingeschaltet, schwarz: SPS ausgeschaltet)

Performance-Vorteile von TA_S gegenüber TS_S bei steigender CPU-Zeit jedoch ab. Dies ist auch der Grund, warum TA_S im Durchschnitt langsamer ist als TS_S, aber dennoch einzelne Testziele durchschnittlich schneller Abdeckt. Weiterhin zeigt Abbildung 3.20, dass TA_S im Durchschnitt signifikant ($p = 0,0$) kleinere Test-Suite (∅ 21,4 Testfälle) als TS_S (∅ 24,9) produziert.

Hypothese **H2**. Die für die Test-Suite-Generierung benötigte CPU-Zeit pro Zielsystem ist signifikant ($p = 0.027$) kleiner, wenn TA_S (77,5s) verwendet wird verglichen mit TA_N (243,6s). Abbildung 3.21 zeigt, dass TA_S für Zielsysteme, für welche die Test-Suite-Generierung im Allgemeinen länger dauert, mehr CPU-Zeit gegen-

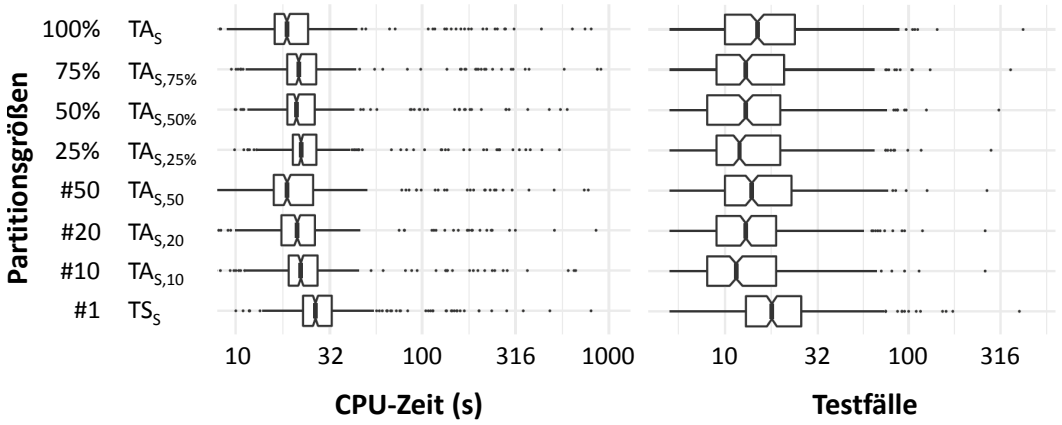


Abbildung 3.22: Auswirkungen der Partitionierung von Testzielen auf die Test-Suite-Generierung

über TA_N spart. Jedoch sind diese Ergebnisse nicht signifikant ($p = 0.85$). Bezüglich der Größe der generierten Test-Suiten produziert TA_N kleinere Test-Suiten (20,3) als TA_S (21,4). Die Effektivität bzgl. der durchschnittlichen Testzielabdeckung und der durchschnittlichen Anzahl von gefundenen Fehlern ist für TA_N und TA_S sehr ähnlich. TA_N deckt im Durchschnitt 99,2% aller Testziele ab und die generierten Test-Suiten finden im Durschschnitt 85,1% aller Fehler und TA_S deckt 99,8% aller Testziele ab und die generierten Test-Suiten finden im Durschschnitt 86,8% der Fehler.

Hypothese **H3**. Abbildung 3.22 zeigt einen Vergleich der CPU-Zeiten für die Test-Suite-Generierung mit TM_S für verschiedene Partitionsgrößen sowie die entsprechenden Test-Suite-Größen. Im Durchschnitt generiert TM_S Test-Suiten pro Zielsystem in kürzerer Zeit (zwischen 45,8 für $TM_{S,20}$ und 76,4s für $TM_{S,75\%}$) als TA_S (77,5s) in allen Fällen und in kürzerer Zeit als TS_S (70,5s) in allen Fällen außer für $TM_{S,75\%}$. Dabei sind die Instanziierungen von TM_S, die Test-Suiten im Durchschnitt in kürzerer Zeit generieren nicht immer die Instanziierungen, die auch einzelne Testziele im Durchschnitt am schnellsten Abdecken. Beispielsweise benötigt $TM_{S,75\%}$ (76,4s) im Durchschnitt geringfügig weniger CPU-Zeit für die Generierung einer Test-Suite als TA_S (77,5s). Dabei benötigt TM_S im Durschschnitt jedoch 1,26s um ein Testziel abzudecken, wohingegen TA_S nur 1,09s benötigt. Der Grund dafür ist, dass gerade für kleinere Programme mit weniger Testzielen die Betrachtung aller Testziele parallel effizienter ist, als die die Betrachtung von Teilmengen von Testzielen wie es auch in Abbildung 3.22 zu sehen ist. Auch gibt es bei TA_S weniger Fälle bei denen die Test-Suite-Generierung sehr viel länger gedauert hat verglichen mit den einzelnen Instanziierungen von TM_S als es bei $TM_{S,75\%}$ der Fall ist. Im Durchschnitt generiert TM_S kleinere Test-Suiten (zwischen 16,0 Testfälle pro Zielsystem für $TM_{S,10}$ und 18,7 für $TM_{S,75\%}$) für alle Partitionsgrößen verglichen mit der Baseline (24,9 Testfälle pro Zielsystem für TS_S und 21,4 für TA_S). Die Effektivität ist stabil für alle betrachteten Partitionsgrößen mit durchschnittlichen Abdeckungsraten zwischen 99,6% ($TM_{S,10}$,$TM_{S,20}$,$TM_{S,75\%}$) und 99,8% ($TM_{S,25\%}$,$TM_{S,50\%}$,TA_S) so-

Tabelle 3.3: Effekte verschiedener Partitionsgrößen auf die Test-Suite-Generierung, Gruppiert nach allen und nach großen Fallstudien (mehr als 100 Testziele).

	TS_S	$TM_{S,10}$	$TM_{S,20}$	$TM_{S,50}$	$TM_{S,25\%}$	$TM_{S,50\%}$	$TM_{S,75\%}$	TA_S
Alle Fallstudien								
∅ CPU-Zeit pro Fallstudie (s)	70,5	46,5	45,8	46,0	46,2	50,5	76,4	77,5
∅ CPU-Zeit pro Testziel (s)	1,2	0,97	0,97	0,89	1,02	0,99	1,26	1,09
∅ # Testfälle pro Fallstudie	24,9	16,0	17,0	18,6	17,1	17,8	18,7	21,4
∅ Abdeckungsrate (%)	99,7	99,6	99,6	99,7	99,8	99,8	99,6	99,8
∅ Fehlerdetektionsrate (%)	89,9	88,1	87,2	86,6	87,6	88,6	86,9	86,8
Große Fallstudien								
∅ CPU-Zeit pro Fallstudie (s)	335,7	161,5	131,5	168,2	166,6	211,5	341,6	425,4
∅ CPU-Zeit pro Testziel (s)	1,48	0,78	0,69	0,71	0,88	0,85	1,81	2,21
∅ # Testfälle pro Fallstudie	79,0	52,2	55,0	57,5	57,6	62,4	66,4	77,4
∅ Abdeckungsrate (%)	98,9	98,6	98,8	98,1	99,1	99,6	98,0	98,8
∅ Fehlerdetektionsrate (%)	93,4	92,3	93,3	90,3	94,7	93,1	91,9	92,1

wie durchschnittlichen Fehlerdetektionsraten zwischen 86,6% ($TM_{S,50}$) und 89,9% (TS_S).

Diskussion

Als erstes diskutieren wir die Ergebnisse bzgl. der Hypothese **H1**, d. h. die Auswirkungen auf die Test-Suite-Generierung, wenn wir alle Testziele gleichzeitig betrachten im Gegensatz zu der Betrachtung von jedem Testziel separat. Im Durchschnitt ist die Effizienz der Test-Suite-Generierung bezogen auf die CPU-Zeit für TA_S etwas schlechter verglichen mit TS_S. Abbildung 3.20 zeigt jedoch, dass die Effizienz der Test-Suite-Generierung mit TA_S gerade für Programme mit weniger Testziele durchweg besser ist vergleich mit TS_S. Erst bei zunehmender Komplexität der Zielsysteme verringern sich die positiven Auswirkungen der Betrachtung aller Testziele gleichzeitig auf die CPU-Zeit. Weiterhin generiert TA_S signifikant kleinere Test-Suiten als TS_S. Bezüglich der Effektivität hat TA_S fast die gleiche Testzielabdeckungsrate und Fehlerdetektionsrate wie TS_S. Somit folgern wir, dass TA_S die Effizienz im Vergleich zu TS_S erhöht, wobei die Effektivität ähnlich hoch ist.

Um die Auswirkungen von SPS auf die Test-Suite-Generierung für alle Testziele gleichzeitig zu untersuchen, vergleichen wir TA_N und TA_S in der Hypothese **H2**. Wenn alle Zielsysteme betrachtet werden, dann generiert TA_S Test-Suiten im Durchschnitt in 77,46s, wohingegen TA_N 243,6s im Durchschnitt benötigt. Die Anzahl der generierten Testfälle ist dabei für TA_N geringfügig kleiner, wohingegen die Effektivität unter Verwendung von TA_S leicht besser ist. Insgesamt verbessert TA_S die Test-Suite-Generierung stark verglichen mit TA_N bzgl. der CPU-Zeit. Anzumerken ist dabei, dass die positiven Auswirkungen durch SPS mit zunehmender Größe der Zielsysteme bezogen auf die Anzahl der Testziele immer stärker werden. Für Zielsysteme mit mehr als 100 Testzielen benötigt TA_S im Durchschnitt nur 335,7s pro Fallstudie für die Test-Suite-Generierung im Gegensatz zu TA_N mit 425,4s. Aus diesen Beobachtungen folgern wir, dass SPS für größere Zielsysteme effizienter ist als für Zielsysteme mit weniger als 100 Testzielen.

Bezüglich der Hypothese **H3** erhöht TM_S die Effizienz im Sinne von CPU-Zeit und Test-Suite-Größe für alle Partitionsgrößen verglichen mit TS_S. Die Effektivität ist dabei für alle Konfigurationen stabil. Verglichen mit TA_S steigert TM_S die Effizienz der Test-Suite-Generierung bzgl. der durchschnittlich benötigten CPU-Zeit um 40% und bzgl. der Größe der generierten Test-Suiten um 15%. TM_S ist in allen Fällen durchschnittlicher schneller als TA_S für die Zielsysteme, die wir betrachtet haben und nur für den Fall $TM_{S,75\%}$ langsamer als TS_S. Weiterhin sind die Test-Suiten, die mit TM_S generiert werden, immer noch wesentlich kleiner als die durch TA_S generierten Test-Suiten. Wieder ist die Effektivität für TM_S für alle Partitionsgrößen TS_S und TA_S stabil.

Unsere Experimente zeigen einen großen Einfluss der verschiedenen Partitionsgrößen auf die Performance der Test-Suite-Generierung. Dennoch gibt es keine Partitionsgröße bei der die Performance durchgängig am besten ist. Zum Beispiel generiert in manchen Fällen $TM_{S,20}$ Test-Suiten schneller als TM_{S10} oder umgekehrt. Weiterhin kann aus den Experimenten geschlossen werden, dass es keine offensichtliche Strategie gibt, nach der eine Partitionsgröße, die am besten für ein Zielsystem passt, ausgewählt werden kann.

Im Allgemeinen führt die Betrachtung mehrerer Testziele gleichzeitig zu einer besseren Effizienz. Dennoch scheint es eine kritische Größe von Testzielpartitionen zu geben, ab der diese Vorteile durch zusätzlichen Berechnungsaufwand durch die Betrachtung mehrere Testziele gleichzeitig beeinträchtigt werden. Diese Auswirkungen sind beobachtbar, wenn $TM_{S,20}$ mit $TM_{S,75\%}$ und TA_S speziell für größere Zielsysteme verglichen wird. Im Durchschnitt generiert $TM_{S,20}$ schneller Test-Suiten als $TM_{S,75\%}$, welches wiederum schneller als TA_S ist. Bezüglich TA_S wird die Effizienz insbesondere durch zusätzlichen Berechnungsaufwand zur Behandlung mehrerer Testziele gleichzeitig negativ beeinflusst.

Eine weitere Erkenntnis ist, dass in unseren Experimenten das ausgewählte Abdeckungskriterium die Effizienz und Effektivität von TM_S nicht bemerkbar beeinflusst. Wir haben ähnliche Ergebnisse bzgl. der Effizienz und Effektivität sowohl für C_0-Abdeckung als auch für C_3-Abdeckung für alle Zielsysteme erhalten.

Zusammengefasst ist eine mittlere Partitionsgröße am wahrscheinlichsten für die Verringerung von Berechnungsaufwand während der Test-Suite-Generierung geeignet, wobei die Effektivität davon nicht beeinflusst wird. Bis zu einem gewissen Grad verringern zu große Partitionsgrößen die positiven Auswirkungen auf die Effizienz der Test-Suite-Generierung durch zusätzlichen Berechnungsaufwand, der durch die Betrachtung vieler Testziele gleichzeitig entsteht.

Gefährdung der Validität

Interne Validität. Die konfigurierbare Technik zur Test-Suite-Generierung basierend auf Multi-Property-Checking bietet einige neue Parametereinstellungen zur Justierung der Test-Suite-Generierung. In der Evaluation haben wir uns auf den Einfluss der Partitionsgröße und der SPS auf die Effizienz und Effektivität der Test-Suite-Generierung mit vordefinierten Timeout-Werten beschränkt, um die Ergebnisse nicht durch zu viele Parameter zu verfälschen und somit die interne Validität der Evaluierung zu gefährden. Die Verteilung der Testziele auf mehrere Partitionen geschah gemäß der Reihenfolge, in der die Testziele aus einer FQL-Query abgeleitet wurden. Diese Partitionierung der Testziele könnte die interne Validität gefährden. Dennoch glauben wir, das die Anwendung von Partitionierungsstrategien unsere Ergebnisse potenziell noch weiter verbessern würde [ABM+16]. Eine weitere Gefährdung der internen Validität könnte aus der Beschränkung auf die gewählten Abdeckungskriterien resultieren. Wir erwarten jedoch ähnliche Ergebnisse für andere Abdeckungskriterien basierend auf unserer Erfahrung.

Externe Validität. Eine Gefährdung der externen Validität besteht in dem fehlenden Vergleich mit anderen Ansätzen. Aufgrund der Neuartigkeit des Ansatzes war ein Vergleich mit ähnlichen Ansätzen jedoch noch nicht möglich. Stattdessen wurde als Baseline der Evaluation die Test-Suite-Generierung so konfiguriert, dass die Partitionierung der Testzielmenge praktisch deaktiviert wird, d. h. als Baseline wurde die Test-Suite-Generierung für jedes Testziel einzeln (*TS*) und für alle Testziele zusammen (*TA*) verwendet. Wir sind eher daran interessiert zu analysieren, welche potenziellen Verbesserungen die Partitionierung von Testzielen und SPS bewirkt, anstatt den Test-Suite-Generierungsansatz mit anderen Ansätzen zu vergleichen.

Die Auswahl der Zielsysteme kann immer als Gefährdung der externen Validität angesehen werden. Dennoch stellen die ohnehin schon anspruchsvollen Systeme für das Benchmarking von Software-Verifizierern von der SVComp für ein noch herausforderndes Problem wie die Test-Suite-Generierung eine solide Basis für unsere Experimente dar. Zusätzlich enthalten viele Systeme aus dem SVComp-Korpus Code aus realen Systeme (z. B. BusyBox).

3.5 Verwandete Arbeiten

In diesem Abschnitt diskutieren wir zu den in diesem Kapitel behandelten Ansätzen und Techniken verwandte Arbeiten. Aufgrund der umfangreichen Forschungsarbeiten im Bereich automatisierter Testgenerierung, werden wir den Fokus auf verwandte Arbeiten legen, die sehr eng mit unseren Ansätzen in Beziehung stehen.

Concolic Testing und Fuzz Testing

Im Gegensatz zu Testgenerierungsansatz, die Branching Conditions und Randomisierung zur Steuerung der Traversierung des Programmzustandsraums einer IUT verwenden, wie Concolic-Testing-Ansätze [GKS05, MS07, GLM08a, CJW15] und Fuzz-Testing-Ansätze [GLM08b], wird die Traversierung des Programmzustandsraums einer IUT in unserem Ansatz durch Testziele geleitet, die als endliche Automaten (TGAs) spezifiziert werden. Die Testziele können dabei flexibel durch ein vom Benutzer spezifiziertes Abdeckungskriterium festgelegt werden. Weiterhin stellen die oben genannten Testgenerierungsansätze im Gegensatz zu unserem Testgenerierungsansatz die Abdeckung von Testzielen nicht sicher. Die Verwendung von Automaten zur Steuerung von Model-Checking-basierter Testgenerierung ist jedoch nicht neu, z. B. werden Automaten zur Spezifikation von Testzielen in [LPY97, BHJP04, HSTV10] verwendet.

Model-Checking-basierte Testgenerierung

Verschiedene Model-Checking-basierte Testgenerierungsansätze wurden in der Literatur vorgeschlagen, z. B. basierend auf SAL2 [HdMR04], UPPAAL [LPY97, BHJP04], Java PathFinder [VPK04], BLAST [BCH+04] und CPAchecker [BHTV13]. Keiner dieser Ansätze verwendet für die Testgenerierung Multi-Property-Checking [ABM+16] zur Generierung von Testfällen für mehrere Testziele während eines Model-Checking-Laufs. Neben der Verwendung von Multi-Property-Checking für die Testgenerierung, untersuchen wir im Gegensatz zu den oben genannten Arbeiten verschieden große Menge von Testzielen, die gleichzeitig während eines Model-Checking-Laufs berücksichtigt werden.

Abstraktion und statische Analysen

Während Abstraktion und statische Analysen wiederholt zum Steuern der Testgenerierung betrachtet werden [BNRS08, CMW16, CS05, CS06, CJW15, DGH16, GNRT10, GHK+06, HdMR04, HSH+00, McM10, VPK04], unterstützt keiner dieser Ansätze Multi-Property-Checking für die Testgenerierung oder evaluiert den

Einfluss von simultan betrachteten Teilmengen von Testzielen während der Testgenerierung. Diese Ansätze betrachten entweder nur ein einzelnes Testziel in einem Schritt der Testgenerierung oder die gesamte Menge aller Testziele.

Search-based Testgenerierung

Die Defizite der Adressierung eines einzigen Testziels zur gleichen Zeit wurden im Kontext von Search-based Testdatengenerierung untersucht [HKL+10, FA13, AF14]. Bei Search-based Testgenerierung wird die komplette Menge von Testzielen gleichzeitig untersucht mit dem Ziel kleine, aber effektive Test-Suiten abzuleiten. Search-based Testgenerierungsansätze unterstützen jedoch nur eine kleine, fixe Menge von Abdeckungskriterien, z. B. Zweigabdeckung, während wir eine vielseitigere Menge von Abdeckungskriterien basierend auf Testzielautomaten unterstützen. Weiterhin ist die technische Grundlage von Search-based Testgenerierung und Model-Checking-basierter Testgenerierung grundsätzlich verschieden. Während die Anzahl von generierten Testfälle grundsätzlich relevant ist, fokussieren wir uns auf den Performance-Aspekt von automatisierter Testgenerierung und anstatt nur die gesamte Test-Suite zu berücksichtigen, untersuchen wir in unserer Arbeit verschieden große Teilmengen von Testzielen, die gleichzeitig während einer Erreichbarkeitsanalyse berücksichtigt werden.

SAT-based Testgenerierung

Der Testgenerator FShell [HSTV08, HSTV09] kodiert eine Menge von Testzielautomaten in endliche Abwicklungen eines Programms als SAT-Term und versucht zu bestimmen, welche durch die Testzielautomaten spezifizierten Testziele erfüllbar sind. Aufgrund der begrenzten Natur der Darstellung des Problems als SAT-Term, kann FShell nicht bestimmen welche Testziele unerfüllbar sind. Unser CPAchecker-basierter-Ansatz erlaubt es hingegen zu bestimmen, wann ein Testziel unerfüllbar ist und unterstützt weiterhin verschiedene Arten von Erreichbarkeitsanalysen.

Wiederverwendung von Erreichbarkeitsanalyseergebnissen

Das Teilen von Informationen zwischen Testgenerierungsläufen für verschiedene Testziele wurde bereits für SAT-Solving-basierte Testgenerierung für Hardware-Designs [MC09, QCM10] und für Model-Checking-basierte Testgenerierung [BHTV13, Hol13, BLB+15] betrachtet. Im Gegensatz zu unserer Arbeit, betrachten diese Testansätze nicht verschiedene Testziele gleichzeitig, sondern untersuchen wie Informationen an die Testfallgenerierung zur Wiederverwendung für das nächste Testziel übergeben werden können.

3.6 Zusammenfassung und Ausblick

White-Box-Ansätze zur automatisieren Generierung von Komponententests auf Basis von Model-Checking skalierten bisher im Allgemeinen nicht bei der Generierung von Test-Suiten für große Software-Programme aufgrund der Ausführung von vielen anspruchsvollen und redundanten Berechnungen. In diesem Kapitel wurde eine Reihe von neuen Techniken vorgestellt, welche die existierenden Ideen und Techniken zur Testgenerierung mit Model-Checking von Beyer et al. [BCH+04, BHTV13] und Holzer [Hol13] mit den Zielen (1) die Testgenerierungstechniken mit Multi-Property-Checking zu kombinieren und (2) die Vereinigung von Programmzuständen im Programmzustandsraum zu optimieren.

(1) Durch die Kombination von Testgenerierungstechniken mit Multi-Property-Checking ist es möglich, während eines Model-Checker-Laufs (d. h. während einer ARG-Konstruktion) für mehrere Testziele in einem Lauf eine Test-Suite zu generieren und dadurch eine systematische Wiederverwendung von Testgenerierungsergebnissen zwischen Testzielen zu ermöglichen. Die Evaluationsergebnisse zeigen, dass durch diese systematische Wiederverwendung von Testgenerierungsergebnissen der Anteil von redundanten Berechnungsschritten während der Testgenerierung deutlich reduziert werden kann.

(2) Mit Hilfe der Verbesserung der Vereinigung von Programmzuständen im Programmzustandsraum kann die Größe des zu berechnenden Programmzustandsraums während der Testgenerierung reduziert werden, indem ähnliche Programmzustände häufiger zusammengefasst werden können. Um dem Verlust an Pfadsensitivität entegegen zu wirken, der durch das vermehrte Vereinigen von Programmzuständen entstehen kann, haben wir das Konzept der symbolischen Pfadsensitivität verwendet. Symbolische Pfadsensitivität erlaubt es, Pfade im Programmzustandsraum eindeutig Testzielen zuzuordnen, sodass trotz der Vereinigung von Programmzuständen korrekte Test-Suiten generiert werden.

Die Ergebnisse der Evaluation zeigen, dass die Test-Suite-Generierung basierend auf Multi-Property-Checking durch die Betrachtung mehrerer Testziele in einem Lauf im Allgemeinen zu einer besseren Testeffizienz (Generierungsdauer und Test-Suite-Größe) bei einer ähnlichen Testeffektivität der generierten Test-Suiten (Abdeckungsrate und Fehlerdetektionsrate) führt. Jedoch können nicht einfach alle Testziele gleichzeitig betrachtet werden, da es eine kritische Anzahl von Testzielen gibt ab der die Vorteile der Test-Suite-Generierung mit Multi-Property-Checking durch zusätzlichen Berechnungsaufwand beeinträchtigt werden, der durch die gleichzeitige Betrachtung sehr vieler Testziele entsteht. Eine weitere Erkenntnis der Evaluation ist, dass das ausgewählte Abdeckungskriterium die Testeffizienz und Testeffektivität von Test-Suite-Generierung mit Multi-Property-Checking nicht bemerkbar beeinflusst. Wir haben ähnliche Ergebnisse bzgl. der Testeffizienz und Testeffektivität sowohl für C_0-Abdeckung als auch für C_3-Abdeckung für alle Fallstudien erhalten, somit der Ansatz wahrscheinlich unabhängig vom gewählten Testverfahren, solange die Testziele als TGAs ausdrückbar sind.

Insgesamt zeigt die Evaluation einige sehr aufschlussreiche Ergebnisse bzgl. der Auswirkungen von SPS und Multi-Property-Checking auf die Test-Suite-Generierung. In zukünftigen Forschungsarbeiten können weitere Evaluationen durchge-

führt werden, um noch weitere Parameter des vorgestellten Test-Suite-Generierungsansatzes zu untersuchen. Ein Beispiel hierfür wäre es, die Auswirkungen der Verwendung anderer Abdeckungskriterien auf die Effizienz und die Effektivität der Test-Suite-Generierung zu evaluieren. Weiterhin sollten zusätzliche Zielsysteme betrachtet werden.

Als Basis für die Implementierung dient das konfigurierbare Model-Checking-Framework CPAchecker, welches hunderte Konfigurationsoptionen enthält, mit denen die Test-Suite-Generierung potenziell weiter optimiert und an die Anforderungen unterschiedlicher Zielsysteme angepasst werden kann. Dies umfasst ebenfalls die Verwendung verschiedener Model-Checking-Technologien, wie explizites Model-Checking oder Bounded Model-Checking. In zukünftigen Forschungsarbeiten können die Auswirkungen der verschiedenen Konfigurationsoptionen und Model-Checking-Techniken des CPAchecker-Frameworks auf die Test-Suite-Generierung untersucht werden, um die Test-Suite-Generierung weiter zu verbessern.

Neben weiteren Evaluationen gibt es auch technische Aspekte, die in zukünftigen Arbeiten untersucht und verbessert werden können. Ein Aspekt ist dabei die Partitionierung und Priorisierung der Testzielmenge. Aktuell wird die Testzielmenge zufällig auf Partitionen einer bestimmten Größe verteilt. Zukünftig könnte eine systematische Partitionierung der Testzielmenge die Effizienz der Test-Suite-Generierung weiter verbessern. Beispielsweise könnten Testziele gemäß des Kontrollflusses im CFA partitioniert werden, sodass Testziele, die auf den gleichen Programmpfaden liegen, gemeinsam betrachtet werden. Weiterhin könnte die Partitionierung dynamisch erfolgen, sodass Testziele basierend auf bereits erlangten Informationen über die IUT des aktuellen Test-Suite-Generierungsprozesses partitioniert werden.

4

AUTOMATISIERTE TESTGENERIERUNG FÜR SOFTWARE-PRODUKTLINIEN

Hochkonfigurierbare Software-Systeme sind heutzutage in vielen modernen Anwendungsdomänen essentiell, um mit immer individualisierteren Kundenwünschen und domänenspezifischen Anforderungen an die Systeme umzugehen. SPLE ist zu einer Schlüsseltechnologie im Umgang mit der Variabilität in hochkonfigurierbaren Systemen geworden [CN01]. SPLE zielt darauf ab, SPLs bestehend aus gleichartigen, aber dennoch unterscheidbaren Software-Produkten auf Basis einer gemeinsamen Kernplattform zu entwickeln. Dementsprechend haben SPLs in den vergangenen Jahren Einzug in verschiedenste Anwendungsdomänen gefunden, z. B. in Fahrzeug-, Informations- und mobilen Systemen [WCKK06].

Wie bei der Entwicklung einzelner Software-Produkte, ist die Qualitätssicherung eine aufwändige und kritische Phase von SPLE. Im Gegensatz zur Software-Entwicklung einzelner Produkte umfasst die Qualitätssicherung beim SPLE nicht nur das Validieren und Verifizieren einer einzelnen Software, sondern einer ganzen Familie von Software [McG01, PS08, SH11]. Somit erweitert die Qualitätssicherung von SPLE die Qualitätssicherung der Software-Entwicklung einzelner Produkte um eine weitere Dimension: dem Variantenreichtum. Da es aufgrund der hohen Anzahl von Produkten (Varianten) üblicherweise unmöglich ist, Qualitätssicherung für jedes Software-Produkt separat (produktweise) durchzuführen, wurden in der Literatur verschiedene Techniken vorgeschlagen, um Qualitätssicherung für ganze Software-Familien auf einmal (familienbasiert) durchzuführen [AtBFG11, COLS11, CHSL11, CCP+12, BLB+15]. Wie für einzelne Software-Produkte (vgl. Kapitel 3), ist die Testgenerierung eine vielversprechende Technik zur Automatisierung der familienbasierten Qualitätssicherung von SPLs. Dabei sollte der Grundsatz von SPLE, Artefakte während der Entwicklung von SPLs möglichst systematisch zwischen Software-Produkten wiederzuverwenden, auch bei der familienbasierten Qualitätssicherung beibehalten werden. Für die automatisierte, familienbasierte Testgenerierung heißt dies insbesondere die Wiederverwendung von (Teil-)Ergebnissen von Erreichbarkeitsanalysen (z. B. Testfälle) während der Testgenerierung (vgl. Kapitel 3) zwischen Testzielen und Software-Produkten zu maximieren.

In diesem Kapitel werden wir die Techniken zur automatisierten Testgenerierung aus Kapitel 3 für SPLs so erweitern, dass Test-Suiten familienbasiert generiert werden können. Das Ziel dabei ist es, Test-Suiten unter Betrachtung kompletter Software-Familien zu generieren, sodass mit den generierten Test-Suiten sämtliche Software-Produkte einer SPL abgedeckt werden können. Dabei liegt der Fokus insbesondere auf einer möglichst hohen Wiederverwendung von Ergebnissen von

```
1  int Calc_SPL(int x, int y) {
2  #if SUM
3      int r = 0;
4  #elif FAC
5      int r = 1;
6  #endif
7
8  #if CHK
9      if (x < 1 || y < 1 || y < x) {
10         return -1;
11     }
12 #endif
13
14     while (x <= y) {
15 #if SUM
16         r = r + x;
17 #elif FAC
18         r = r * x;
19 #endif
20         x = x + 1;
21     }
22
23     return r;
24 }
```

Abbildung 4.1: Überarbeitete SPL-Implementierung der Calc-SPL

Erreichbarkeitsanalysen zwischen Testzielen und Software-Produkten, sodass der Aufwand für die Test-Suite-Generierung im Vergleich zu produktweisen Testableitung möglichst reduziert wird (vgl. Forschungsherausforderungen FH2.1 und FH2.2).

4.1 White-Box-Testgenerierung für Software-Produktlinien

In diesem Abschnitt wird der Ansatz zur automatisierten White-Box-Testgenerierung aus Abschnitt 3.1 um die Möglichkeit erweitert, Test-Suiten zur Abdeckung von kompletten Software-Familien zu generieren. Das Ziel dabei ist es, familienbasiert Test-Suiten für alle Produkte einer SPL in einem Durchlauf zu generieren und nicht nur für einzelne Software-Produkte in separaten Durchläufen [BLB+15]. Der Fokus der familienbasierten Testgenerierung liegt dabei auf einer möglichst hohen Wiederverwendung von Testartefakten (z. B. generierte Testfälle) zwischen Software-Produkten und Testzielen.

4.1.1 *Software-Produktlinien-Implementierungen*

Die in diesem Kapitel beschriebenen Techniken werden anhand der Calc-SPL illustriert, die in Abschnitt 2.3.4 eingeführt wurde. Damit die Darstellung von Artefakten der Calc-SPL (z. B. CFAs und ARGs) in diesem Kapitel handhabbar bleibt, werden wir die Implementierung der Calc-SPL aus Abbildung 2.12 an zwei Stellen verändern. Zum einen wird der Vergleich $x \leq y$ zwischen den Variablen x und y aus

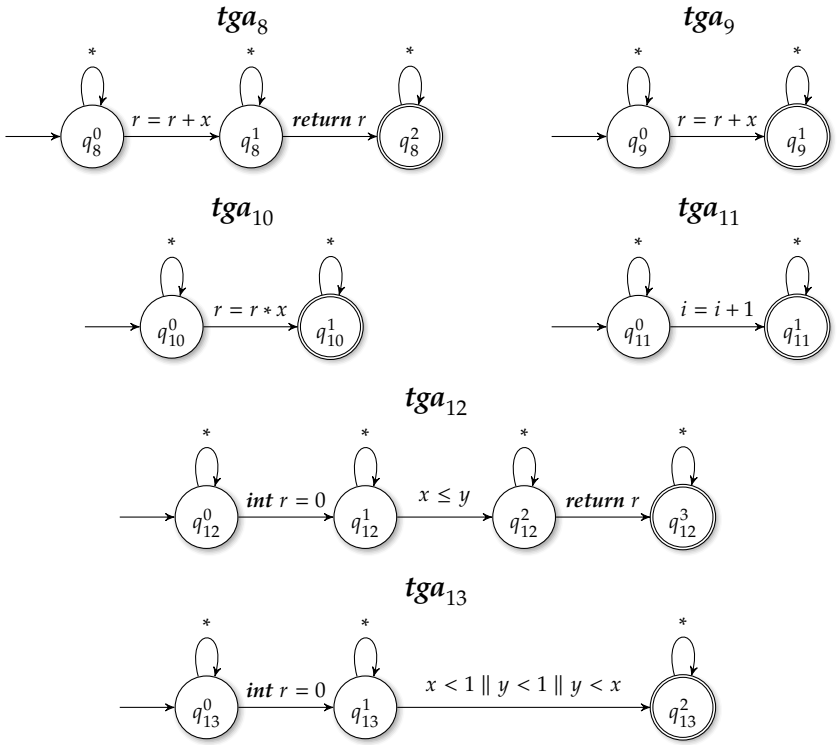


Abbildung 4.2: Testzielautomaten für die Calc-SPL

allen Produkten der Calc-SPL entfernt. Dieser Vergleich sorgt dafür, dass die While-Schleife immer vom kleineren Wert von x und y zum größeren Wert durchlaufen wird (vgl. Zeile 16-20 in Abbildung 2.12). Das Entfernen des Vergleichs führt dazu, dass die Funktion Calc-SPL für Eingaben, für die $y < x$ gilt, falsche Ergebnisse berechnet. Beispielsweise wird für das Produkt *Sum* für die Eingabe $[x = 2, y = 1]$ der falsche Wert $[r = 0]$ zurückgegeben. Um diese fehlerhafte Berechnung in den Produkten der Calc-SPL zu vermeiden, die das Feature *Chk* implementieren, wird als zweite Änderung an der Calc-SPL die Überprüfung auf valide Eingabewerte um den Vergleich $y < x$ erweitert (vgl. Zeile 11 in Abbildung 2.12). Durch diese Erweiterung wird für Eingaben, für die $y < x$ gilt, der Wert -1 zurückgegeben, wenn das Feature *Chk* ausgewählt ist. Basierend auf diesen zwei Änderungen wurden weiterhin die Vorkommen der Variablen i durch die Variable x ersetzt und die Vorkommen der Variablen n durch die Variable y, da die Variablen i und n nicht mehr benötigt werden. Die aus diesen Änderungen resultierende SPL-Implementierung ist in Abbildung 4.1 dargestellt.

Neben den Testzielen aus Abbildung 3.5 werden zur Illustration der Testgenerierungsansätze für SPLs in diesem Kapitel weitere Testziele betrachtet, die für die überarbeiteten SPL-Implementierung der Calc-SPL spezifiziert wurden. Abbildung 4.2 zeigt TGAs für die SPL-Implementierung der Calc-SPL aus Abbildung 4.1. Der TGA tga_8 spezifiziert ein Testziel, das durch eine vollständige Programmausführung abgedeckt wird, welche die Anweisung $r = r + x$ ausführt. Die TGAs tga_9, tga_{10} und tga_{11} spezifizieren jeweils ein Testziel, das die Ausführung der Anweisungen $r = r + x$, $r = r * x$ oder $i = i + 1$ fordert und der TGA tga_{12} spezifiziert

ein Testziel, das durch eine vollständige Programmausführung abgedeckt wird, welche die Anweisung ***int*** $r = 0$ gefolgt von der Anweisung $x \leq y$ traversiert. Der TGA tga_{13} spezifiziert ein Testziel, das die Traversierung der Anweisung ***int*** $r = 0$, gefolgt von der Bedingung $x < 1 \parallel y < 1 \parallel y < x$ durch einen Testfall fordert.

SPL-Implementierungen erweitern die Programmdarstellungen aus dem Abschnitt 3.1.2 um Features. Zu diesem Zweck erweitert eine SPL-Implementierung eine Produktimplementierung durch das Hinzufügen einer Menge von schreibgeschützten, Boole'schen Feature-Variablen $V_F \rightarrow \mathbb{B}$ zusätzlich zur Menge von Programmvariablen V_P. Somit ist die Menge aller Variablen einer SPL-Implementierung $V = V_P \uplus V_F$ die disjunkte Vereinigung der Variablenmengen V_F und V_P. Für die SPL-Implementierung der Calc-SPL ist $V_P = \{x, y, r\} \subseteq V$ die Teilmenge der Programmvariablen und $V_P = \{\textsc{Sum}, \textsc{Fac}, \textsc{Chk}\} \subseteq V$ die Teilmenge der Feature-Variablen. Dementsprechend ist $V = \{x, y, r, \textsc{Sum}, \textsc{Fac}, \textsc{Chk}\}$ die Menge aller Variablen der Calc-SPL. Weiterhin ist $V_I = \{x, y\} \subseteq V_P$ die Menge der Eingabevariablen und $V_O = \{r\} \subseteq V_P$ die Menge der Ausgabevariablen der Calc-SPL.

Die Menge der Programmoperationen $\mathcal{L}_{assume}(V) = \mathcal{L}_{p\text{-}assume}(V_P) \cup \mathcal{L}_{f\text{-}assume}(V_F)$ eines Programms, die das Teilalphabet der Prädikate über V darstellt, wird in die Teilmengen $\mathcal{L}_{p\text{-}assume}(V_P)$ und $\mathcal{L}_{f\text{-}assume}(V_F)$ unterteilt. Dabei stellt $\mathcal{L}_{p\text{-}assume}(V_P)$ die Menge der Prädikate über den Programmvariablen V_P dar und $\mathcal{L}_{f\text{-}assume}(V_F)$ die Menge von (partiellen) Feature-Konfigurationen über der Menge der Feature-Variablen V_F. Eine Feature-Konfiguration $\gamma \in \mathcal{L}_{f\text{-}assume}(V_F)$ entspricht dem Feature-Modell $fm \in \mathcal{L}_{f\text{-}assume}(V_F)$ der SPL, d. h. $\gamma \models fm$. Dabei kann das Feature-Modell selbst als propositionale Formel $fm \in \mathcal{L}_{f\text{-}assume}(V_F)$ über der Feature-Menge V_F dargestellt werden [Bat05] (vgl. Tabelle 2.1). Auf Basis dieser Notationen kann die Repräsentation einer SPL-Implementierung wie in Kapitel 3 für einzelne Produkte als CFA erfolgen.

Beispiel 4.1 (CFA der Calc-SPL)**.** Abbildung 4.3 zeigt den CFA der Implementierung der Calc-SPL. Mit Prädikaten aus $\mathcal{L}_{f\text{-}assume}(V_F)$ über den Feature-Variablen V_F beschriftete CFA-Kanten sind gestrichelt dargestellt und repräsentieren Entscheidungen über Feature-Variablen. Beispielsweise ist die Anweisung ***int*** $r = 0$ nur in Produkten enthalten, die das Feature *Sum* implementieren, da die Anweisung ***int*** $r = 0$ nur über CFA-Pfade erreichbar ist, die vorher die CFA-Kante $2 \xrightarrow{\textsc{Sum}} 3$ traversieren.

Viele Anweisungen einer SPL-Implementierung sollen nur in bestimmten Produkten, d. h. für bestimmte Feature-Kombinationen, ausgeführt werden, wie in Beispiel 4.1 demonstriert wurde. Verallgemeinert bedeutet das, dass auch bestimmte vollständige Programmausführungen nur auf bestimmten Produkten ausgeführt werden können. Um zu beschreiben, auf welchen Produkten eine Programmausführung ausführbar ist, verwenden wir eine sogenannte *Presence Condition*. Eine Presence Condition $pc \in \mathcal{L}_{f\text{-}assume}(V_F)$ ist ein logischer Ausdruck über der Menge der Feature-Variablen V_F einer SPL-Implementierung und somit Teil der Region eines Programmzustands. Die Region eines Programmzustands einer SPL-Implementierung besteht aus zwei Teilen [BLB+15]:

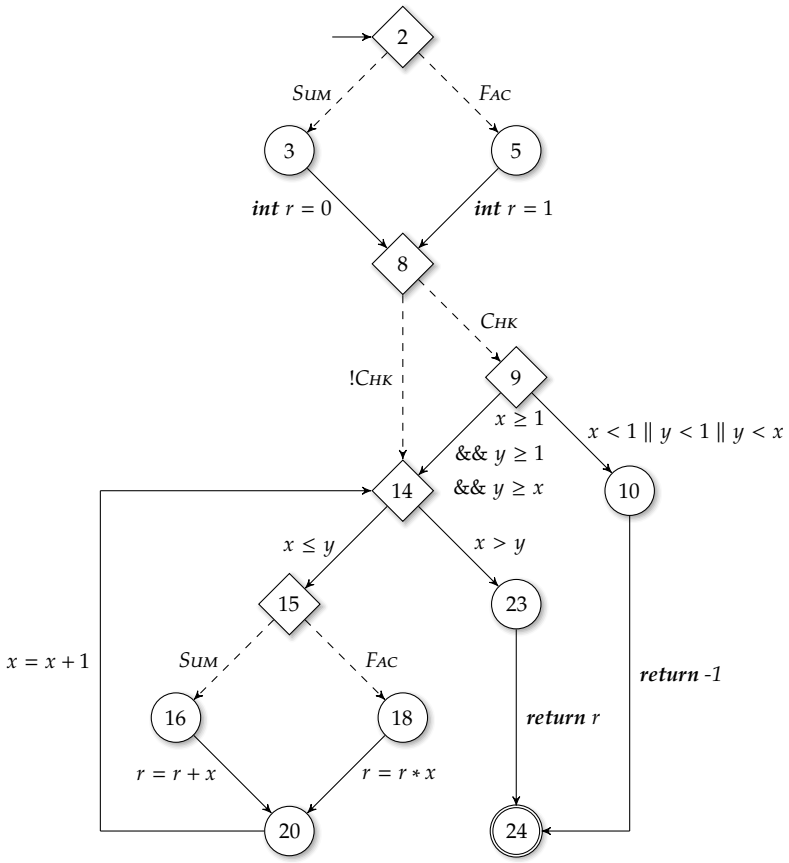


Abbildung 4.3: CFA der SPL-Implementierung der Calc-SPL

1. Eine Pfadbedingung $\phi \in \mathcal{L}_{p\text{-}assume}(V_P)$ über den Programmvariablen V_P, die wie bisher die Variablenwerte beschreibt, mit denen ein Programmzustand erreicht werden kann.

2. Eine Presence Condition $pc \in \mathcal{L}_{f\text{-}assume}(V_F)$ über den Feature-Variablen V_F, die beschreibt, in welchen Produkten ein Programmzustand erreichbar ist.

Die Aufspaltung der Region eines Programmzustands in zwei Teile kann aufgrund der Trennung von Programm- und Feature-Variablen in logischen Ausdrücken in C-Programmen geschehen, die häufig in SPL-Implementierung vorgenommen und in dieser Arbeit für eine verständlichere und übersichtlichere Darstellung gefordert wird. Somit beschreibt ein Programmzustand (ℓ, r) mit $r = (\phi \wedge pc)$ einer symbolischen SPL-Programmausführung, welche Programm-Location $\ell \in L$ mit welchen Variablenbelegungen $\phi \in \mathcal{L}_{p\text{-}assume}(V_P)$ auf welchen Produkten $pc \in \mathcal{L}_{f\text{-}assume}(V_F)$ erreicht werden kann. Die Presence Condition des initialen Programmzustands einer Programmausführung wird mit der aussagenlogischen Repräsentation des Feature-Modells *fm* einer SPL initialisiert, sodass die Programmausführung auf valide Programmzustandsübergänge beschränkt wird, die das Feature-Modell erfüllen. Dadurch werden Programmzustandsübergänge ausgeschlossen, die zu einem Programmzustand mit einer nicht erfüllbaren Presence Condition führen, da der Programmzustand nicht Teil eines Produktes sein kann.

Beispiel 4.2 (Symbolische Programmausführung einer SPL)**.** Eine Übersetzung des Feature-Modells der Calc-SPL aus Abbildung 2.10 gemäß der Übersetzungsregeln aus Tabelle 2.1 in einen logischen Ausdruck ist

$$fm_C = \mathit{CALC} \wedge (\mathit{CHK} \rightarrow \mathit{CALC}) \wedge \big(\mathit{SUM} \leftrightarrow (\neg \mathit{FAC} \wedge \mathit{CALC}) \wedge \mathit{FAC} \leftrightarrow (\neg \mathit{SUM} \wedge \mathit{CALC})\big).$$

Für das Feature-Modell fm_C und den Pfad

$$2 \xrightarrow{\mathit{SUM}} 3 \xrightarrow{\textbf{\textit{int}}\ r = 0} 8 \xrightarrow{!\mathit{CHK}} 14 \xrightarrow{x \leq y} 15 \xrightarrow{\mathit{SUM}} 16 \xrightarrow{r = r + x} 20 \xrightarrow{x = x + 1} 14 \xrightarrow{x > y} 23 \xrightarrow{\textbf{\textit{return}}\ r} 24$$

des CFAs der Calc-SPL aus Abbildung 4.3 ist

$$\begin{aligned}
& (2, \mathit{true} \wedge fm_C) \\
\xrightarrow{\mathit{SUM}}\ & (3, \mathit{true} \wedge fm_C \wedge \mathit{SUM}) \\
\xrightarrow{\textbf{\textit{int}}\ r = 0}\ & (8, r_0 = 0 \wedge fm_C \wedge \mathit{SUM}) \\
\xrightarrow{!\mathit{CHK}}\ & (14, r_0 = 0 \wedge fm_C \wedge \mathit{SUM} \wedge \neg \mathit{CHK}) \\
\xrightarrow{x \leq y}\ & (15, r_0 = 0 \wedge x_0 \leq y_0 \wedge fm_C \wedge \mathit{SUM} \wedge \neg \mathit{CHK}) \\
\xrightarrow{\mathit{SUM}}\ & (16, r_0 = 0 \wedge x_0 \leq y_0 \wedge fm_C \wedge \mathit{SUM} \wedge \neg \mathit{CHK}) \\
\xrightarrow{r = r + x}\ & (20, r_0 = 0 \wedge x_0 \leq y_0 \wedge r_1 = r_0 + x_0 \wedge fm_C \wedge \mathit{SUM} \wedge \neg \mathit{CHK}) \\
\xrightarrow{x = x + 1}\ & (14', r_0 = 0 \wedge x_0 \leq y_0 \wedge r_1 = r_0 + x_0 \wedge x_1 = x_0 + 1 \wedge fm_C \wedge \mathit{SUM} \\
& \wedge \neg \mathit{CHK}) \\
\xrightarrow{x > y}\ & (23, r_0 = 0 \wedge x_0 \leq y_0 \wedge r_1 = r_0 + x_0 \wedge x_1 = x_0 + 1 \wedge x_1 > y_0 \wedge fm_C \\
& \wedge \mathit{SUM} \wedge\ \neg \mathit{CHK}) \\
\xrightarrow{\textbf{\textit{return}}\ r}\ & (24, r_0 = 0 \wedge x_0 \leq y_0 \wedge r_1 = r_0 + x_0 \wedge x_1 = x_0 + 1 \wedge x_1 > y_0 \wedge fm_C \\
& \wedge \mathit{SUM} \wedge\ \neg \mathit{CHK})
\end{aligned}$$

eine symbolische Programmausführung. Die Presence Condition für diese symbolische Programmausführung ist $fm_C \wedge \mathit{SUM} \wedge \neg \mathit{CHK}$. Somit sind alle konkreten Programmausführungen, die durch die symbolische Programmausführung charakterisiert werden, auf allen Produkten ausführbar, deren Feature-Auswahl die Presence Condition $fm_C \wedge \mathit{SUM} \wedge \neg \mathit{CHK}$ erfüllen. Im Fall der Calc-SPL ist das nur das Produkt mit der Feature-Auswahl $\{\mathit{SUM}\}$.

Im Allgemeinen, ist eine symbolische Programmausführung nicht immer nur auf einem Produkt ausführbar. Die Presence Condition der partiellen symbolischen Programmausführung

$$\begin{aligned}
& (2, \mathit{true} \wedge fm_C) \\
\xrightarrow{\mathit{SUM}}\ & (3, \mathit{true} \wedge fm_C \wedge \mathit{SUM}) \\
\xrightarrow{\textbf{\textit{int}}\ r = 0}\ & (8, r_0 = 0 \wedge fm_C \wedge \mathit{SUM})
\end{aligned}$$

ist $fm_C \wedge \mathit{Sum}$. Somit ist diese partielle Programmausführung für alle Produkte gültig, die der Presence Condition $fm_C \wedge \mathit{Sum}$ entsprechen. Im Fall der Calc-SPL sind das die zwei Produkte $\{\mathit{Sum}\}$ und $\{\mathit{Sum}, \mathit{Chk}\}$.

Für den CFA-Pfad

$$2 \xrightarrow{\mathit{Sum}} 3 \xrightarrow{\textbf{\textit{int}}\ r = 0} 8 \xrightarrow{!\mathit{Chk}} 14 \xrightarrow{x \leq y} 15 \xrightarrow{\mathit{Fac}} 18$$

des CFAs der Calc-SPL ist die partielle symbolische Programmausführung

$$\begin{aligned}
& (2, \mathit{true} \wedge fm_C) \\
\xrightarrow{\mathit{Sum}} \quad & (3, \mathit{true} \wedge fm_C \wedge \mathit{Sum}) \\
\xrightarrow{\textbf{\textit{int}}\ r = 0} \quad & (8, r_0 = 0 \wedge fm_C \wedge \mathit{Sum}) \\
\xrightarrow{!\mathit{Chk}} \quad & (14, r_0 = 0 \wedge fm_C \wedge \mathit{Sum} \wedge \neg \mathit{Chk}) \\
\xrightarrow{x \leq y} \quad & (15, r_0 = 0 \wedge x_0 < y_0 \wedge fm_C \wedge \mathit{Sum} \wedge \neg \mathit{Chk}) \\
\xrightarrow{\mathit{Fac}} \quad & (18, r_0 = 0 \wedge x_0 < y_0 \wedge fm_C \wedge \mathit{Sum} \wedge \neg \mathit{Chk} \wedge \mathit{Fac})
\end{aligned}$$

eine *invalide* Programmausführung, die auf keinem Produkt ausgeführt werden kann. Der Programmzustandswechsel von Programmzustand 15 nach Programmzustand 18 ist für kein Produkt gültig, da die Presence Condition von Programmzustand 18 $fm_C \wedge \mathit{Sum} \wedge \neg \mathit{Chk} \wedge \mathit{Fac}$ nicht erfüllbar ist (Sum und Fac schließen sich im Feature-Modell fm_C gegenseitig aus). Der Programmzustandsübergang von Programmzustand 15 nach Programmzustand 18 wird durch den Strongest-Postcondition-Operator sp_V ausgeschlossen, der fordert, dass für einen Programmzustandsübergang $r \xrightarrow{e} r'$ mit $e = (\ell, a, \ell')$ die Bedingung $r \wedge a \models r'$ gelten muss (vgl. Definition 3.7). Somit werden Programmzustandsübergänge im Programmzustandsraum, die nicht in mindestens einem Produkt einer SPL erlaubt sind, durch den Strongest-Postcondition-Operator ausgeschlossen.

Zur Darstellung aller symbolischen Programmausführungen einer SPL-Implementierung kann ein ARG gemäß Definition 3.9 konstruiert werden. Da sowohl Pfadbedingung und Presence Condition in der Region eines Programmzustands kodiert sind, muss die ARG-Definition nicht angepasst werden.

Beispiel 4.3 (ARG für eine SPL-Implementierung)**.** Abbildung 4.4 zeigt einen Ausschnitt des ARGs für die SPL-Implementierung der Calc-SPL. Jeder Programmzustand (ℓ, r) besteht aus einer Programm-Location ℓ und einer Region $r = (\phi \wedge pc)$. Die Presence Condition pc der Region r wird dabei im oberen rechten Teil eines Programmzustands und die Pfadbedingung ϕ im unteren Teil eines Programmzustands dargestellt. Der initiale Programmzustand $(2, \mathit{true} \wedge fm_C)$ wird mit dem Feature-Modell fm_C initialisiert, sodass nur Programmzustände Teil des ARG sind, die das Feature-Modell fm_C erfüllen. In Beispiel 4.2 wurde beispielsweise beschrieben, dass der durchgestrichene Programmzustand $(18, r_0 = 0 \wedge x_0 < y_0 \wedge fm_C \wedge \mathit{SUM} \wedge \mathit{Chk} \wedge \mathit{Fac})$ kein valider

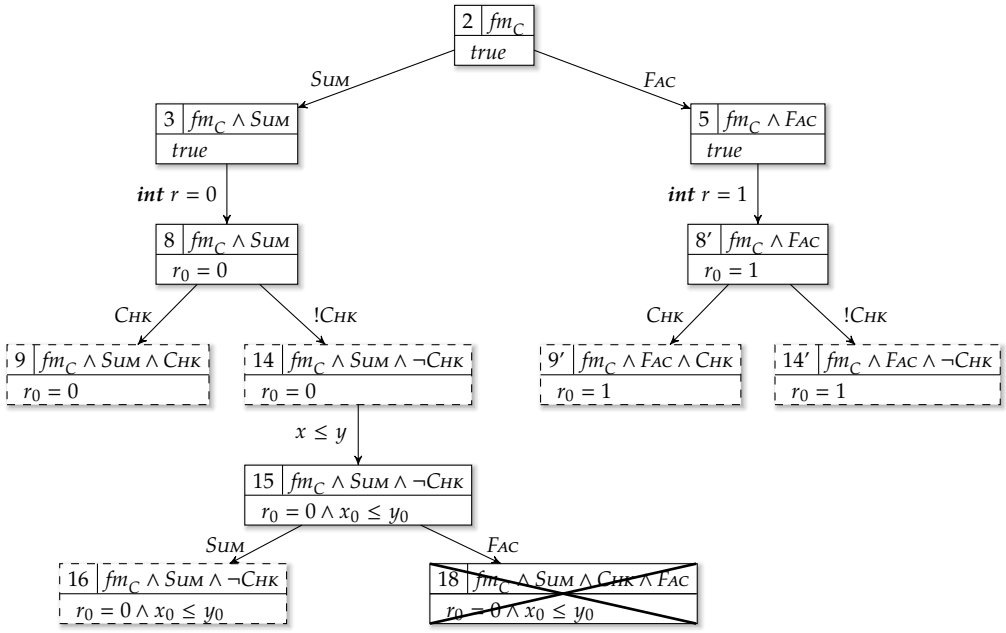


Abbildung 4.4: Ausschnitt des ARGs der Calc-SPL

Nachfolger des Programmzustands (15, . . .) ist, da die Presence Condition des Programmzustands 18 nicht erfüllbar ist.

Durch die Kodierung der Presence Condition in der Region eines Programmzustands lassen sich die Definitionen für CFAs, Strongest-Postcondition und ARGs aus Kapitel 3 für SPL-Implementierungen ohne Anpassung weiter verwenden. Im Folgenden werden die Besonderheiten bei der Testfallgenerierung für SPL-Implementierungen auf Basis der in diesem Abschnitt beschriebenen Artefakte erläutert.

4.1.2 *Testfallgenerierung für Software-Produktlinien*

Die Verwendung von SPL-Implementierungen zur familienbasierten Testfallableitung erlaubt es, alle Produkte einer SPL während der Testfallableitung zusammen zu betrachten. Dadurch muss nicht mehr jedes Produkt separat betrachtet werden, wie es beim produktweisen Testen der Fall ist (vgl. Abschnitt 2.3.4). Der Fokus der Testableitung anhand von SPL-Implementierungen liegt auf der Wiederverwendung von Teilergebnissen von Erreichbarkeitsanalysen und Testartefakten zwischen Software-Produkten, z. B. von Testfällen und ARGs. Durch die Wiederverwendung kann ein Testfall, der für ein Testziel auf Basis einer SPL-Implementierung erstellt wurde, potenziell auf mehreren Software-Produkten ausgeführt werden und das Testziel somit auf mehreren Software-Produkten abdecken.

Beispiel 4.4 (Familienbasierte Testfallausführung)**.** Testfälle können auf SPL-Implementierungen genauso wie auf einzelnen Software-Produkten ausgeführt werden. Die dabei traversierten Code-Teile hängen allerdings nicht nur von der Testeingabe ab, wie es bei einzelnen Software-Produkten der Fall ist, sondern auch von der gewählten Produktkonfiguration. Zum Beispiel traver-

siert der Eingabevektor $I = [x = 2, y = 3]$ die Additionsanweisung in Zeile 16 der SPL-Implementierung der Calc-SPL, wenn das Feature *Sum* ausgewählt wurde und die Multiplikationsanweisung in Zeile 18, wenn das Feature *Fac* ausgewählt wurde. Dies gilt ebenfalls für Produkte, die das Feature *Chk* implementieren, auf denen zusätzlich die If-Bedingung in Zeile 9 ausgeführt wird. Die Return-Anweisung in Zeile 10 wird für den Eingabevektor I hingegen in keinem Produkt ausgeführt.

Weiterhin kann sich die *Ausgabe* der Calc-SPL-Funktion abhängig von den ausgewählten Features für den gleichen Eingabevektor unterscheiden. Wenn das Feature *Sum* ausgewählt wurde, ist die Ausgabe für den Eingabevektor I $[r = 5]$, wohingegen die Ausgabe $[r = 6]$ ist, wenn das Feature *Fac* ausgewählt wurde. Die An-/Abwahl des Features *Chk* hat dagegen für den Eingabevektor I keine weitere Auswirkung auf die Ausgabe der Testfälle.

Wie das vorherige Beispiel zeigt, müssen zum Umgang mit Verhaltensvariabilität während der Testfallableitung und der Testfallausführung neben der Implementierung auch Testfälle variabilitätsgewahr sein. Das bedeutet, dass für einen abgeleiteten Testfall bekannt sein muss, auf welchen Produkten der Testfall ausführbar ist. Deswegen werden Testfälle für SPLs im Folgenden um eine Presence Condition erweitert, welche die Produkte charakterisiert, auf denen ein Testfall ausführbar ist.

Definition 4.5 (SPL-Testfall (basierend auf [COLS11]))**.** Gegeben sei ein Programm mit einer Menge von Programmvariablen $V = V_P \cup V_F$, einer Menge von Eingabevariablen $V_I \subseteq V_P$ und einer Menge von Ausgabevariablen $V_O \subseteq V_P$. Ein SPL-Testfall $tc = (I, O, pc)$ ist ein Tripel, wobei I ein Eingabevektor, O ein optionaler Ausgabevektor und pc eine Presence Condition über den Feature-Variablen V_F ist.

Im weiteren Verlauf dieses Kapitels werden die Begriffe Testfall und SPL-Testfall synonym verwendet, wobei immer SPL-Testfall gemeint ist.

Beispiel 4.6 (SPL-Testfall)**.** Ein Beispiel für einen SPL-Testfall, der für die SPL-Implementierung der Calc-SPL gültig ist, ist der SPL-Testfall $tc = ([x = 2, y = 3], [r = 5], fm_C \wedge Sum)$. Der SPL-Testfall tc ist für alle Produkte gültig, die das Feature *Sum* implementieren. Insbesondere heißt das, dass der Testfall zwischen Produkten mit und ohne dem Feature *Chk* wiederverwendbar ist.

Für die automatisierte Ableitung eines Testfalls für eine SPL-Implementierung kann die gleiche Technik verwendet werden wie für die Ableitung von Testfällen für einzelne Software-Produkte (vgl. Abschnitt 3.1.4). Dabei müssen die Ergebnisse allerdings so interpretiert werden, dass für Regionen zwischen Pfadbedingung und Presence Condition unterschieden wird. Es werden jedoch die gleichen zwei Schritte zur Ableitung eines Testfalls für ein Testziel *tga* ausgeführt, wobei der zweite Schritt um die Ableitung einer Presence Condition erweitert wird:

1. Das Finden eines Pfades durch den CFA einer IUT, der durch das Testziel *tga* akzeptiert wird und das Testziel somit abdeckt.

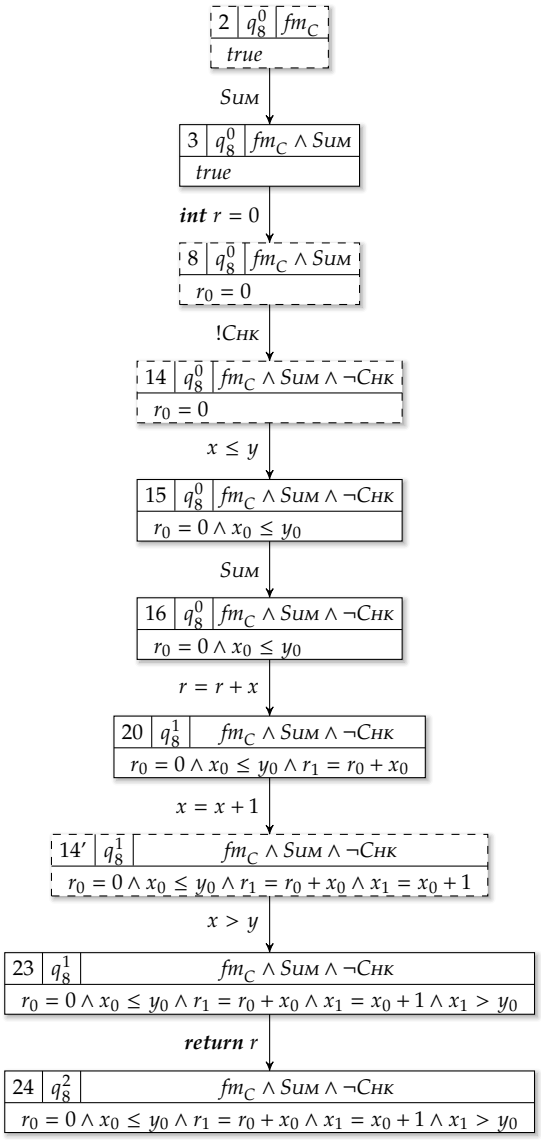


Abbildung 4.5: Ausschnitt des ARG_T der Calc-SPL für das Testziel tga_8

2. Die Ableitung von Eingabewerten und ggf. Ausgabewerten, die der Pfadbedingung des gefundenen CFA-Pfades entsprechen, *sowie einer Presence Condition*.

Für den zweiten Schritt kann wieder ein ARG_T durch einen Model-Checker konstruiert werden, dessen Programmzustände eine Komponente enthalten, die den aktuellen Zustand des gerade betrachteten Testziels *tga* enthalten. Somit wird ein ARG_T wie in Abschnitt 3.1.4 für ein Testziel *tga* konstruiert.

Beispiel 4.7 (Testfallableitung für SPL-Implementierungen)**.** Abbildung 4.5 zeigt den teilweise konstruierten ARG_T für das Testziel tga_8 aus Abbildung 4.2 für die Calc-SPL. Jeder Programmzustand des ARG_T besteht aus vier Komponenten: einer Programm-Location, dem aktuellen TGA-Zustand des TGAs tga_8 sowie einer Presence Condition und einer Pfadbedingung, welche die Regi-

on des Programmzustands bilden. Ein Programmzustand beschreibt, welche Programm-Location mit welchen Variablenbelegungen in welchen Produkten erreicht werden kann und welchen TGA-Zustand der TGA dabei einnimmt.

Der TGA tga_8 hat in Programmzustand 24 seinen Endzustand q_8^2 erreicht und es kann durch das Lösen der Pfadbedingung des Programmzustands 24 ein konkreter Testfall abgeleitet werden, der das Testziel tga_8 erfüllt. Die Presence Condition des abgeleiteten Testfalls ist die Presence Condition von Programmzustand 24. Ein möglicher Testfall ist somit $tc_1 = ([x = 2, y = 3], [r = 5], fm_C \wedge \textsc{Sum} \wedge \neg \textsc{Chk})$. Der Testfall tc_1 deckt das Testziel tga_8 auf allen Produkten ab, welche die Presence Condition $fm_C \wedge \textsc{Sum} \wedge \neg \textsc{Chk}$ erfüllen. In diesem Fall ist das nur das Produkt $\{\textsc{Sum}\}$.

Neben dem Produkt $\{\textsc{Sum}\}$ gibt es noch ein weiteres Produkt der Calc-SPL, welches das Testziel tga_8 enthält, das Produkt $\{\textsc{Sum}, \textsc{Chk}\}$. Für das Produkt $\{\textsc{Sum}, \textsc{Chk}\}$ wurde jedoch kein Testfall abgeleitet, der das Testziel tga_8 auf dem Produkt $\{\textsc{Sum}, \textsc{Chk}\}$ abdeckt.

Wie in Beispiel 4.7 zu sehen ist, kann es vorkommen, dass ein abgeleiteter Testfall ein Testziel nur auf einer Teilmenge aller Produkte abdeckt, die das Testziel enthalten. In vielen Fällen ist es jedoch wünschenswert, dass zum Testen aller Produkte ein Testziel entsprechend auf allen Produkten einer SPL durch eine Test-Suite abgedeckt wird, d. h. *vollständige SPL-Abdeckung* für ein Testziel erreicht wird.

Definition 4.8 (Vollständige SPL-Abdeckung eines Testziels (basierend auf [COLS11]))**.** Ein Testziel *tga* wird *vollständig* auf einer SPL-Implementierung durch eine Test-Suite *TS* abgedeckt, wenn die Test-Suite *TS* das Testziel *tga* auf jedem Produkt der SPL durch mindestens einen Testfall abdeckt, welches das Testziel *tga* enthält.

Somit erfordert die Testfallableitung für die vollständige Abdeckung eines Testziels auf einer SPL-Implementierung, dass weitere Testfälle für ein Testziel abgeleitet und zu einer Test-Suite hinzugefügt werden, bis das Testziel auf allen Produkten durch die Test-Suite abgedeckt wird, die das Testziel enthalten.

Um eine vollständige Abdeckung eines Testziels *tga* auf einer SPL zu erreichen, wird zuerst der ARG_T schritteweise für das Testziel *tga* konstruiert, bis daraus ein Testfall $tc_1 = (I_1, O_1, pc_1)$ abgeleitet werden kann, der das Testziel *tga* auf den Produkten pc_1 abdeckt (vgl. Beispiel 4.7). Nachdem der Testfall tc_1 abgeleitet wurde, wird der ARG_T schrittweise weiter konstruiert bis der nächste Testfall $tc_2 = (I_2, O_2, pc_2)$ abgeleitet werden kann, der das Testziel *tga* auf weiteren Produkten pc_2 abdeckt. Dies wird solange wiederholt, bis das Testziel *tga* auf allen Produkten abgedeckt wurde, auf denen es erreichbar ist. Um sicherzustellen, dass nur Testfälle generiert werden, die das Testziel *tga* auf Produkten abdecken, auf denen es noch nicht abgedeckt wurde, wird der Programmzustandsraum während der ARG_T-Konstruktion auf die Programmzustände der Produkte eingeschränkt, auf denen *tga* noch nicht abgedeckt ist [BLB+15]. Dabei wird der gleiche Ansatz verwendet, der dafür sorgt, dass der konstruierte ARG_T nur Programmzustände enthält, die in mindestens einem Produkt einer SPL enthalten sind (vgl. Beispiel 4.3). Immer dann, wenn ein neuer Testfall $tc_i = (I_i, O_i, pc_i)$ abgeleitet wurde, werden die

Presence Conditions von neu berechneten Programmzuständen während der weiteren Konstruktion des ARG_T um das Prädikat $\neg pc_i$ erweitert. Dadurch wird die Berechnung von Programmzuständen verhindert, die nur in Produkten enthalten sind, die $\neg pc_i$ erfüllen, wodurch die Konstruktion des ARG_T auf Produkte eingeschränkt wird, für die das Testziel *tga* noch nicht abgedeckt wurde [BLB+15].

Beispiel 4.9 (Vollständige Abdeckung eines Testziels auf einer SPL)**.** In Beispiel 4.7 wurde ein Testfall $tc_1 = ([x = 2, y = 3], [r = 5], fm_C \wedge Sum \wedge \neg Chk)$ für die Calc-SPL abgeleitet, der das Testziel tga_8 aus Abbildung 4.2 auf Produkten der Calc-SPL abdeckt, deren Produkt-Konfiguration der Presence Condition $fm_C \wedge Sum \wedge \neg Chk$ entsprechen. Zur Ableitung des Testfalls tc_1 wurde der ARG_T aus Abbildung 4.5 konstruiert, der Programmzustände enthält, zu denen es noch weitere nachfolgende Programmzustände gibt, die noch nicht berechnet wurden (Programmzustände mit gestricheltem Rahmen). Zur Ableitung einer Test-Suite, die das Testziel tga_8 vollständig auf der Calc-SPL abdeckt, müssen weitere Testfälle abgeleitet werden, die das Testziel tga_8 auf weiteren Produkten abdecken, welche das Testziel tga_8 enthalten.

Zur Ableitung weiterer Testfälle wird die Konstruktion des ARG_T aus Abbildung 4.5 fortgesetzt, wobei die Konstruktion auf die Teile des ARG_T beschränkt wird, welche die Presence Condition $pc_1 = fm_C \wedge Sum \wedge \neg Chk$ von tc_1 nicht erfüllen. Das Ergebnis der weiteren ARG_T-Konstruktion ist in Abbildung 4.6 dargestellt. Die Konstruktion des ARG_T aus Abbildung 4.5 wurde dabei in Programmzustand 8 mit dem neuen Nachfolgeprogrammzustand 9 fortgesetzt. Die Presence Condition von Programmzustand 9 wurde dabei mit der Negation der Presence Condition $pc_1 = fm_C \wedge Sum \wedge \neg Chk$ des Testfalls tc_1 konjugiert, sodass kein Programmzustand mehr berechnet wird, der nicht in einem Produkt enthalten ist, auf dem das Testziel tga_8 noch nicht abgedeckt wurde. Beispielsweise ist die Presence Condition von Programmzustand 23″ (der durchgestrichen dargestellt ist) konjugiert mit $\neg pc_1$ nicht erfüllbar. Deswegen wird Programmzustand 23″ während der fortgesetzten ARG_T-Konstruktion nicht betrachtet.

Aus dem abstrakten Testfall $2-3-8-9-14''-15'-16'-20'-14'''-23'-24'$ kann ein konkreter Testfall abgeleitet werden, der das Testziel tga_8 erfüllt, da tga_8 in Programmzustand 24′ einen Endzustand erreicht hat. Ein möglicher konkreter Testfall ist $tc_2 = ([x = 2, y = 3], [r = 5], fm_C \wedge Sum \wedge Chk \wedge \neg pc_1)$. Der Testfall tc_2 deckt das Testziel tga_8 auf dem Produkt $\{Sum, Chk\}$ ab. Dabei unterscheiden sich die Testfälle tc_1 und tc_2 lediglich in ihren Presence Conditions. Da die Testfälle jedoch aus verschiedenen ARG_T-Pfaden abgeleitet wurden, können die Testfälle nicht zusammen als ein Testfall abgeleitet werden.

Die Test-Suite $TS = \{tc_1, tc_2\}$ deckt das Testziel tga_8 vollständig auf der Calc-SPL ab, d. h. auf allen Produkten in denen es vorkommt.

Im Allgemeinen ist nicht syntaktisch bestimmbar, in welchen Produkten welche Testziele enthalten sind. Somit gibt es keine a-priori Abbruchbedingung für das oben beschriebene Verfahren, dass entscheidet, wann ein Testziel vollständig auf einer SPL-Implementierung abgedeckt ist. Eine Lösung für das Problem ist, dass das Verfahren solange ausgeführt wird, bis der Model-Checker ausgibt, dass das

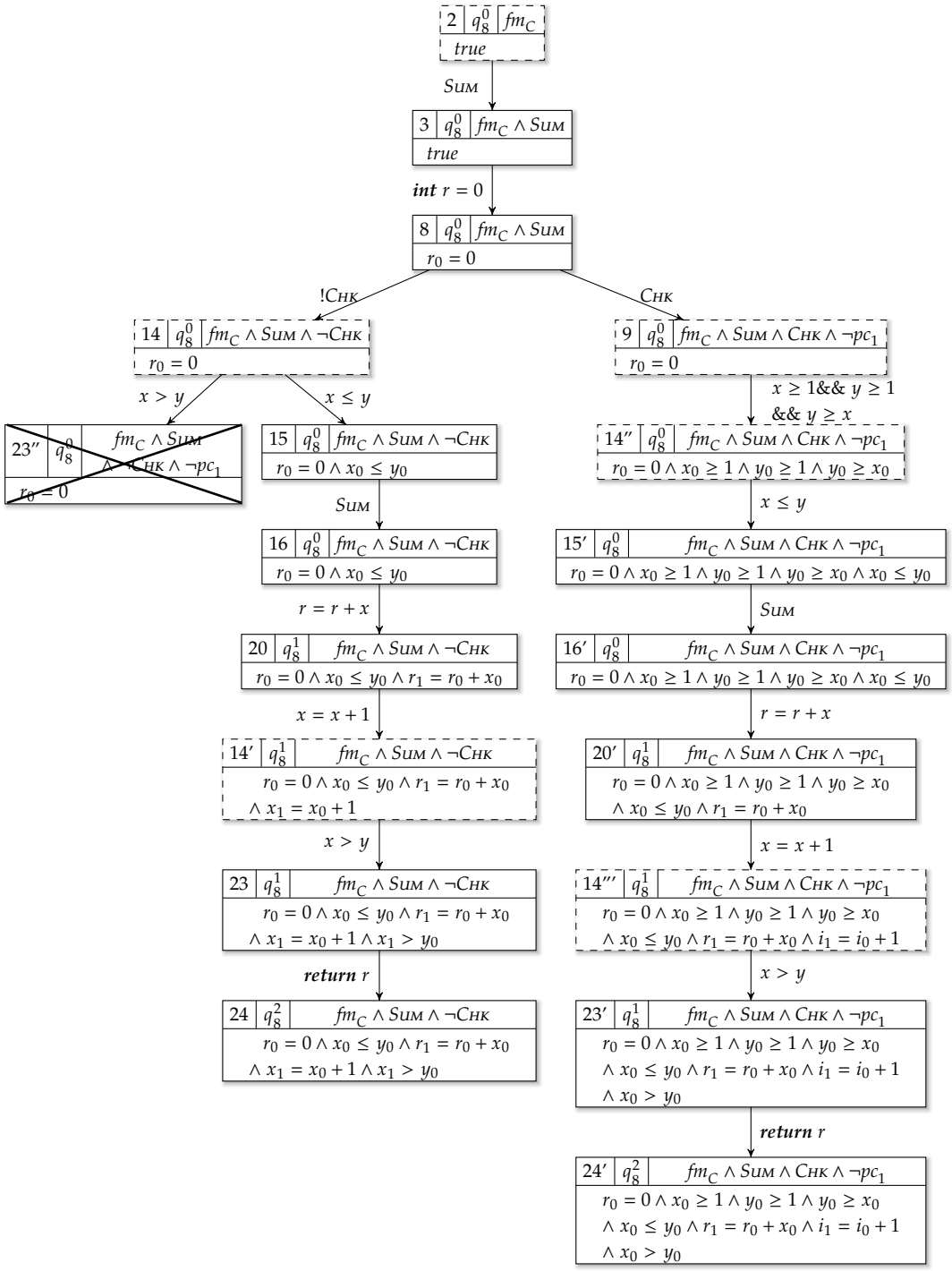


Abbildung 4.6: Ausschnitt des ARG_T der Calc-SPL für tga_8 mit $pc_1 = fm_C \wedge Sum \wedge \neg Chk$

Testziel im restlichen Programmzustandsraum unerfüllbar ist, d. h. auf keinem weiteren Produkt abdeckbar oder unbekannt ist.

Beispiel 4.10 (Beenden der ARG_T-Konstruktion)**.** Nachdem in Beispiel 4.9 Testfall $tc_2 = ([x = 2, y = 3], [r = 5], fm_C \wedge Sum \wedge Chk \wedge \neg pc_1)$ abgeleitet wurde, wird die ARG_T-Konstruktion weiter fortgesetzt. Dabei wird der zu konstruierende

> Zustandsraum weiter durch das Prädikat $pc_3 = \neg pc_1 \wedge \neg pc_2$ eingeschränkt, d. h. auf die Produkte, auf denen das Testziel tga_8 noch nicht abgedeckt ist. Da das Testziel im restlichen Zustandsraum nicht mehr erreichbar ist, wird zurückgegeben, dass das Testziel tga_8 unerfüllbar ist. Somit ist das Testziel tga_8 auf keinem Produkt erreichbar, dessen Produktkonfiguration der Presence Condition pc_3 entspricht. Im Falle der Calc-SPL sind das die Produkte, die das Feature *Fac* implementieren und somit die Anweisung $r = r + x$ nicht enthalten.

Das Verfahren zur Testfallableitung für einzelne Software-Produkte aus Abschnitt 3.1.4 ist ein Spezialfall des in diesem Abschnitt beschriebenen Verfahrens zur Testfallableitung für SPLs. Wird das Verfahren zur Testfallableitung für SPLs auf ein einzelnes Software-Produkt, d. h. auf einer Produktimplementierung ohne Features, zur Abdeckung eines Testziels *tga* angewendet, ist die Presence Condition in jedem Programmzustand eines konstruierten ARG_T *true*. Dadurch hat ein abgeleiteter Testfall, der das Testziel *tga* abdeckt, immer die Presence Condition *true*, da es für eine Produktimplementierung kein Feature-Modell gibt. Danach wird die weitere ARG_T-Konstruktion für das Testziel *tga* gestoppt, da die Presence Condition von jedem neuen Programmzustand während der weiteren ARG_T-Konstruktion mit $\neg true$ konjugiert wird und der Programmzustand somit nicht berücksichtigt wird, da seine Presence Condition nicht erfüllbar ist.

Im folgenden Abschnitt wird die Konstruktion von ARG_T für eine SPL-Implementierung beschrieben sowie die Behandlung von Feature-Variablen während der ARG_T-Konstruktion.

4.1.3 *Variabilitätsgewahre ARG_T-Konstruktion*

In Abschnitt 3.1.5 wurde die Konstruktion von ARG_T für einzelne Software-Produkte mit Hilfe der CPA $\text{CPA}_\mathbb{R}$ beschrieben. Die CPA $\text{CPA}_\mathbb{R}$ besteht dabei aus drei Komponenten: der Location-CPA $\text{CPA}_\mathbb{L}$ zur Traversierung eines CFA, der Prädikaten-CPA $\text{CPA}_\mathbb{P}$ zur Behandlung der Menge der Programmvariablen V als Regionen r und der Automaten-CPA $\text{CPA}_\mathbb{A}$ zur TGA-gerichteten Programmanalyse. Die Konstruktion eines variabilitätsgewahren ARG_T erweitert die Konstruktion eines ARG_T für einzelne Software-Produkte um die Behandlung von Feature-Variablen durch die Aufspaltung der Region $r = (\phi \wedge pc)$ eines Programmzustandes in zwei Teile: einer Pfadbedingung ϕ über den Programmvariablen $V_P \subseteq V$ und einer Presence Condition pc über den Feature-Variablen $V_F \subseteq V$. Zur Behandlung der unterschiedlichen Teile der Region kann die CPA $\text{CPA}_\mathbb{R}$ im Folgenden unverändert weiter verwendet werden. Dabei Behandelt die Prädikaten-CPA nicht mehr nur die Programmvariablen V_P, sondern auch die Feature-Variablen V_F. Da wir fordern, dass Bedingungen in einer IUT entweder über den Programmvariablen V_P formuliert werden ($\mathcal{L}_{p\text{-}assume}(V_P)$) oder über den Feature-Variablen V_F ($\mathcal{L}_{f\text{-}assume}(V_F)$) können die Pfadbedingung ϕ und die Presence Condition pc einer Region $r = (\phi \wedge pc)$ klar voneinander getrennt werden. Weiterhin kann Algorithmus 1, der Anhand einer CPA einen ARG aufbaut, zur Konstruktion eines variabilitätsgewahren ARG_T unverändert verwendet werden.

Die Prädikaten-CPA $\text{CPA}_\mathbb{P}$ behandelt bei der variabilitätsgewahren ARG_T-Konstruktion neben den Programmvariablen auch Feature-Variablen. Daraus resultie-

ren zwei neue Aufgaben, die durch die Prädikaten-CPA $\text{CPA}_{\mathbb{P}}$ erfüllt werden müssen:

1. Die Speicherung und Berechnung einer Presence Condition, die angibt in welchen Produkten einer SPL ein Programmzustand enthalten ist.
2. Das Abbrechen beim Erreichen von Programmzuständen, die in keinem Produkt einer SPL enthalten sind.

Die erste Aufgabe wird dadurch erfüllt, dass die Prädikaten-CPA $\text{CPA}_{\mathbb{P}}$ sämtliche Variablen eines Programms behandelt, inklusive der Feature-Variablen, und somit Presence Conditions als Teil von Regionen speichert. Die zweite Aufgabe wird durch den Strongest-Postcondition-Operator sp_V erfüllt (vgl. Definition 3.7). Der Strongest-Postcondition-Operator fordert, dass für einen Programmzustandsübergang $r \xrightarrow{e} r'$ mit $e = (\ell, a, \ell')$ die Bedingung $r \wedge a \models r'$ gelten muss (vgl. Definition 3.7). Damit diese Bedingung gilt, muss $r' = (\phi' \wedge pc')$ und somit auch die Presence Condition pc' von r' erfüllbar sein. Da die Presence Condition pc' immer das Feature-Modell fm einer SPL enthält, muss der Programmzustand mit der Region r' in einem Produkt der SPL enthalten sein.

Beispiel 4.11 (Variabilitätsgewahre ARG_{T}-Konstruktion)**.** Abbildung 4.6 zeigt einen ARG_{T}, der durch die komponierte CPA $\text{CPA}_{\mathbb{R}}$, bestehend aus den CPAs $\text{CPA}_{\mathbb{L}}$, $\text{CPA}_{\mathbb{P}}$ und $\text{CPA}_{\mathbb{A}}$, für das Testziel tga_8 für die Calc-SPL konstruiert wurde. Die CPAs $\text{CPA}_{\mathbb{L}}$ und $\text{CPA}_{\mathbb{T}}$ funktionieren dabei wie in Kapitel 3. Die Regionen $r = (\phi \wedge pc)$ der Programmzustände sind jeweils in eine Pfadbedingung ϕ und Presence Condition pc unterteilt und werden von der Prädikaten-CPA $\text{CPA}_{\mathbb{P}}$ berechnet. Die Presence Condition kann dabei jederzeit durch einen äußeren Algorithmus eingeschränkt werden. Beispielsweise wird die Presence Condition im initialen Programmzustand $(2, q_8^0, true \wedge fm_C)$ des ARG_{T} auf die Menge aller validen Produkte der Calc-SPL eingeschränkt, die durch das Feature-Model fm_C beschrieben werden, sodass nur Programmzustände konstruiert werden, die tatsächlich in mindestens einem Produkt der Calc-SPL enthalten sind.

Die CPA $\text{CPA}_{\mathbb{P}}$ verfeinert die Pfadbedingung bei einem Übergang $\Psi \rightsquigarrow_{\text{S}} \Psi'$ für eine CFA-Kante $e = (\ell, a, \ell)$ zwischen den zwei abstrakten Programmzuständen Ψ und Ψ', wenn die Programmoperation $a \in \mathcal{L}_{f\text{-}assume}(V_P)$ auf der Menge der Programmvariablen $V_P \subseteq V$ arbeitet, wie es auch in Kapitel 3 für Software-Produkte der Fall ist. Beispielsweise wird beim Übergang

$$(3, q_8^0, true \wedge fm_C \wedge \textit{Sum}) \xrightarrow{r = 0} (8, q_8^0, r_0 = 0 \wedge fm_C \wedge \textit{Sum})$$

von Programmzustand $(3, q_8^0, true \wedge fm_C \wedge \textit{Sum})$ nach Programmzustand $(8, q_8^0, r_0 = 0 \wedge fm_C \wedge \textit{Sum})$ die Pfadbedingung angepasst. Weiterhin verfeinert die CPA $\text{CPA}_{\mathbb{P}}$ die Presence Condition bei einem Übergang $\Psi \rightsquigarrow_{\text{S}} \Psi'$ für eine CFA-Kante $e = (\ell, a, \ell)$, wenn die Programmoperation $a \in \mathcal{L}_{f\text{-}assume}(V_F)$ auf der Menge der Feature-Variablen $V_F \subseteq V$ arbeitet. Beispielsweise wird beim Übergang

$$(2, q_8^0, true \wedge fm_C) \xrightarrow{\textit{Sum}} (3, q_8^0, true \wedge fm_C \wedge \textit{Sum})$$

von Programmzustand $(2, q_8^0, true, fm_C)$ nach Programmzustand $(3, q_8^0, true, fm_C \wedge Sum)$ die Presence Condition auf Produkte eingeschränkt, die das Feature *Sum* implementieren, da dies durch die Bedingung *Sum* des Programmzustandsübergangs gefordert wird.

Die Transferrelation $\rightsquigarrow_{\mathbb{P}}$ der Prädikaten-CPA berechnet gemäß der Strongest-Postcondition nur Übergänge zu Programmzuständen, die in mindestens einem Produkt enthalten sind. Beispielsweise ist der Programmzustand $(23'', q_8^0, r_0 = 0 \wedge fm_C \wedge Sum \wedge \neg Sum \wedge \neg(fm_C \wedge Sum \wedge \neg CHK))$ kein valider Nachfolgeprogrammzustand von Programmzustand 14, da die Presence Condition von Programmzustand 23'' nicht erfüllbar ist, d. h.

$$fm_C \wedge Sum \wedge \neg Sum \wedge \neg(fm_C \wedge Sum \wedge \neg CHK) \Leftrightarrow false.$$

Basierend auf der ARG_T-Konstruktion aus diesem Abschnitt und dem Verfahren für die SPL-Testfallgenerierung aus Abschnitt 4.1.2 werden wir im Folgenden Abschnitt einen Algorithmus zur Generierung vollständiger Test-Suiten für SPLs vorstellen, d. h. von Test-Suiten, die für jedes Testziel vollständige SPL-Abdeckung garantieren.

4.1.4 *Vollständige Test-Suite-Generierung für Software-Produktlinien*

Zur Generierung einer vollständigen SPL-Test-Suite für eine Menge von Testzielen für eine SPL-Implementierung iteriert ein Testfallgenerator zur Testfallableitung über die Menge von Testzielen, ähnlich wie es in Abschnitt 3.1.6 für Software-Produkte beschrieben wurde. Im Gegensatz zum Verfahren aus Abschnitt 3.1.6 wird jedoch nicht pro Testziel maximal ein Testfall abgeleitet, der das Testziel abdeckt, sondern eine Menge von Testfällen, welche die vollständige SPL-Abdeckung für ein Testziel erfüllen, d. h. ein Testziel auf jedem Software-Produkt abdecken, welches das Testziel enthält.

Definition 4.12 (Vollständige SPL-Test-Suite (basierend auf [COLS11]))**.** Eine SPL-Test-Suite *TS* für eine Menge von Testzielen *G* ist *vollständig*, wenn die Test-Suite *TS* für jedes Testziel $g \in G$ vollständige SPL-Abdeckung erfüllt.

Die Ableitung einer vollständige SPL-Test-Suite basiert auf den Verfahren aus Abschnitt 4.1.2 und Abschnitt 4.1.3.

Beispiel 4.13 (Generierung einer vollständigen SPL-Test-Suite)**.** Es soll eine vollständige SPL-Test-Suite für die Testzielmenge $\{tga_9, tga_{10}, tga_{11}\}$ aus Abbildung 4.2 für die Calc-SPL generiert werden. Zur Generierung der Test-Suite iteriert ein Testfallgenerator über der Testzielmenge und erstellt für jedes Testziel eine Menge von SPL-Testfällen, um eine vollständige SPL-Abdeckung zu erzielen, wie es in Beispiel 4.9 für ein einzelnes Testziel beschrieben wurde.

Begonnen wird die Testgenerierung beispielsweise mit dem Testziel tga_9. Abbildung 4.7 zeigt einen Ausschnitt des ARG_T, der zur Ableitung von SPL-Testfällen für das Testziel tga_9 konstruiert wurde. Der ARG_T wurde in zwei Schritten konstruiert. Im ersten Schritt wurde der Teil des ARG_T konstruiert, der mit

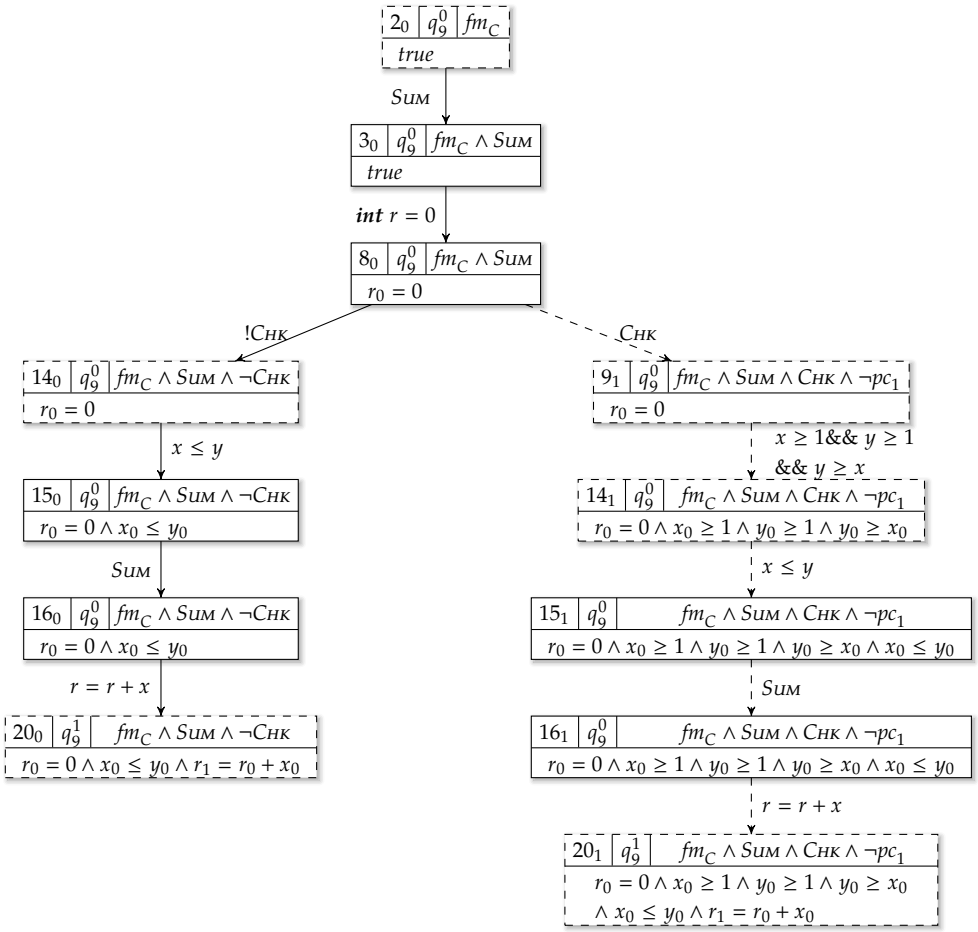


Abbildung 4.7: Ausschnitt des ARG_T der Calc-SPL für tga_9 mit $pc_1 = fm_C \wedge Sum \wedge \neg Chk$

durchgezogenen Transitionen verbunden ist. In Programmzustand 20_0 ist der TGA tga_9 in einem Endzustand und es kann der Testfall $tc_1 = ([x = 1, y = 2], pc_1)$ mit $pc_1 = fm_C \wedge Sum \wedge \neg Chk$ abgeleitet werden. Für den Testfall tc_1 kann kein Ausgabevektor abgeleitet werden, da der Programmzustand 20_0 kein Programmende repräsentiert. Der Testfall tc_1 deckt das Testziel tga_9 auf allen Produkten ab, die der Presence Condition pc_1 entsprechen. In einem zweiten Schritt wird der Teil vom ARG_T konstruiert, der mit gestrichelten Transitionen verbunden ist, wobei der zu konstruierende Programmzustandsraum auf Programmzustände eingeschränkt wird, die nicht durch pc_1 charakterisiert werden (vgl. Beispiel 4.9). In Programmzustand 20_1 hat der TGA tga_9 wieder einen Endzustand erreicht und es kann ein weiterer Testfall $tc_2 = ([x = 1, y = 2], pc_2)$ mit $pc_2 = fm_C \wedge Sum \wedge Chk \wedge \neg pc_1$ abgeleitet werden, der das Testziel tga_9 auf weiteren Produkten abdeckt. Wie in Beispiel 4.10 beschrieben wurde, wird der ARG_T teilweise weiter konstruiert, um zu überprüfen, ob das Testziel tga_9 in weiteren Produkten abdeckbar ist. Die ARG_T-Konstruktion wird dabei durch das Feature-Modell und die Presence Condition $\neg pc_1 \wedge \neg pc_2$ eingeschränkt, für die das Testziel tga_9 bereits abgedeckt ist. Die Testfälle tc_1 und tc_2 decken das Testziel tga_9 auf allen Produkten der Calc-SPL ab, die das Testziel tga_9 enthalten.

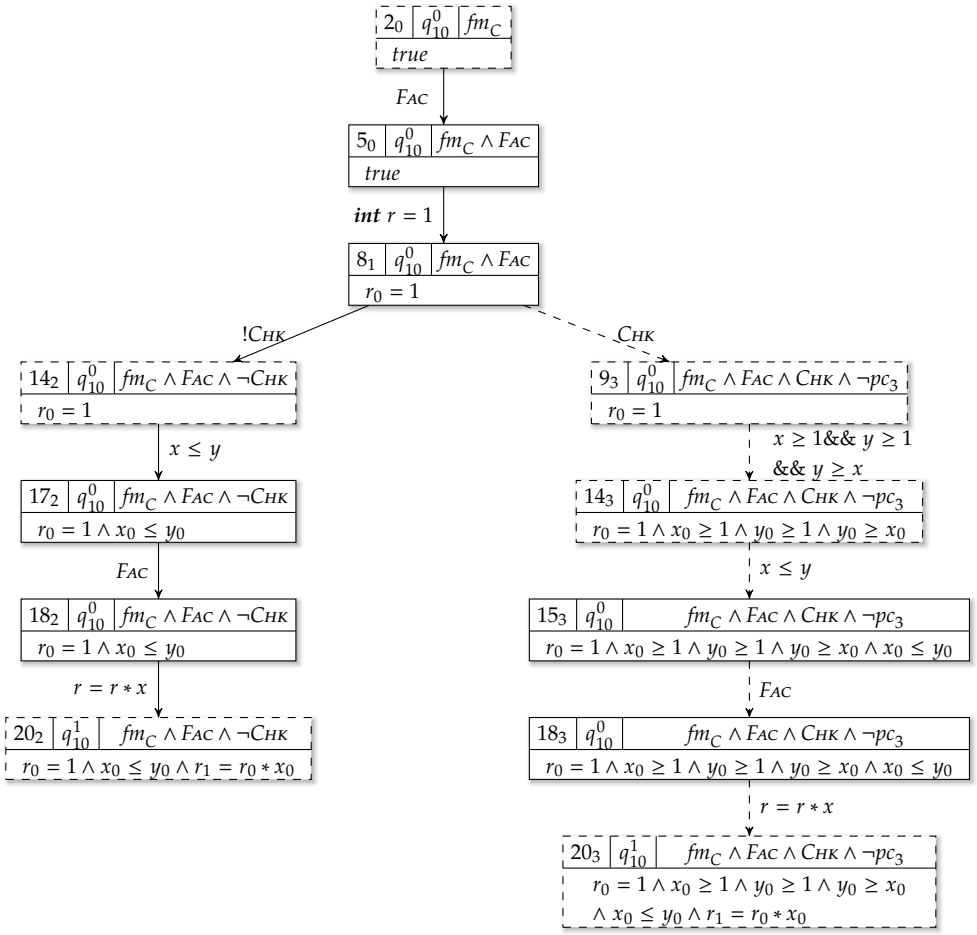


Abbildung 4.8: Ausschnitt des ARG_T der Calc-SPL für tga_{10} mit $pc_3 = fm_C \wedge FAC \wedge \neg CHK$

Das gleiche Verfahren wird für die Testziele tga_{10} und tga_{11} durchgeführt. Für das Testziel tga_{10} wird der ARG_T aus Abbildung 4.8 konstruiert und es werden die Testfälle $tc_3 = ([x = 2, y = 4], pc_3)$ mit $pc_3 = fm_C \wedge FAC \wedge \neg CHK$ (Programmzustand 20_2) und $tc_4 = ([x = 2, y = 4], pc_4)$ mit $pc_4 = fm_C \wedge FAC \wedge CHK \wedge \neg pc_3$ (Programmzustand 20_3) generiert. Für das Testziel tga_{11} wird der ARG_T aus Abbildung 4.9 in vier Schritten konstruiert und es werden die Testfälle $tc_5 = ([x = 1, y = 2], pc_1)$ (Programmzustand 14_4), $tc_6 = ([x = 1, y = 2], pc_2)$ (Programmzustand 14_5), $tc_7 = ([x = 2, y = 4], pc'_3)$ mit $pc'_3 = fm_C \wedge FAC \wedge CHK \wedge \neg pc_2$ (Programmzustand 14_6) und $tc_8 = ([x = 2, y = 4], pc_4)$ mit $tc_4 = fm_C \wedge FAC \wedge CHK \wedge \neg pc'_3$ (Programmzustand 14_7) generiert.

Die Test-Suite $TS = \{tc_1, tc_2, tc_3, tc_4, tc_5, tc_6, tc_7, tc_8\}$ ist eine vollständige SPL-Test-Suite zur Abdeckung der Testziele tga_9, tga_{10} und tga_{11} auf allen Produkten, auf denen die Testziele erreichbar sind.

Algorithmus 6 beschreibt das Vorgehen zur Generierung einer vollständigen SPL-Test-Suite für eine Menge von Testzielen aus Beispiel 4.13. Die Eingaben für den Algorithmus sind der Kontrollflussautomat CFA der IUT, ein Feature-Modell *fm* als aussagenlogischer Term und eine Menge von Testzielen *TG*, die abgedeckt werden sollen. Die Ausgabe des Algorithmus ist eine abstrakte SPL-Test-Suite *TS*. Die Test-Suite *TS* wird mit der leeren Menge initialisiert (Zeile 1) und die Menge der noch

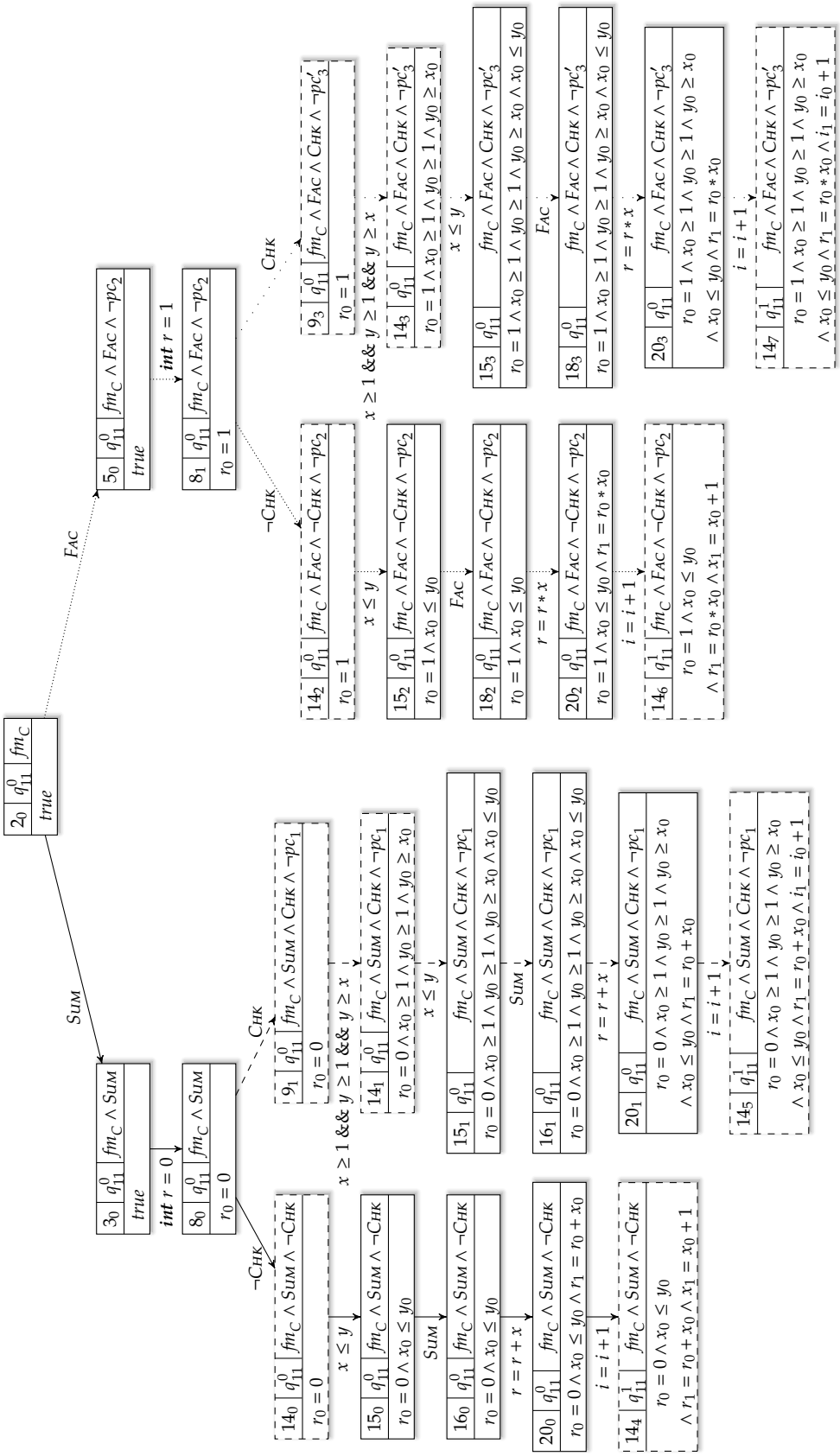


Abbildung 4.9: Ausschnitt des ARG_T der Calc-SPL für tga_{11}, $pc_1 = fm_C \wedge Sum \wedge \neg Chk$, $pc_2 = fm_C \wedge Sum \wedge Chk \wedge \neg pc_1$, $pc'_3 = fm_C \wedge Fac \wedge \neg Chk \wedge \neg pc_2$

Algorithmus 6 Familienbasierte Test-Suite-Generierung [BLB+15]

Input: Kontrollflussautomat $CFA = (L, \ell_0, \ell_t, E)$,
Feature-Modell fm,
Menge von Testzielautomaten $TG = \{tga_1, \ldots, tga_n\}$
Output: Abstrakte SPL-Test-Suite TS

1: $TS := \{\}$
2: $UC := TG$
3: **for each** $tga \in TG$ **do**
4: $\quad CS[tga] := fm$
5: **while** $UC \neq \emptyset$ **do**
6: $\quad tga := (Q, \Sigma, \Delta, q_0, F) \in UC$
7: $\quad pc_0 := CS[tga]$
8: $\quad$ **while** $\exists\ \omega := \psi_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{k-1}} \psi_k$ mit $\psi_i = (\ell_i, q_i, r_i)$, $r_i = (\phi_i, pc_i)$, $i = 0, \ldots, k$, $q_k \in F$ und $sat(pc_k \wedge CS[tga])$ **do**
9: $\quad\quad TS := TS \cup (tga, \omega, pc_k)$
10: $\quad\quad CS[tga] := CS[tga] \wedge \neg pc_k$
11: $\quad\quad pc_0 := CS[tga]$
12: $\quad UC := UC \setminus tga$

nicht abgearbeiteten Testziele UC mit der Menge aller Testziele TG (Zeile 2). Der Vektor CS (Cover Set) speichert für jedes Testziel tga eine Presence Condition über Feature-Variablen, die beschreibt für welche Produkte tga noch nicht abgedeckt ist. CS wird für jedes Testziel tga mit dem Feature-Modell fm initialisiert (Zeile 3–4), d. h. dass das Testziel tga noch auf keinem Produkt abgedeckt ist. Solange die Menge der noch nicht bearbeiteten Testziele UC nicht leer ist, iteriert der Algorithmus über diese Menge und generiert SPL-Testfälle zur Abdeckung der Testziele in UC (Zeile 5). Zur Generierung von Testfällen wählt der Algorithmus das nächste zu betrachtende Testziel tga aus der Menge UC aus (Zeile 6) und initialisiert die Presence Condition pc_0 des initialen Programmzustands der IUT mit der Menge der Produkte für die das Testziel tga noch nicht abgedeckt ist (Zeile 7). Zu Beginn ist dies im Allgemeinen das Feature-Modell fm, d. h. bei der Konstruktion des ARG_T zur Testfallableitung werden alle Produkte einer SPL betrachtet. Es werden solange Testfälle für ein Testziel tga abgeleitet bis es keinen neuen Testfall mehr gibt, der das Testziel tga auf einem Produkt abdeckt, auf dem es noch nicht abgedeckt ist (Zeile 8). Gibt es noch einen neuen abstrakten Testfall ω der das Testziel tga auf einer Menge von Produkten abdeckt, auf der es noch nicht abgedeckt ist, wird der Testfall ω zusammen mit dem Testziel tga und der Presence Condition pc_k, für die das Testziel durch den Testfall abgedeckt wird, als Tripel zur Test-Suite TS hinzugefügt (Zeile 9). Die Prüfung, ob ein Testfall ein Testziel auf einer neuen Menge von Produkten abdeckt, geschieht durch das Prüfen der Erfüllbarkeit der Formel $pc_k \wedge CS[tga]$, wobei die Erfüllbarkeit einer Formel φ durch die Funktion $sat(\varphi)$ geprüft werden kann. Die Funktion $sat(\varphi)$ gibt *true* zurück, wenn φ erfüllbar ist und sonst *false*. Weiterhin wird gespeichert, dass das Testziel tga für eine neue Menge von Produkten abgedeckt wurde (Zeile 10) und die weitere Analyse wird auf Produkte eingeschränkt, für die das Testziel tga noch nicht abgedeckt ist (Zeile 11).

Wurde das Testziel für alle Produkte abgedeckt, die das Testziel enthalten, wird das Testziel aus der Menge *UC* entfernt (Zeile 12).

Algorithmus 6 ist eine Verallgemeinerung von Algorithmus 2, d. h. für den Fall, dass es in einer IUT keine Features gibt, verhält sich der Algorithmus 6 genau wie Algorithmus 2. Der Vektor *CS* wird für eine IUT ohne Features mit *true* initialisiert, d. h. es gibt kein Feature-Modell und somit im gesamten Programmzustandsraum nach einem Testfall gesucht wird. Sobald ein Testfall für ein Testziel *tga* abgeleitet wurde, wird der Programmzustandsraum auf die Negation der Presence Condition des Testfalls (d. h. auf $\neg true$) eingeschränkt und somit wird die Testgenerierung für dieses Testziel beendet.

Algorithmus 6 generiert Test-Suiten mit Wiederverwendung von Testfällen zwischen Produkten, d. h. ein generierter Testfall kann ein Testziel auf mehreren Produkten abdecken, sodass nicht jedes Produkt einzeln analysiert werden muss. Zur Verbesserung der SPL-Test-Suite-Generierung kann die Wiederverwendung von Testfällen zwischen Produkten mit der Wiederverwendung von Testfällen zwischen Testzielen aus Abschnitt 3.1.7 kombiniert werden. So kann beispielsweise die Generierung der gleichen Testfälle für unterschiedliche Testziele in Beispiel 4.13 vermieden werden. Die Kombination der Wiederverwendung von Testfällen zwischen Produkten und Testzielen wird im folgenden Abschnitt beschrieben.

4.1.5 *Test-Suite-Generierung für Software-Produktlinien mit Wiederverwendung von Testfällen zwischen Produkten und Testzielen*

Wie in Abschnitt 3.1.7 für Software-Produkte beschrieben wurde, können generierte Testfälle zwischen Testzielen wiederverwendet werden, um die Anzahl von generierten Testfällen und die dafür benötigte Anzahl von ARG_T-Konstruktionen zu verringern. Diese Technik kann auch auf die Test-Suite-Generierung für SPLs angewendet werden [BLB+15].

Beispiel 4.14 (Testgenerierung für SPLs mit Wiederverwendung von Testfällen)**.** Wie in Beispiel 4.13 soll eine vollständige SPL-Test-Suite für die Testzielmenge $\{tga_9, tga_{10}, tga_{11}\}$ aus Abbildung 4.2 für die Calc-SPL generiert werden. Im Gegensatz zu Beispiel 4.13 sollen bei der Test-Suite-Generierung Testfälle zwischen Produkten *und* Testzielen wiederverwendet werden.

Begonnen wird mit der Generierung von Testfällen zur Abdeckung von Testziel tga_{11}. Hierfür wird der ARG_T aus Abbildung 4.9 konstruiert. Zuerst wird der Teil des ARG_T konstruiert, der mit durchgezogen dargestellten Transitionen verbunden ist. In Programmzustand 14_4 ist der TGA tga_{11} in einem Endzustand und es kann ein Testfall zur Abdeckung des Testziels tga_{11} abgeleitet werden. Der abgeleitete Testfall ist $tc_1 = ([x = 1, y = 2], pc_1)$ mit $pc_1 = fm_C \wedge \textsc{Sum} \wedge \neg \textsc{Chk}$. Dieser Testfall kann zur Abdeckung des Testziels tga_9 auf den Produkten pc_1 wiederverwendet werden. Wie in Beispiel 4.13 wird der ARG_T weiter konstruiert und es werden Testfälle zur Abdeckung des Testziels tga_{11} abgeleitet, die zur Abdeckung der Testziele tga_9 und tga_{10} wiederverwendet werden können.

Im nächsten Schritt wird der Testfall $tc_2 = ([x = 1, y = 2], pc_2)$ mit $pc_2 = fm_C \wedge \text{Sum} \wedge \text{Chk} \wedge \neg pc_1$ aus Programmzustand 14_5 abgeleitet, der ebenfalls zur Abdeckung des Testziels tga_9 wiederverwendet werden kann. Weiterhin werden die Testfälle $tc_3 = ([x = 2, y = 4], pc'_3)$ mit $pc'_3 = fm_C \wedge \text{Fac} \wedge \neg\text{Chk} \wedge \neg pc_2$ und $tc_4 = ([x = 2, y = 4], pc_4)$ mit $pc_4 = fm_C \wedge \text{Fac} \wedge \text{Chk} \wedge \neg pc'_3$ aus den Programmzuständen 14_6 und 14_7 zur Abdeckung des Testziels tga_{11} abgeleitet. Beide Testfälle können wiederverwendet werden, um das Testziel tga_{10} auf den Produkten pc'_3 und pc_4 abzudecken. Die generierte Test-Suite $TS = \{tc_1, tc_2, tc_3, tc_4\}$ erfüllt die vollständige SPL-Abdeckung für das Testziel tga_{11}.

Für die Testziele tga_9 und tga_{10} muss weiterhin geprüft werden, ob die Testziele auf weiteren Produkten erfüllbar sind, auf denen die Testziele durch die Test-Suite TS noch nicht abgedeckt wurden (vgl. Beispiel 4.10). Da dies nicht der Fall ist, erfüllt die Test-Suite TS ebenfalls vollständige SPL-Abdeckung für die Testziele tga_9 und tga_{10}.

Im Gegensatz zu Beispiel 4.13 wurden bei der SPL-Test-Suite-Generierung mit Wiederverwendung von Testfällen zwischen Testzielen nur vier statt acht Testfälle generiert. Außerdem wurden wesentlich weniger redundante ARG_T-Informationen berechnet. Die Reihenfolge in der die Testziele bei der SPL-Test-Suite-Generierung mit Wiederverwendung von Testfällen zwischen Produkten und Testzielen betrachtet werden, hat wie in Abschnitt 3.1.7 einen Einfluss darauf wieviele Testfälle wiederverwendet werden können. Allerdings ist eine Optimierung der Reihenfolge der betrachteten Testziele nicht Gegenstand dieser Arbeit.

Algorithmus 7 erweitert Algorithmus 6 um die Wiederverwendung von Testfällen zwischen Testzielen. Hierfür wird, nachdem ein Testfall ω für das aktuelle Testziel tga generiert wurde, in der Zeile 9 eine For-Schleife eingefügt, die über die Menge der noch nicht abgedeckten Testziele UC iteriert. Für jedes Testziel $tga' \in UC$ wird geprüft, ob es durch den Testfall ω auf einem Software-Produkt abgedeckt wird, auf dem es bisher noch nicht abgedeckt wurde (Zeile 10). Deckt der Testfall ω das Testziel tga auf einem neuen Software-Produkt ab, wird der Testfall ω zusammen mit dem Testziel tga' und der Presence Condition pc_k als Tripel zur Test-Suite TS hinzugefügt (Zeile 11). Weiterhin wird die Menge der Software-Produkte auf denen das Testziel tga' bereits abgedeckt entsprechend der Presence Condition pc_k aktualisiert (Zeile 12).

Während der ARG_T-Konstruktion für SPL-Implementierungen werden viele Programmzustände mehrfach berechnet, da im ARG_T für SPLs keine Programmzustände vereinigt werden. Um die Anzahl der zu berechnenden Programmzustände eines ARG_T zu verringern, können wie in Abschnitt 3.2 Programmzustände zusammengefasst werden. Das Vorgehen zur ARG_T-Konstruktion für SPL-Implementierung mit der Vereinigung von Programmzuständen und die dabei auftretenden Herausforderungen werden im folgenden Abschnitt beschrieben.

4.1.6 *ARG_T mit Programmzustandsvereinigungen für Software-Produktlinien*

Während der ARG_T-Konstruktionen für SPL-Implementierungen wurden bisher viele Programmzustände mehrfach berechnet, da Programmzustände in den ARG_T

Algorithmus 7 Familienbasierte Test-Suite-Generierung mit Wiederverwendung von Testfällen zwischen Produkten und Testzielen

Input: Kontrollflussautomat $CFA = (L, \ell_0, \ell_t, E)$,
Feature-Modell FM,
Menge von Testzielautomaten $TG = \{tga_1, \ldots, tga_n\}$
Output: Abstrakte SPL-Test-Suite TS

1: $TS := \{\}$
2: $UC := TG$
3: **for each** $tga \in TG$ **do**
4: $\quad CS[tga] := fm$
5: **while** $UC \neq \emptyset$ **do**
6: $\quad tga := (Q, \Sigma, \Delta, q_0, F) \in UC$
7: $\quad pc_0 := CS[tga]$
8: $\quad$ **while** $\exists\ \omega := \psi_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{k-1}} \psi_k$ mit $\psi_i = (\ell_i, q_i, r_i)$, $r_i = (\phi_i, pc_i)$, $i = 0, \ldots, k$, $q_k \in F$ und $sat(pc_k \wedge CS[tga])$ **do**
9: $\quad\quad$ **for each** $tga' \in UC$ **do**
10: $\quad\quad\quad$ **if** tga' akzeptiert ω und $sat(pc_k \wedge CS[tga'])$ **then**
11: $\quad\quad\quad\quad TS := TS \cup (tga', \omega, pc_k)$
12: $\quad\quad\quad\quad CS[tga'] := CS[tga'] \wedge \neg pc_k$
13: $\quad\quad pc_0 := CS[tga]$
14: $\quad UC := UC \setminus tga$

nicht vereinigt wurden. Um die Anzahl der zu berechnenden Programmzustände eines ARG_T zu verringern, ist es deshalb sinnvoll Programmzustände wie in Abschnitt 3.2 zusammenzufassen. Das Vorgehen zur ARG_T-Konstruktion für SPL-Implementierung mit dem Vereinigen von Programmzuständen sowie die zwei folgenden, wesentlichen Herausforderungen und Optimierungen werden in diesem Abschnitt beschrieben:

1. Pfadbedingungen und Presence Conditions sind nicht mehr so leicht voneinander zu trennen, d. h. im Allgemeinen gilt eine Presence Condition nicht mehr für die gesamte Pfadbedingung eines Programmzustands, sondern Teile von Presence Conditions gelten für Teile von Pfadbedingungen (z. B. können zusammengeführte Kontrollflusspfade unterschiedliche Presence Conditions haben).

2. Programmzustände können durch mehrere verschiedene ARG_T-Pfade erreicht werden, die ein Testziel in verschiedenen Software-Produkten abdecken. Deshalb können aus einem Programmzustand unter Umständen mehrere Testfälle zur Abdeckung eines Testziels auf verschiedenen Software-Produkten abgeleitet werden.

Ein variabilitätsgewahrer ARG_T mit vereinigten Programmzuständen kann mit Hilfe der CPA $\text{CPA}_{\mathbb{R}'}$ aus Abschnitt 3.2.2 konstruiert werden, die sich aus der Komposition der Location-CPA $\text{CPA}_\mathbb{L}$, der Prädikaten-CPA $\text{CPA}_\mathbb{P}$ und der Automaten-CPA $\text{CPA}_{\mathbb{A}'}$ ergibt. Während der Konstruktion eines variablitätsgewahren ARG_T werden Programmzustände immer dann vereinigt, wenn der Kontrollfluss

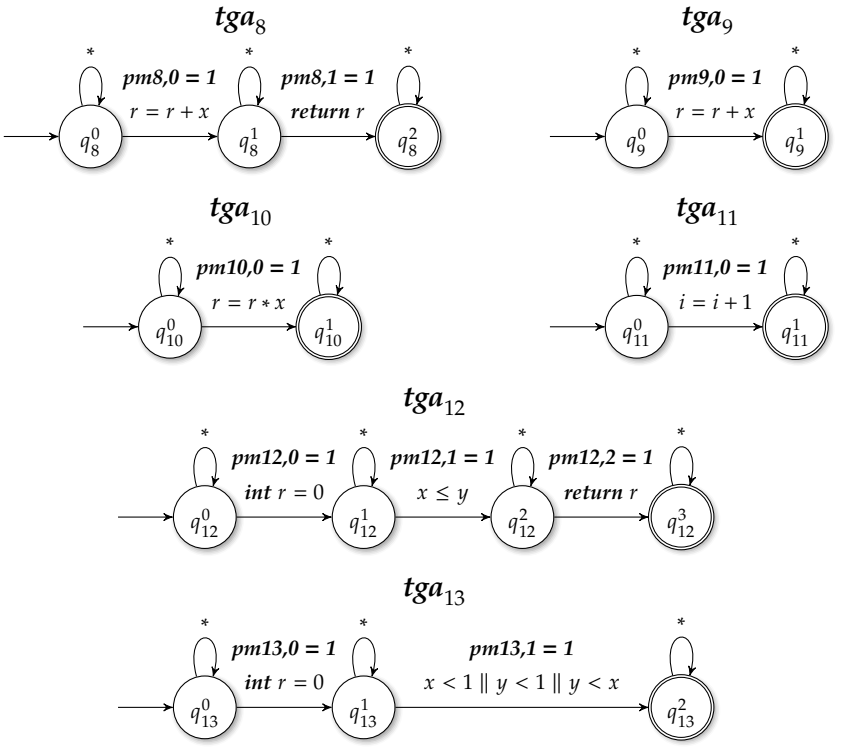


Abbildung 4.10: Testzielautomaten für die Calc-SPL mit Testpfadmarkern

zusammenfließt, und es wird für jeden Programmzustand gespeichert, in welchen Software-Produkten ein Programmzustand erreichbar ist. Um den Verlust von Kontroll- und Datenflussinformationen nach der Vereinigung von Programmzuständen zu verhindern, werden, wie in Abschnitt 3.2, Testpfadmarkervariablen verwendet. Abbildung 4.10 zeigt die TGAs aus Abbildung 4.2 mit Testpfadmarkervariablen. Dadurch dass die Pfadbedingung und die Presence Condition zusammen betrachtet werden, verhindern Testpfadmarkervariablen den Verlust von Kontroll- und Datenflussinformationen sowohl für die Pfadbedingung als auch für die Presence Condition eines Programmzustands. Dies bedeutet, dass nach einer Vereinigung von Programmzuständen eindeutig identifiziert werden kann, welche Teile einer Pfadbedingung für welche Produkte gültig ist und somit auf welchen Produkten ein abgeleiteter Testfall gültig ist.

Beispiel 4.15 (Ableitung variabilitätsgewahrer Testfälle aus ARG_T mit vereinigten Programmzuständen)**.** Abbildung 4.11 zeigt den ARG_T mit vereinigten Programmzuständen, der für das Testziel tga_{12} mit Testpfadmarkern aus Abbildung 4.10 mit Hilfe der CPA $\text{CPA}_{\mathbb{R}'}$ konstruiert wurde. Jeder Programmzustand besteht aus einer Programm-Location, einer Menge von TGA-Zuständen sowie einer Region, welche die Pfadbedingung und die Presence Condition des Programmzustands beinhaltet. Somit beschreibt jeder Programmzustand, welche Programm-Location in welchen Software-Produkten, durch welche Variablenbelegungen erreicht werden kann und in welchen TGA-Zuständen sich ein TGA tga_{12} in dem Programmzustand befinden kann. Da die Pfadbedingung und die

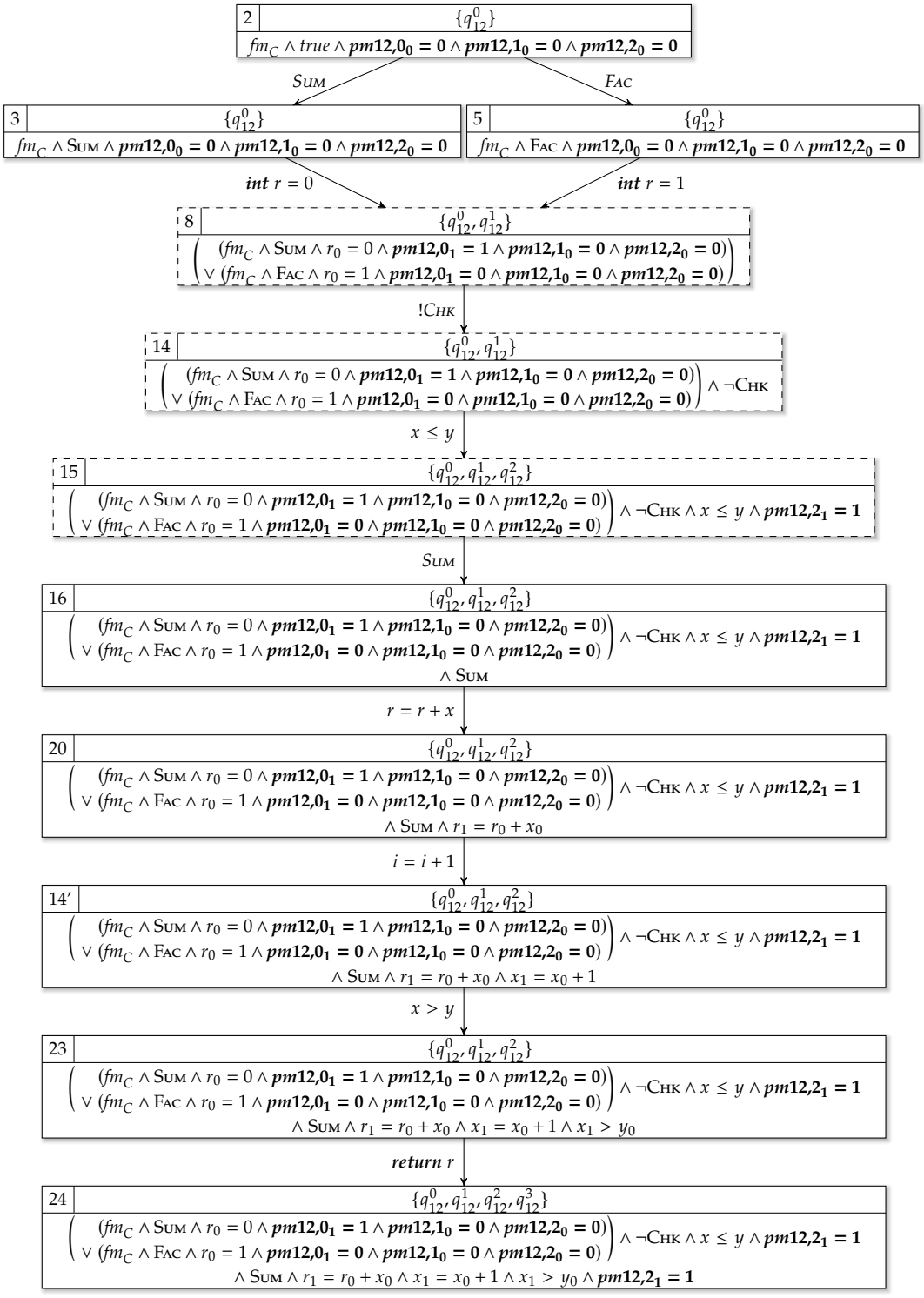


Abbildung 4.11: Ausschnitt des variabilitätsgewahren ARG_T der Calc-SPL für das Testziel tga_{12} mit der Vereinigung von Programmzuständen und Testpfadmarkern

Presence Condition eines Programmzustands nach der Vereinigung von Programmzuständen verwoben sein können, werden die Pfadbedingung und die Presence Condition nicht mehr getrennt im Programmzustand dargestellt.

Programmzustände dürfen immer dann vereinigt werden, wenn der Kontrollfluss zusammenfließt, z. B. in Programmzustand 8. Bei der Vereinigung zweier Programmzustände werden die TGA-Zustandsmengen der vereinigten Programmzustände zusammengefasst, sowie die Disjunktion der Regionen der Programmzustände gebildet. Die Testpfadmarkervariablen verhindern einen Verlust von Pfadinformationen für Programmvariablen und Features und sorgen somit dafür, dass die Analyse trotz der Vereinigung von Programmzuständen pfadsensitiv bleibt (vgl. Beispiel 3.31).

Der TGA tga_{12} ist in Programmzustand 24 in einem Endzustand, weshalb in Programmzustand 24 ein Testfall zur Abdeckung des Testziels tga_{12} abgeleitet werden kann. Sowohl der Ein- und der Ausgabevektor des Testfalls als auch die Presence Condition werden aus der Region von Programmzustand 24 abgeleitet. Wie in Beispiel 3.31 geschieht die Testfallableitung durch das Lösen der Region in Programmzustand 24 konjugiert mit den Testpfadmarkervariablen des TGAs tga_{12}, d. h. der Formel

$$\left(\begin{array}{c} \left(\begin{array}{c} (fm_C \wedge \textsc{Sum} \wedge r_0 = 0 \wedge \boldsymbol{pm12{,}0_1 = 1} \wedge \boldsymbol{pm12{,}1_0 = 0} \wedge \boldsymbol{pm12{,}2_0 = 0}) \\ \vee\, (fm_C \wedge \textsc{Fac} \wedge r_0 = 1 \wedge \boldsymbol{pm12{,}0_1 = 0} \wedge \boldsymbol{pm12{,}1_0 = 0} \wedge \boldsymbol{pm12{,}2_0 = 0}) \end{array}\right) \\ \wedge\, \neg\textsc{Chk} \wedge x \leq y \wedge \boldsymbol{pm12{,}2_1 = 1} \wedge\, \textsc{Sum} \\ \wedge\, r_1 = r_0 + x_0 \wedge x_1 = x_0 + 1 \wedge x_1 > y_0 \wedge \boldsymbol{pm12{,}2_1 = 1} \\ \wedge\, (\boldsymbol{pm12{,}0_1 = 1} \wedge \boldsymbol{pm12{,}1_1 = 1} \wedge \boldsymbol{pm12{,}2_1 = 1}) \end{array}\right).$$

Für die Programmvariablen ist eine mögliche Lösung $x = 1, y = 2$ und $r = 3$ und für die Feature-Variablen entsprechend Sum, ¬Fac und ¬Chk. Somit ist ein Testfall zur Abdeckung des Testziels tga_{12} $tc_2 = ([x = 1, y = 2], [r = 3], fm_C \wedge \textsc{Sum} \wedge \neg\textsc{Fac} \wedge \neg\textsc{Chk})$.

Da in der Region von Programmzustand 24 nur die Features auftauchen, die für den ARG_T-Pfad gemäß des Testfalls tc_2 relevant sind, ist die Presence Condition allgemein genug, um eine Menge von Software-Produkte zu spezifizieren für die der Testfall gültig ist. Gäbe es ein Feature in der Calc-SPL, welches auf dem ARG_T-Pfad $2 - 3 - 8 - 14 - 15 - 16 - 20 - 14' - 23 - 24$ nicht verwendet worden wäre, dann würde das Feature nicht in der Region von Programmzustand 24 auftauchen und könnte somit einen beliebigen Wert in der Presence Condition von Testfall tc_2 annehmen.

Beispiel 4.15 zeigt wie mit Hilfe der in früheren Abschnitten eingeführten CPAs und Algorithmen variabilitätsgewahre Testfälle für SPLs abgeleitet werden können, wobei in einem ARG_T der für ein Testziel konstruiert wird Programmzustände vereinigt werden. Das Vereinigen von Programmzuständen in einem variabilitätsgewahren ARG_T hat einen weiteren Effekt, der für eine Optimierung der Testfallgenerierung für SPLs verwendet werden kann. Für einzelne Software-Produkte wurde bisher aus jedem Programmzustand, in dem ein Testziel erfüllt war, genau ein Testfall abgeleitet, der das Testziel abdeckt. Für eine SPL-Implementierung kann es

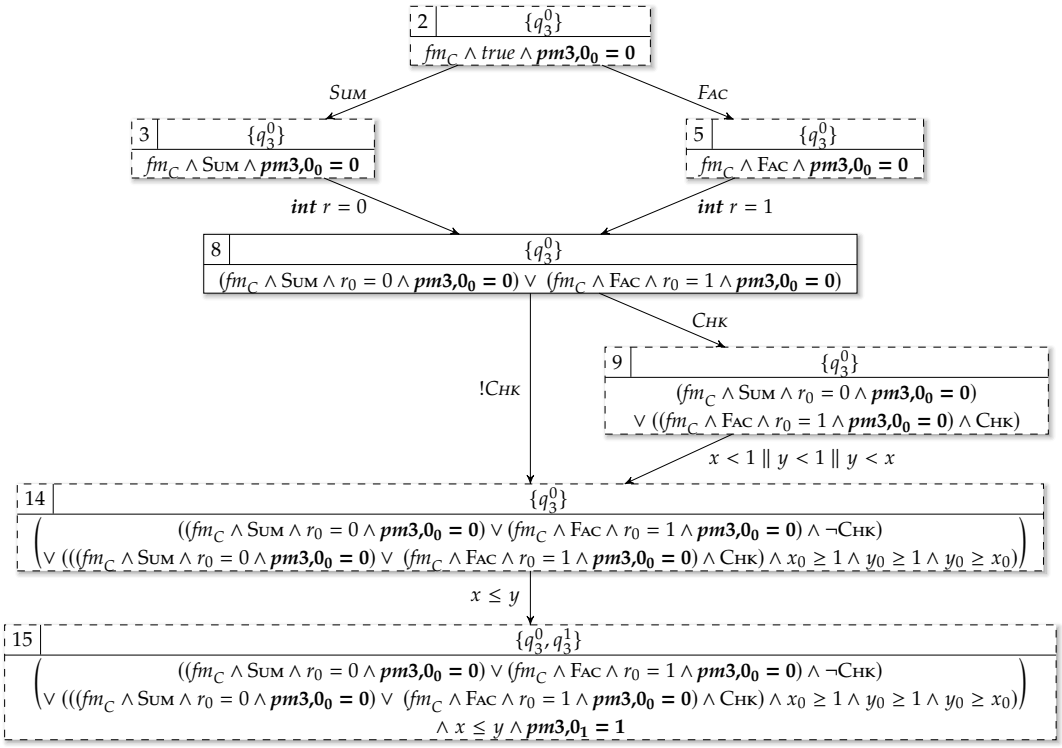


Abbildung 4.12: Ausschnitt des variabilitätsgewahren ARG_T der Calc-SPL für das Testziel tga_3 mit der Vereinigung von Programmzuständen und Testpfadmarkern

jedoch vorkommen, dass es sinnvoll ist, mehrere Testfälle aus einem Programmzustand abzuleiten, wenn ein Testziel in diesem Programmzustand erfüllt ist. Dies ist immer dann der Fall, wenn die verschiedenen ARG_T-Pfade zu unterschiedlichen Software-Produkten gehören und die zugehörigen Testfälle ein Testziel somit auf verschiedenen Software-Produkten abdecken.

Beispiel 4.16 (Ableitung mehrerer Testfälle aus einem Programmzustand)**.** Abbildung 4.12 zeigt den variabilitätsgewahren ARG_T, der für das Testziel tga_3 aus Abbildung 3.14 konstruiert wurde. In Programmzustand 15 ist der TGA tga_3 in einem Endzustand und das Testziel tga_3 ist somit erfüllt. Wie in Beispiel 4.15 kann ein Testfall zu der Region bestehend aus einem Eingabevektor und einer Presence Condition abgeleitet werden. Beispielsweise der Testfall $tc_1 = ([x = 1, y = 2], fm_C \wedge \text{Sum} \wedge \neg\text{Fac} \wedge \neg\text{Chk})$ der den Programmpfad $\omega_1 = 2 - 3 - 8 - 14 - 15$ traversiert.

Neben dem Programmpfad ω_1 gibt es drei weitere Programmpfade, die zum Programmzustand 15 führen und das Testziel tga_3 abdecken. Die drei Programmpfade sind alle jeweils in unterschiedlichen Software-Produkten vorhanden und können somit als abstrakte Testfälle zur vollständigen SPL-Abdeckung von Testziel tga_3 verwendet werden. Zu diesem Zweck können aus Programmzustand 15 weitere konkrete Testfälle abgeleitet werden, indem die Regionen von Programmzustand 15 mehrfach gelöst wird, wobei die Region nach jeder Lösung durch die Negation der Presence Condition der vorherigen Lösung eingeschränkt wird.

Somit wird nach der Ableitung des Testfalls tc_1 die Region von Programmzustand 15 um die Negation der Presence Condition des Testfalls tc_1 erweitert, d. h. zu der Formel

$$\left(\begin{array}{c} ((fm_C \wedge \textsc{Sum} \wedge r_0 = 0 \wedge \boldsymbol{pm3{,}0_0 = 0}) \\ \vee (fm_C \wedge \textsc{Fac} \wedge r_0 = 1 \wedge \boldsymbol{pm3{,}0_0 = 0}) \wedge \neg\textsc{Chk}) \\ \vee \left(((\textsc{Sum} \wedge r_0 = 0 \wedge \boldsymbol{pm3{,}0_0 = 0}) \vee\ (\textsc{Fac} \wedge r_0 = 1 \wedge \boldsymbol{pm3{,}0_0 = 0}) \wedge \textsc{Chk}\right) \\ \wedge\ x_0 \geq 1 \wedge y_0 \geq 1 \wedge x_0 \geq y_0) \end{array}\right) .$$
$$\wedge\ x \leq y \wedge \boldsymbol{pm3{,}0_1 = 1} \wedge \boldsymbol{pm3{,}0_1 = 1}$$
$$\wedge\ \neg(fm_C \wedge \textsc{Sum} \wedge \neg\textsc{Fac} \wedge \neg\textsc{Chk})$$

Aus dieser Formel kann ein weiterer Testfall $tc_2 = ([x = 1, y = 2], fm_C \wedge \textsc{Sum} \wedge \neg\textsc{Fac} \wedge \textsc{Chk})$ abgeleitet werden, der auf anderen Produkten ausführbar ist als der Testfall tc_1. Dieses Vorgehen wird solange wiederholt bis keine Lösung mehr aus der Region konjugiert mit negierten Presence Conditions der bereits generierten Testfälle mehr gefunden werden kann. Aus Programmzustand 15 können noch zwei weitere Testfälle abgeleitet werden, die Testfälle $tc_3 = ([x = 1, y = 2], fm_C \wedge \textsc{Sum} \wedge \textsc{Fac} \wedge \neg\textsc{Chk})$ und $tc_4 = ([x = 1, y = 2], fm_C \wedge \textsc{Sum} \wedge \textsc{Fac} \wedge \textsc{Chk})$. Die Test-Suite $TS = \{tc_1, tc_2, tc_3, tc_4\}$ deckt das Testziel tga_3 vollständig auf der Calc-SPL ab.

Die konkreten Testfälle aus der Test-Suite *TS* wurden aus verschiedenen Programmpfaden abgeleitet, weshalb die Testfälle die gleichen Eingaben mit unterschiedlichen Presence Conditions haben können und somit verschiedenen abstrakten Testfällen entsprechen. Zur Minimierung der Test-Suite können die Testfälle zusammengefasst werden. Es ist jedoch nicht immer zu erwarten, dass aus verschiedenen Regionen die gleichen Eingabewerte berechnet werden wie es in diesem Beispiel der Fall ist. Beispielsweise könnte für den Testfall tc_2 auch der Eingabevektor $[x = 2, y = 4]$ berechnet werden.

Die Kombination der SPL-Testgenerierung aus Abschnitt 4.1 mit der pfadsensitiven Vereinigung von Programmzuständen in ARG_T aus Abschnitt 3.2 kann zu einer Verkleinerung des Programmzustandsraums führen und somit zu einer effizienteren Testgenerierung. Allerdings muss im Allgemeinen für jedes Testziel weiterhin ein ARG_T zur Ableitung einer Test-Suite konstruiert werden. Um dies zu vermeiden kann die SPL-Testgenerierung mit der Test-Suite-Generierung basierend auf Multi-Property-Checking aus Abschnitt 3.3 kombiniert werden, wodurch mehrere Testziele während einer ARG_T-Konstruktion parallel behandelt werden können und somit weniger ARG_T für die Test-Suite-Generierung für eine Menge von Testzielen konstruiert werden müssen.

4.2 Erweiterung der Testgenerierung für Software-Produktlinien um Multi-Property-Checking

Durch die familienbasierte Test-Suite-Generierung aus Abschnitt 4.1 können Berechnungsinformationen während der Test-Suite-Generierung wiederverwendet

werden. Dies umfasst insbesondere die Wiederverwendung von Testfällen zwischen Produkten und Testzielen. Wie in Abschnitt 3.3 ist es zusätzlich wünschenswert, Ergebnisse der ARG_T-Konstruktionen bei der Test-Suite-Generierung wiederzuverwenden. Dies kann zu einer Verringerung der Anzahl der ARG_T-Konstruktionen während der Test-Suite-Generierung führen und somit zu einer Effizienzsteigerung bei der Ableitung einer Test-Suite für eine gegebene Menge von Testzielen. Zu diesem Zweck wird die familienbasierte Test-Suite-Generierung in diesem Abschnitt mit der Test-Suite-Generierung basierend auf Multi-Property-Checking aus Abschnitt 3.3 kombiniert.

4.2.1 *Test-Generierung für Software-Produktlinien mit Multi-Property-Checking*

Das Ziel der Kombination der familienbasierten ARG-Konstruktion mit Multi-Property-Checking ist es, einen einzigen ARG_M (vgl. Definition 3.32) zur Ableitung einer SPL-Test-Suite für mehrere Testziele gleichzeitig zu konstruieren. Dadurch soll entsprechend der Test-Suite-Generierung mit Multi-Property-Checking für Software-Produkte vermieden werden, dass Teile des ARG einer IUT mehrfach für verschiedene Testziele konstruiert werden. Stattdessen soll ein einziger ARG_M wie in Abschnitt 3.3 konstruiert werden, der alle Pfadinformationen zur Ableitung einer SPL-Test-Suite für eine Menge von Testzielen enthält.

Zur Ableitung einer SPL-Test-Suite für mehrere Testziele muss ein ARG_M für jeden Programmzustand speichern,

- in welchen Software-Produkten der Programmzustand erreichbar ist, d. h. eine Presence Condition, und
- in welchen TGA-Zuständen sich die verschiedenen Testziele befinden, sodass für jedes Testziel bekannt ist, wann das Testziel erfüllt ist.

Mit Hilfe dieser Informationen kann bestimmt werden, welches Testziel in welcher Programm-Location auf welchen Produkten erfüllt ist. Wie bei der Testgenerierung für SPLs ohne Multi-Property-Checking erlaubt die Presence Condition ebenfalls die Einschränkung des Programmzustandsraums auf die Software-Produkte, die noch nicht abgedeckte Testziele enthalten können.

Beispiel 4.17 (Familienbasierter ARG_M)**.** Abbildung 4.13 zeigt den ARG_M, der zur Ableitung einer Test-Suite für die Testziele tga_9 und tga_{13} aus Abbildung 4.2 konstruiert wurde. Jeder Programmzustand des familienbasierten ARG_M enthält eine Programm-Location, die aktuellen (nicht-deterministisch berechneten) TGA-Zustände der verschiedenen Testziele, eine Presence Condition und eine Region r. Die Region $r = \phi \wedge pc$ besteht aus einer Pfadbedingung ϕ und einer Presence Condition pc, die beide für eine bessere Übersichtlichkeit getrennt dargestellt werden können, da im ARG_M keine Vereinigung von Programmzuständen vorgenommen wird. Somit beschreibt jeder Programmzustand mit welchen Variablenbelegungen (Pfadbedingung) welche Programm-Location in welchen Software-Produkten (Presence Condition) erreichbar ist und ob Testziele in dem Programmzustand erfüllt sind (TGA-Zustände). Diese

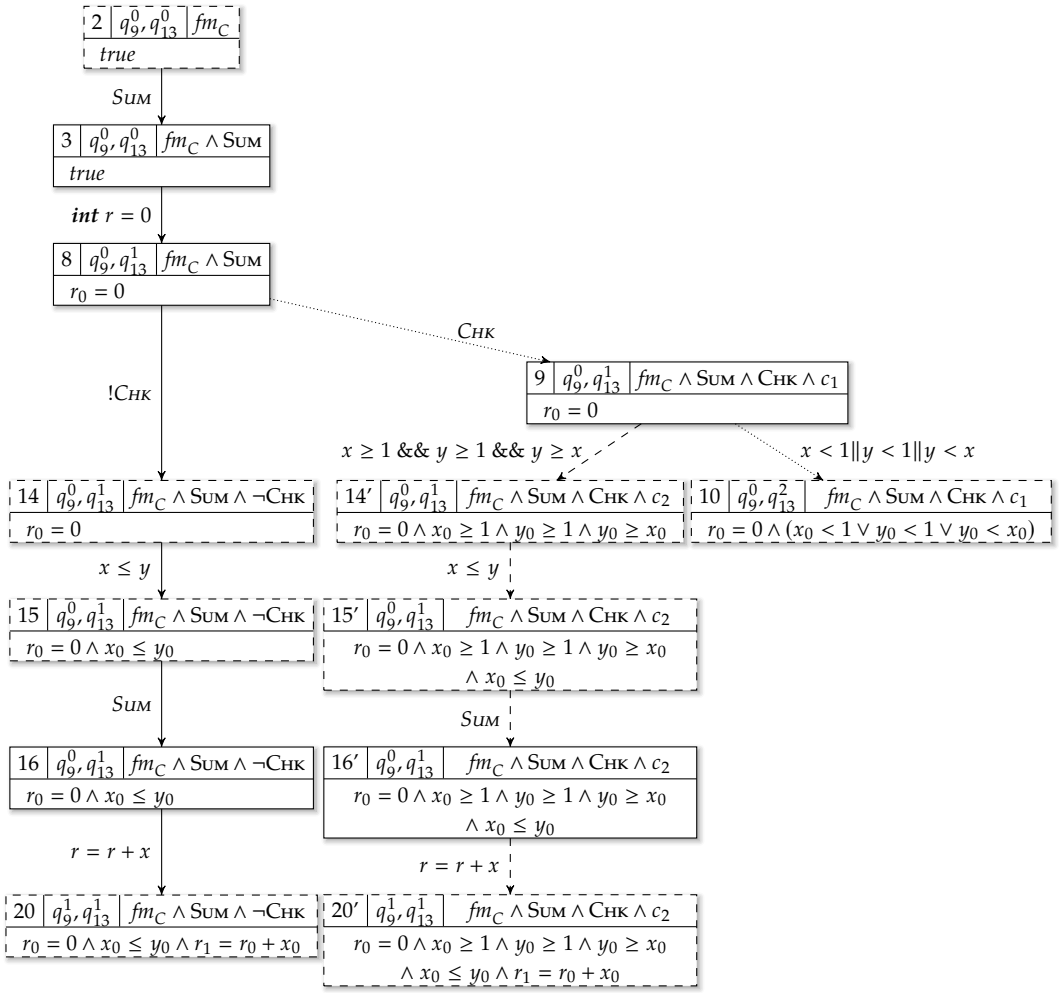


Abbildung 4.13: Ausschnitt des ARG_M der Calc-SPL für die Testziele tga_9 und tga_{13} mit $c_1 = (fm_C \wedge \neg(fm_C \wedge \text{Sum} \wedge \neg\text{Chk})) \vee (fm_C)$ und $c_2 = (fm_C \wedge \neg(fm_C \wedge \text{Sum} \wedge \neg\text{Chk})) \vee (fm_C \wedge \neg(fm_C \wedge \text{Sum} \wedge \text{Chk} \wedge c_1))$

Informationen erlauben die Ableitung einer SPL-Test-Suite für mehrere Testziele. Der ARG_M aus Abbildung 4.13 wurde in drei Schritten konstruiert, welche im Folgenden beschrieben werden. Dabei wird die Kombination der Techniken zur Test-Suite-Generierung mit Multi-Property-Checking und der familienbasierten Test-Suite-Generierung veranschaulicht.

Im ersten Schritt wird der Teil des ARG_M konstruiert, der mit durchgezogen dargestellten Transitionen verbunden ist. In Programmzustand 20 ist der TGA tga_9 in einem Endzustand und das Testziel tga_9 ist somit abgedeckt. Aus der Pfadbedingung und der Presence Condition kann, wie in Beispiel 4.7, ein Testfall $tc_1 = ([x = 1, y = 2], pc_1)$ mit $pc_1 = fm_C \wedge \text{Sum} \wedge \neg\text{Chk}$ abgeleitet werden. Da das Testziel tga_9 noch auf anderen Software-Produkten abgedeckt werden könnte und das Testziel tga_{13} noch gar nicht abgedeckt wurde, muss die Konstruktion des ARG_M zur Ableitung einer SPL-Test-Suite fortgesetzt werden.

In Beispiel 4.9 wird bei der familienbasierten Test-Suite-Generierung eine SPL-Test-Suite für ein einzelnes Testziel abgeleitet. Nach der Ableitung eines

Testfalls wird der Programmzustandsraum auf die Programmzustände der Software-Produkte eingeschränkt, für die das Testziel noch nicht abgedeckt wurde. Hierfür wird die Presence Condition von jedem neuen Programmzustand mit der Negation der Presence Condition des abgeleiteten Testfalls konjugiert. Dadurch werden keine neuen Programmzustände mehr betrachtet, die nur in Software-Produkten enthalten sind, auf denen das Testziel bereits durch einen Testfall abgedeckt wurde.

Dieser Ansatz zur Einschränkung des Programmzustandsraums kann bei der Berücksichtigung mehrerer Testziele gleichzeitig während einer ARG_M-Konstruktion nicht ohne Anpassungen übernommen werden. Angenommen der Programmzustandsraum würde nach der Ableitung des Testfalls tc_1 auf die Software-Produkte eingeschränkt werden, die der Presence Condition $\neg pc_1$ entsprechen, dann würden keine Programmzustände mehr berechnet werden, die nur im Software-Produkt {Sum, ¬Chk} enthalten sind. Würde das Testziel tga_{13} in dem Software-Produkt {Sum, ¬Chk} enthalten sein, würde es durch diese Einschränkung des Programmzustandsraums nicht möglich sein, einen Testfall zur Abdeckung des Testziels tga_{13} auf dem Software-Produkt (Sum, ¬Chk) abzuleiten, da der entsprechende Programmzustand, in dem der TGA tga_{13} in einem Endzustand ist, nicht berechnet werden würde.

Somit muss die Einschränkung des Programmzustandsraums so angepasst werden, dass die Programmzustände der Software-Produkte berechnet werden, die mindestens ein Testziel enthalten können, das noch nicht durch einen Testfall auf diesen Software-Produkten abgedeckt wurde. Um dies zu erreichen, wird der zu konstruierende Programmzustandsraum durch die Bedingung eingeschränkt, die aus der Disjunktion der Presence Conditions für jedes Testziel resultiert, für die das jeweilige Testziel noch abgedeckt werden muss. Für dieses Beispiel bedeutet das, dass der Programmzustandsraum durch die Bedingung $c = (fm_C \wedge \neg pc_1) \vee (fm_C)$ eingeschränkt wird. Das Testziel tga_9 wurde bereits auf den Software-Produkten pc_1 abgedeckt und das Testziel tga_{13} wurde noch auf keinem Software-Produkt abgedeckt (d. h. es muss noch für die Presence Condition fm_C abgedeckt werden). Die Disjunktion von $(fm_C \wedge \neg pc_1)$ und fm_C ergibt die Bedingung c, welche die weitere Konstruktion des Programmzustandsraums einschränkt.

Im zweiten Schritt wird der Teil des ARG_M konstruiert, der mit gepunktet dargestellten Transitionen verbunden ist. In Programmzustand 10 ist das Testziel tga_{13} in einem Endzustand, und es kann ein Testfall $tc_2 = ([x = 0, y = 2], pc_2)$ mit $pc_2 = fm_C \wedge \text{Sum} \wedge \text{Chk} \wedge c_1$ zur Abdeckung des Testziels tga_{13} aus Programmzustand 10 abgeleitet werden. Da die Testziele tga_9 und tga_{13} noch auf weiteren Software-Produkten erfüllbar sein könnten, wird die ARG_M-Konstruktion weiter fortgesetzt. Die weitere ARG_M-Konstruktion wird dabei nach obigem Verfahren auf die Software-Produkte eingeschränkt, auf denen das Testziel tga_9 oder das Testziel tga_{13} noch nicht abgedeckt ist. Die entsprechende Bedingung zur Einschränkung des Programmzustandsraums ist $c_2 = (fm_C \wedge \neg(fm_C \wedge \text{Sum} \wedge \neg\text{Chk})) \vee (fm_C \wedge \neg(fm_C \wedge \text{Sum} \wedge \text{Chk} \wedge c_1))$. Dabei

ist der erste Teil der Bedingung c_2 die Einschränkung für das Testziel tga_9 und der zweite Teil die Einschränkung für das Testziel tga_{13}.

Im dritten Schritt wird der Teil des ARG_M konstruiert, der mit gestrichelten Transitionen verbunden ist. In Programmzustand 20′ ist das Testziel tga_9 erfüllt und es kann ein Testfall $tc_3 = ([x = 1, y = 2], pc_3)$ mit $pc_3 = fm_C \wedge \textsc{Sum} \wedge \textsc{Chk} \wedge c_2$ abgeleitet werden.

Die Test-Suite $TS = \{tc_1, tc_2, tc_3\}$ erfüllt eine vollständige SPL-Abdeckung der Testzielmenge $\{tga_{10}, tga_{13}\}$ auf der Calc-SPL.

Beispiel 4.17 zeigt die Kombination der Techniken für die familienbasierte Test-Suite-Generierung und des Multi-Property-Checkings. Die wesentliche Erweiterung, damit die Kombination korrekt funktioniert, ist die Anpassung der Konstruktion der Bedingung, mit der die Konstruktion eines ARG_M auf bestimmte Software-Produkte eingeschränkt wird. Es ist nicht mehr ausreichend einfach die Produkte während der ARG_M-Konstruktion auszuschließen, für die ein Testziel abgedeckt wurde, da andere Testziele im Allgemeinen noch auf diesen Produkten abgedeckt werden können. Somit dürfen Produkte erst von der ARG_M-Konstruktion ausgeschlossen werden, wenn alle Testziele auf diesen Produkten abgedeckt wurden bzw. als nicht erreichbar identifiziert wurden. Um dies zu erreichen wird die ARG_M-Konstruktion jedes Mal, wenn ein Testfall abgeleitet wurde, um die Disjunktion der Presence Conditions eingeschränkt, für die einzelne Testziele noch abgedeckt werden müssen.

Für die Umsetzung der familienbasierten ARG_M-Konstruktion kann die CPA $\text{CPA}_{\mathbb{R}_M}$ aus Abschnitt 3.3.2 verwendet werden. Die CPA $\text{CPA}_{\mathbb{R}_M}$, bestehend aus der Komposition der Location-CPA $\text{CPA}_\mathbb{L}$, der Prädikaten-CPA $\text{CPA}_\mathbb{P}$ und der Automaten-CPA $\text{CPA}_{\mathbb{A}_M}$, arbeitet wie in Abschnitt 3.3.2 beschrieben. Während der ARG_M-Konstruktion durch die $\text{CPA}_{\mathbb{R}_M}$ berechnet die Prädikaten-CPA $\text{CPA}_\mathbb{P}$ wie in Abschnitt 4.1.3 die Region für jeden Programmzustand, d. h. eine Pfadbedingung und eine Presence Condition.

Im folgenden Abschnitt wird ein Algorithmus beschrieben, der auf Basis der CPA $\text{CPA}_{\mathbb{R}_M}$ einen ARG_M konstruiert und dabei den zu konstruierenden Programmzustandsraum gemäß der Strategie aus Beispiel 4.17 immer weiter einschränkt.

4.2.2 *Vollständige Test-Suite-Generierung für Software-Produktlinien mit Multi-Property-Checking*

In diesem Abschnitt wird ein Algorithmus zur Generierung einer Test-Suite basierend auf der familienbasierten ARG_M-Konstruktion aus dem letzten Abschnitt beschrieben. Der Algorithmus kombiniert und erweitert dabei Algorithmus 5 zur Generierung von Test-Suiten mit Multi-Property-Checking und Algorithmus 7 zur Generierung von SPL-Test-Suiten. Die Erweiterung besteht aus der Konstruktion der Bedingung zur Einschränkung der ARG_M-Konstruktion auf bestimmte Produkte unter Berücksichtigung mehrerer Testziele (vgl. Beispiel 4.17). Weiterhin werden in dem Algorithmus Testfälle sowohl zwischen Testzielen als auch zwischen Software-Produkten wiederverwendet.

Algorithmus 8 beschreibt das Vorgehen zur Generierung einer SPL-Test-Suite mit Multi-Property-Checking für eine Menge von Testzielen aus Beispiel 4.17. Die

Algorithmus 8 Familienbasierte Test-Suite-Generierung mit Multi-Property-Checking und Wiederverwendung von Testfällen zwischen Produkten und Testzielen

Input: Kontrollflussautomat $CFA = (L, \ell_0, \ell_t, E)$,
Feature-Modell FM,
Menge von Testzielautomaten $TG = \{tga_1, \ldots, tga_n\}$
Output: Abstrakte SPL-Test-Suite TS

1: $TS := \{\}$
2: $UC := TG$
3: **for each** $tga \in TG$ **do**
4: $\quad CS[tga] := fm$
5: **while** $UC \neq \emptyset$ **do**
6: $\quad TG' := \{tga'_1, \ldots, tga'_l\} \subseteq UC$ mit $tga'_i = (Q'_i, \Sigma'_i, \Delta'_i, q'_{0_i}, F'_i), 1 \leq i \leq l$
7: $\quad pc_0 := \bigvee CS[tga_i], 1 \leq i \leq l$
8: $\quad$ **while** $\exists\, \omega := \psi_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{k-1}} \psi_k$ mit $\psi_i = (\ell_i, q'_{i_1}, \ldots, q'_{i_l}, r_i)$,
$\qquad\qquad r_i = (\phi_i, pc_i), i = 0, \ldots, k$,
$\qquad\qquad \exists q'_{k_j} \in F'_j$ und $sat(pc_k \wedge CS[tga_j]), 1 \leq j \leq l$ **do**
9: $\qquad$ **for each** $tga'_m \in UC$ mit $q'_{k_m} \in F'_m, 1 \leq m \leq l$ **do**
10: $\qquad\quad$ **if** tga'_m akzeptiert ω und $sat(pc_k \wedge CS[tga'_m])$ **then**
11: $\qquad\qquad TS := TS \cup (tga'_m, \omega_k, pc_k)$
12: $\qquad\qquad CS[tga'_m] := CS[tga'_m] \wedge \neg pc_k$
13: $\qquad pc_0 := \bigvee CS[tga_i], 1 \leq i \leq l$
14: $\quad UC := UC \setminus TG'$

Eingaben für den Algorithmus sind der Kontrollflussautomat *CFA* der IUT, ein Feature-Modell *fm* als aussagenlogischer Term und eine Menge von Testzielen *TG*, die abgedeckt werden sollen. Die Ausgabe des Algorithmus ist eine abstrakte SPL-Test-Suite *TS*. Die Test-Suite *TS* wird mit der leeren Menge initialisiert (Zeile 1) und die Menge der noch nicht abgearbeiteten Testziele *UC* mit der Menge aller Testziele *TG* (Zeile 2). Der Vektor *CS* speichert für jedes Testziel *tga* eine Presence Condition über Feature-Variablen, die beschreibt für welche Produkte *tga* noch nicht abgedeckt ist. *CS* wird für jedes Testziel *tga* mit dem Feature-Modell *fm* initialisiert (Zeile 3–4), d. h. das Testziel *tga* noch auf keinem Produkt abgedeckt ist. Solange die Menge der noch nicht bearbeiteten Testziele *UC* nicht leer ist, iteriert der Algorithmus über diese Menge und generiert SPL-Testfälle zur Abdeckung der Testziele in *UC* (Zeile 5). Zur Generierung von Testfällen wählt der Algorithmus die nächsten zu betrachtende Testziele $TG' \subseteq UC$ aus der Menge *UC* aus (Zeile 6) und initialisiert die Presence Condition pc_0 des initialen Programmzustands mit Disjunktion der Presence Conditions für welche die Testziele aus TG' jeweils noch nicht abgedeckt sind (Zeile 7). Zu Beginn ist dies immer das Feature-Modell *fm*, d. h. bei der Konstruktion des ARG_M zur Testfallableitung werden alle Produkte einer SPL betrachtet. Es werden solange Testfälle abgeleitet bis es keinen neuen Testfall mehr gibt, der ein Testziel $tga_j \in TG'$ auf einem Produkt abdeckt auf dem das Testziel noch nicht abgedeckt ist (Zeile 8). Angenommen es gibt einen neuen abstrakten Testfall ω, der das Testziel tga_j auf einer Menge von Produkten abdeckt, auf der es noch nicht

abgedeckt ist (d. h. $sat(pc_k \wedge CS[tga_j])$ liefert *true* zurück). In diesem Fall wird für jedes Testziel $tga'_m \in UC$ geprüft, ob es durch den Testfall ω erfüllt wird (Zeile 9–10). Ist dies der Fall wird das Testziel tga'_m zusammen mit dem Testfall ω und der Presence Condition pc_k, für die das Testziel durch den Testfall abgedeckt wird, als Tripel zur Test-Suite *TS* hinzugefügt (Zeile 11). Weiterhin wird gespeichert, dass das Testziel tga'_m für eine neue Menge von Produkten abgedeckt wurde (Zeile 12). Nachdem der Testfall zusammen mit allen Testzielen, die der Testfall abdeckt, und der entsprechenden Presence Condition zur Test-Suite hinzugefügt wurde, wird die weitere ARG_M-Konstruktion wird auf die Produkte eingeschränkt, auf denen noch mindestens ein Testziel nicht abgedeckt ist (Zeile 13). Gibt es keinen neuen Testfall mehr, der ein Testziel $tga_j \in TG'$ auf einem Software-Produkt abdeckt auf dem es noch nicht abgedeckt wurde, wird die Menge TG' aus der Menge der noch nicht bearbeiteten Testziele *UC* entfernt (Zeile 14).

Algorithmus 8 kombiniert verschiedene Möglichkeiten Testartefakten wiederzuverwenden, die in dieser Arbeit vorgestellt wurden. Zum einen werden Testfälle zwischen Produkten durch Presence Conditions und zwischen Testzielen (vgl. Zeile 10–12) wiederverwendet und zum anderen werden durch die Konstruktion eines ARG_M, der mehrere Testziele gleichzeitig berücksichtigt, Teile des ARG_M zwischen den Testzielen wiederverwendet, d. h. nur einmal für eine Menge von Testzielen TG' konstruiert. Für eine bessere Verständlichkeit der neuen Konzepte wurden bei der Test-Suite-Generierung, wie sie im vorherigen und in diesem Abschnitt vorgestellt wurden, keine Vereinigungen von Programmzuständen während der ARG_M-Konstruktion durchgeführt, die für eine Verkleinerung des Programmzustandsraums führen können und somit zu einer effizienteren Test-Suite-Generierung. Im folgenden Abschnitt wird deshalb die hier vorgestellte SPL-Test-Suite-Generierung auf Basis von Multi-Property-Checking mit der Vereinigung von Programmzuständen in ARG_T für SPLs ohne Verlust von Pfadsensitivität aus Abschnitt 4.1.6 kombiniert.

4.2.3 *ARG_M mit pfadsensitiver Programmzustandsvereinigungen für Software-Produktlinien mit Multi-Property-Checking*

Um die Größe des zu konstruierenden Programmzustandsraums auch während der SPL-Testgenerierung auf Basis von Multi-Property-Checking soweit wie möglich zu reduzieren, wird in diesem Abschnitt die familienbasierte Testgenerierung auf Basis von Multi-Property-Checking um die pfadsensitive Vereinigung von Programmzuständen für ARG_T aus Abschnitt 4.1.6 erweitert. Mit Hilfe der pfadsensitiven Vereinigung können Programmzustände bei der Konstruktion eines ARG_M unter Berücksichtigung einer Menge von Testzielen immer dann vereinigt werden, wenn der Kontrollfluss im CFA der IUT zusammenfließt. Wie in Abschnitt 3.2 werden Testpfadmarker verwendet, um ARG_M-Pfade Testzielen zuordnen zu können.

Beispiel 4.18 (Ableitung von SPL-Testfällen aus ARG_M mit vereinigten Programmzuständen)**.** Abbildung 4.14 zeigt einen ARG_M mit vereinigten Programmzuständen, der zur Ableitung von Testfällen für die Testziele tga_9 und

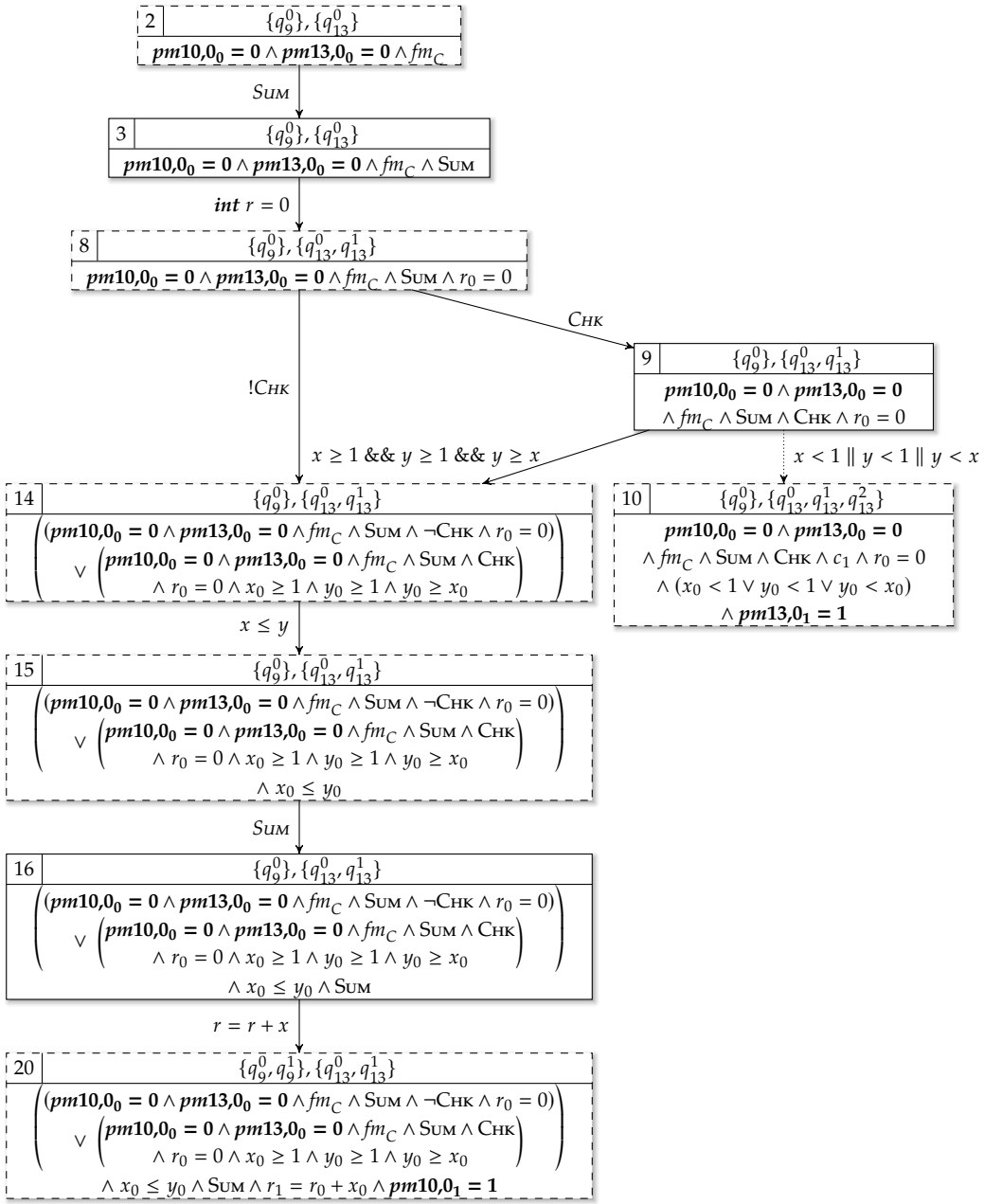


Abbildung 4.14: Ausschnitt des ARG_M der Calc-SPL für die Testziele tga_9 und tga_{13} mit $c_1 = (fm_C \wedge \neg(fm_C \wedge \text{Sum} \wedge (\text{Chk} \vee \neg\text{Chk}))) \wedge (\text{fm}_C)$

tga_{13} konstruiert wurde. Die Region der Programmzustände wird wieder in einer Komponente dargestellt, da die Pfadbedingungen und die Presence Conditions der Programmzustände nach der Vereinigung von Programmzuständen verwoben sein können. Weiterhin wird die Region um Testpfadmarkervariablen zum Erhalt der Pfadsensitivität für eine korrekte Testfallableitung angereichert. Der ARG_M wurde in zwei Schritten konstruiert.

Im ersten Schritt wurde der Teil des ARG_M konstruiert, der mit durchgezogen dargestellten Transitionen verbunden ist. In Programmzustand 20 ist das Test-

ziel tga_9 erfüllt. Gemäß des Vorgehens aus Beispiel 4.16 können aus Programmzustand 20 die zwei Testfälle $tc_1 = ([x = 1, y = 2], fm_C \wedge \text{Sum} \wedge \neg\text{Chk})$ und $tc_2 = ([x = 1, y = 2], fm_C \wedge \text{Sum} \wedge \text{Chk})$ abgeleitet werden. Die Testfälle tc_1 und tc_2 entsprechen verschiedenen ARG_M-Pfaden und können auf verschiedenen Software-Produkten ausgeführt werden. Durch die Testfälle tc_1 und tc_2 wird das Testziel tga_9 auf den Software-Produkten $fm_C \wedge \text{Sum} \wedge (\text{Chk} \vee \neg\text{Chk})$ abgedeckt. Da das Testziel tga_{13} noch auf keinen Software-Produkten abgedeckt wurde, kann die weitere ARG_M-Konstruktion zur weiteren Abdeckung der Testziele tga_9 und tga_{13} durch die Bedingung $c_1 = (fm_C \wedge \neg(fm_C \wedge \text{Sum} \wedge (\text{Chk} \vee \neg\text{Chk}))) \vee (fm_C)$ eingeschränkt werden (vgl. Beispiel 4.17).

Im zweiten Schritt wurde der Teil des ARG_M konstruiert, der mit gepunktet dargestellten Transitionen verbunden ist. Die ARG_M-Konstruktion wurde dabei entsprechend der Bedingung c_1 auf Produkte eingeschränkt auf denen mindestens ein Testziel noch nicht abgedeckt wurde. In Programmzustand 10 ist das Testziel tga_{13} erfüllt und es kann ein Testfall $tc_3 = ([x = -1, y = 0], fm_C \wedge \text{Sum} \wedge \text{Chk} \wedge c_1)$ zu Abdeckung des Testziels tga_{13} abgeleitet werden.

Die Test-Suite $TS = \{tc_1, tc_2, tc_3\}$ erfüllt eine vollständige SPL-Abdeckung der Testzielmenge $\{tga_{10}, tga_{13}\}$ auf der Calc-SPL.

Der ARG_M aus Abbildung 4.14 kann durch Algorithmus 8 konstruiert werden. Dabei wird die CPA $\text{CPA}_{\mathbb{R}'_M}$ verwendet, die mehrere TGAs verwalten kann (vgl. Abschnitt 3.3.2).

Wie in Abschnitt 3.3 führt die Verwendung von Testpfadmarkervariablen zum Erhalt der Pfadsensitivität zu einer Vergrößerung des Programmzustandsraums der IUT. Dadurch können während der ARG_M-Konstruktion zwar Programmzustände vereinigt werden, aber die ARG_M-Konstruktion kann gleichzeitig sehr viel aufwendiger werden. Auch für die Test-Suite-Generierung für SPLs mit Hilfe von Multi-Property-Checking ist es deshalb sinnvoll, die Menge von Testzielen zu partitionieren und somit in handhabbare Teilmengen zu zerlegen. Für die Partitionierung können zusätzliche SPL-spezifische Kriterien verwendet werden. Zum Beispiel, dass Testziele in einer Partition zusammen betrachtet werden, die in der gleichen Menge von Produkten enthalten sind. Welche Testziele in welchen Produkten enthalten sind, kann dabei durch eine vorgeschaltete statische Analyse ermittelt werden. Die Entwicklung von Strategien zur Partitionierung der Testzielmenge ist jedoch nicht Bestandteil dieser Arbeit.

4.3 Experimentelle Evaluation

In diesem Abschnitt wird die Kombination von zwei Wiederverwendungsstrategien von Testfällen während der Test-Suite-Generierung für SPLs evaluiert. Dies umfasst zum einen die Wiederverwendung von Testfällen zwischen Produkten (vgl. Abschnitt 4.1.4) und zum anderen die Wiederverwendung von Testfällen zwischen Testzielen (vgl. Abschnitt 3.1.7). Dafür vergleichen wir zwei verschiedene Ideen zur Test-Suite-Generierung für SPLs miteinander. Zum einen die produktweise Test-Suite-Generierung (*PW*) basierend auf dem Test-Suite-Generierungsansatz aus Abschnitt 3.1, d. h. es werden für jedes Produkt nacheinander Test-Suiten generiert. Zum anderen die familienbasierte Test-Suite-Generierung (*FB*), d. h. es werden

Test-Suiten auf einer SPL-Implementierung mit Wiederverwendung von Testfällen zwischen Produkten generiert. Zur Evaluation der Auswirkungen von Wiederverwendung von Testfällen zwischen Testzielen wird der produktweise Test-Suite-Generierungsansatz *PW* entsprechend erweitert zu dem Ansatz *PWTG*. Weiterhin kombinieren wir die Wiederverwendung von Testfällen zwischen Produkten und Testzielen (*FBTG*, vgl. Abschnitt 4.1.5), um mögliche Synergien zu evaluieren. Die zwei produktweisen Ansätzen bilden dabei die Baseline für die Evaluation. Um potenzielle positive und negative Auswirkungen auf die Test-Suite-Generierung, die durch die Verwendung von Multi-Property-Checking entstehen können bei der Evaluation der Wiederverwendungsstrategien ausschließen zu können, werden wir in dieser Evaluation kein Multi-Property-Checking verwenden. Die Auswirkungen von Multi-Property-Checking auf die Test-Suite-Generierung wurden bereits in Abschnitt 3.4 evaluiert.

Forschungsfragen

Das Ziel der Evaluation ist es, die zwei Forschungsfragen **RQ1** und **RQ2** bzgl. der verschiedenen Wiederverwendungsstrategien von Testfällen zu beantworten.

- **RQ1 (Effizienz):** Erhöht die Wiederverwendung von Testfällen durch familienbasierte Test-Suite-Generierungsansätze (*FB* und *FBTG*) die Effizienz der Test-Suite-Generierung für SPLs verglichen mit produktweisen Test-Suite-Generierungsansätzen (*PW* und *PWTG*)?
- **RQ2 (Effektivität):** Wie beeinflusst die Verwendung von familienbasierten Test-Suite-Generierungsansätzen (*FB* und *FBTG*) die Effektivität der Test-Suite-Generierung für SPLs verglichen mit produktweisen Test-Suite-Generierungsansätzen (*PW* und *PWTG*)?

Die Forschungsfrage **RQ1** zielt darauf ab, den Einfluss der Wiederverwendung von Testfällen auf die Effizienz der SPL-Test-Suite-Generierung zu evaluieren. Die Effizienz umfasst dabei zum einen den Berechnungsaufwand für die Test-Suite-Generierung im Sinne von durchschnittlicher CPU-Zeit und zum anderen die Größe generierter Test-Suiten (Anzahl der Testfälle), welche ein Indikator für die Ausführungszeit der Test-Suiten sein kann. Somit werden die zwei abhängigen Variablen *durchschnittliche CPU-Zeit* und *Test-Suite-Größe* zur Bestimmung der Testeffizienz verwendet.

Die Forschungsfrage **RQ2** zielt darauf ab, den Einfluss der Wiederverwendung von Testfällen auf die Effektivität der SPL-Test-Suite-Generierung zu evaluieren. Hierfür wird untersucht, ob durch die Wiederverwendung von Testfällen mit den familienbasierten SPL-Test-Suite-Generierungsansätzen eine höhere Abdeckung von Testzielen im Gegensatz zur produktweisen Test-Suite-Generierung erreicht werden kann, trotz der vermutlich höheren Komplexität des Kontrollflusses von SPL-Implementierungen. Um dies zu erfassen, werden die drei abhängigen Variablen *Anzahl erfüllbarer Testziele*, *Anzahl unerfüllbarer Testziele* und *Anzahl von Testzielen, deren Erreichbarkeit unbekannt ist*, verwendet.

Experimentaufbau

Fallstudien. Für die Evaluation haben wir zwei Zielsysteme ausgewählt, die bereits in anderen Evaluationen verwendet wurden, z. B. im Bereich Produktlinienverifikation und im Bereich Detektion von Feature-Interaktionen [AvRW+13].

- *Mine-Pump (MP)* implementiert ein Wasserpumpensystem basierend auf dem CONIC-Projekt [KMSL83]. Die SPL-Implementierung enthält 279 Zeilen Code, 7 Features (z. B. einen Methansensor) und 64 Produktkonfigurationen.
- *E-Mail (EM)* basiert auf dem Model eines E-Mail-Systems, das von Hall entwickelt wurde [Hal05]. Die SPL-Implementierung enthält 233 Zeilen Code, 4 Features (z. B. Verschlüsselung und eine automatische Weiterleitung) und 8 Produktkonfigurationen.

Tool-Unterstützung. Um die Forschungsfragen **RQ1** und **RQ2** zu beantworten, haben wir einen SPL-Test-Suite-Generator basierend auf dem C-Model-Checking-Framework CPAchecker [BK11, BHT07] implementiert. Für die Spezifikation von Abdeckungskriterien verwenden wir FQL [HSTV10, HTSV10]. Als SMT-Solver kam SMTInterpol zum Einsatz.

Experiment-Design. Als Baseline verwenden wir die produktweisen Test-Suite-Generierungsansätze *PW* und *PWTG* wie oben beschrieben und vergleichen sie mit den familienbasierten Test-Suite-Generierungsansätzen *FB* und *FBTG*. Dabei wird *Basic-Block-Abdeckung* als Abdeckungskriterium für die Test-Suite-Generierung verwendet, d. h. jedes einfache Programmstatement entspricht einem Testziel.

Aufbau der Messungen. Die gemessene CPU-Zeit umfasst den kompletten Test-Suite-Generierungsprozess bestehend aus der TGA-Konstruktion aus einer FQL-Query, der CPA-Initialisierung und der Ausführung des Test-Generators. Jedes Testziel wird maximal 900s betrachtet, d. h. wenn ein Testziel nach 900s nicht erfüllt wurde, wird das nächste Testziel betrachtet. Der gesamte Test-Suite-Generierungsprozess wird pro SPL-Implementierung bzw. Produktimplementierung nach maximal 24h abgebrochen. Die Evaluation wurde auf einem Computer mit einem E5-2650-Prozessor (2GHz) mit 30GB RAM ausgeführt.

Datensammlung. Für jedes der vier Experimente (*PW*, *PWTG*, *FB*, *FBTG*) sammeln wir die folgenden Daten. Um die Effizienz der Test-Suite-Generierung zu evaluieren, messen wir die CPU-Zeit des gesamten Test-Suite-Generierungsprozesses in Stunden, die Anzahl der benötigten Model-Checker-Aufrufe und die Größe der generierten Test-Suiten. Zur Bestimmung der Effektivität messen wir die Anzahl der erfüllbaren und der unerfüllbaren Testziele sowie die Anzahl der Testziele, deren Erreichbarkeit unbekannt ist (z. B. wegen einer Zeitüberschreitung). Zur Beantwortung der Forschungsfragen werden die für die einzelnen Experimente gemessenen Werte zur Bestimmung der Effizienz (**RQ1**) und der Effektivität (**RQ2**) miteinander verglichen.

Tabelle 4.1: Ergebnisse der Evaluation (# CPA: Model-Checker-Ausführungen; die Anazhl der Testziele bei den produktweisen Ansätzen ergibt sich aus der Summe der Testziele der Produkte)

Zielsystem	Generierungsansatz	#Testziele	#Bearbeitet	#Erfüllbar	#Unerfüllbar	#Unbekannt	#Testfälle	#CPA	CPU-Zeit (in h)
MP	PW	3792	3792	3298	490	4	3298	3720	13,9
	PWTG	3792	3792	3300	490	2	427	847	3,7
	FB	93	93	42	1	50	52	145	24
	FBTG	93	81	42	1	38	46	127	24
EM	PW	1084	1084	804	280	0	800	1874	32,9
	PWTG	1084	1084	804	280	0	169	533	10,4
	FB	198	131	78	16	37	154	285	24
	FBTG	198	160	91	30	39	103	224	24

Ergebnisse

Die Ergebnisse zur Beantwortung der Forschungsfragen **RQ1** und **RQ2** wurden in Tabelle 4.1 zusammengefasst.

RQ1. Zu beobachten ist, dass die familienbasierten Ansätze *FB* und *FBTG* grundsätzlich mit weniger Model-Checker-Aufrufen deutlich kleinere Test-Suiten generieren als die produktweisen Ansätze *PW* und *PWTG*. Für die Test-Suite-Generierung benötigen die familienbasierten Ansätze jedoch mehr Zeit als *PW* und *PWTG* brauchen und schöpfen das 24-Stunden-Zeitlimit immer aus. Für das E-Mail-System schöpft der Ansatz *PW* ebenfalls das Zeitlimit aus und überschreitet es aus technischen Gründen sogar deutlich (32.9h). Somit hat die Wiederverwendung von Testfällen zwischen Produkten und Testzielen insbesondere auf die Test-Suite-Größe einen positiven Einfluss. Durch die Kombination der Wiederverwendung von Testfällen zwischen Produkten und Testzielen (*FBTG*) können die positiven Auswirkungen der Test-Suite-Generierung mit Wiederverwendung von Testfällen zwischen Produkten (*FB*) weiter verbessert werden.

RQ2. Bei der Test-Suite-Generierung mit *FB* und *FBTG* ist die Erreichbarkeit für wesentlich mehr Testziele *unbekannt*, da es mehr Überschreitungen der Zeitlimits gibt als bei den produktweisen Ansätzen *PW* und *PWTG*.

Diskussion

Mit Hilfe der familienbasierten Test-Suite-Generierung, d. h. mit Wiederverwendung von Testfällen zwischen Produkten, können die Größe generierter Test-Suiten und die Anzahl der dafür benötigten Model-Checker-Aufrufe verglichen mit der produktweisen Test-Suite-Generierung drastisch verringert verwenden. Aus

der Kombination der Wiederverwendung von Testfällen zwischen Produkten und Testzielen ergeben sich Synergien, welche die positiven Auswirkungen der Wiederverwendungsstrategien weiter verstärken. Werden die Test-Suite-Größen und die Anzahl von Model-Checker-Aufrufen unter Berücksichtigung der Testziele, deren Erreichbarkeit unbekannt ist, extrapoliert, schneidet die familienbasierte Test-Suite-Generierung weiterhin besser als die produktweise Test-Suite-Generierung. Jedoch benötigt die familienbasierte Test-Suite-Generierung mehr CPU-Zeit als die produktweise Test-Suite-Generierung aufgrund der höheren Komplexität des Kontrollflusses der SPL-Implementierungen, die sämtliche Kontrollflüsse aller Produkte in sich vereinigen. Dies führt ebenfalls dazu, dass es nach dem Überschreiten des Zeitlimits zur Test-Suite-Generierung bei den familienbasierten Ansätzen mehr Testziele gibt, deren Erreichbarkeit unbekannt ist. Dies könnte jedoch durch eine Erhöhung der Zeitlimits vermieden werden.

Zusammengefasst können durch die Wiederverwendung von Testfällen die Größe generierter Test-Suiten und die dafür benötigten Model-Checker-Aufrufe reduziert werden. Dabei ist jedoch zu berücksichtigen, dass die familienbasierten Ansätze durch die Komplexität durch die Kodierung aller Varianten in einem Programm mehr CPU-Zeit für die Test-Suite-Generierung benötigen können.

Gefährdung der Validität

Die größte Gefährdung der Validität der Evaluation ist die geringe Anzahl der betrachteten Zielsysteme. Da nur zwei Zielsysteme betrachtet wurden, geben die Ergebnisse nur Hinweise auf die tatsächlichen Auswirkungen der Wiederverwendung von Testfällen zwischen Produkten und Testzielen. Für eine valide Aussage müssen weitere Experimente mit weiteren Zielsystemen durchgeführt werden. Eine weitere Gefährdung der Validität ist, dass die Test-Suite-Generierung nur für ein Abdeckungskriterium evaluiert wurde. Obwohl Basic-Block-Abdeckung ein weitverbreitetes Abdeckungskriterium ist, sollten in weiteren Experimenten andere Abdeckungskriterien zur Test-Suite-Generierung betrachtet werden. Außerdem sollte untersucht werden, inwieweit die Wiederverwendung von Testfällen und somit das Weglassen von potenziell unterschiedlichen Testfällen die Effektivität der generierten Test-Suiten beeinflussen kann. Weiterhin ist der fehlende Vergleich zu anderen Test-Suite-Generierungsansätzen eine Gefährdung der Validität der Evaluation. Jedoch unterstützen aktuelle Tools zur Test-Suite-Generierung entweder keine SPLs oder die Tools arbeiten nicht auf C-Code bzw. sogar komplett modellbasiert wie auch im folgenden Abschnitt zu verwandten Arbeiten dargelegt werden wird. Eine weitere Gefährdung der Validität könnte durch die Zeitüberschreitungen während der Test-Suite-Generierung entstehen. Diesbezüglich haben wir die Ergebnisse jedoch auf Basis unserer Erfahrungen extrapoliert, wodurch die Gefährdung der Validität minimiert werden kann.

4.4 Verwandte Arbeiten

In diesem Abschnitt diskutieren wir zu den in diesem Kapitel behandelten Ansätzen und Techniken verwandte Arbeiten. Auf Arbeiten zu Testansätzen für einzelne

Software-Produkte wird in diesem Abschnitt nicht eingegangen, da diese bereits in Abschnitt 3.5 diskutiert wurden. Stattdessen liegt der Fokus auf SPL-Ansätzen.

Testgenerierung für Software-Produktlinien

Die familienbasierte Generierung von Test-Suiten zur Erreichung von SPL-Abdeckung von Testzielen wurde bereits in anderen Arbeiten untersucht. Cichos et al. [COLS11] haben einen modellbasierten (Black-Box)-Ansatz zur Generierung von SPL-Test-Suiten basierend auf dem Model-Checker SPIN beschrieben, der eine nachträgliche Analyse bzgl. der Wiederverwendung von Testfällen zwischen Software-Produkten erlaubt. Die Wiederverwendung von Testfällen lässt dabei die Wiederverwendung von Erreichbarkeitsanalyseergebnissen zwischen Testzielen außer Acht. Luthmann et al. [LSBL17] beschreiben die familienbasierte Testgenerierung für SPLs mit dem Model-Checker Uppaal auf Basis von konfigurierbaren, parametrisierten Timed-Automata unter Berücksichtigung des Zeitverhaltens einer SPL. Dabei wird die Wiederverwendung von Testfällen zwischen Testzielen und Produkten explizit unterstützt. Die Wiederverwendung von Teilen des Programmzustandsraums einer IUT, wie es die Verwendung von Multi-Property-Checking zur Test-Suite-Generierung erlaubt, wird hingegen nicht unterstützt.

Ein weiterer familienbasierter Testgenerierungsansatz zur modellbasierten Testgenerierung auf Basis von Featured Transition Systems stammt von Devroey et al. [DPS14, DPL+14]. Der Ansatz erlaubt zwar die Wiederverwendung von Testfällen zwischen Produkten, jedoch nicht die Wiederverwendung von Ergebnissen der Erreichbarkeitsanalyse zwischen Testzielen und Produkten wie es die Verwendung von Multi-Property-Checking erlaubt. Weiterhin ist der Ansatz auf modellbasierte Testgenerierung beschränkt. In [VBM15] beschreiben Varshosaz et al. einen inkrementellen, modellbasierten Testgenerierungsansatz, der schrittweise für Software-Produkte einer SPL Testfälle generiert. Dabei werden Testfälle zwischen Produkten so wiederverwendet, dass für ein betrachtetes Produkt nur Testfälle für die Teile der Implementierung generiert werden müssen, die noch nicht durch Testfälle abgedeckt werden, die für andere Produkte generiert wurden. Im Gegensatz dazu erlaubt unser Ansatz einen weitaus höheren Grad der Wiederverwendung von Testartefakten, da der Ansatz zum einen familienbasiert ist und somit alle Produkte einer SPL gleichzeitig betrachtet wurden anstatt nacheinander und zum anderen Multi-Property-Checking zur Testgenerierung verwendet. Somit können Erreichbarkeitsanalyseergebnisse vielfältig wiederverwendet werden. Weiterhin unterstützt der Ansatz aus dieser Arbeit sowohl die modellbasierte Testgenerierung als auch die Testgenerierung auf Basis von Quellcode.

Model-Checking

Die Anwendung von CPAchecker zur Verifikation von SPLs wurde von Apel et al. beschrieben [AvRW+13], wobei eine BDD-Analyse für die Wiederverwendung von Verifikationsergebnissen zwischen Produkten einer SPL verwendet wird [ABF+13]. Dieser Ansatz unterstützt allerdings nicht die Wiederverwendung von Verifikationsergebnissen zwischen zu überprüfenden Properties indem Multi-Property-Checking-Techniken angewendet werden. Im Gegensatz dazu unterstützt der Mul-

ti-Property-Checking-Ansatz von Apel et al. [ABM+16] keine Wiederverwendung von Verifikationsergebnissen zwischen Produkten einer SPL.

Andere Model-Checking-Ansätze für SPLs wie [CHSL11, CCP+12] verwenden ebenfalls symbolische Model-Checking-Techniken und behandeln Features als spezielle Eingabevariablen. Die Ansätze betrachten jedoch nur eine familienbasierte Evaluation von einer bestimmten Model-Checking-Anfrage ohne systematische Wiederverwendung von Analyseergebnissen.

Die Feature-basierte SPL-Model-Checking-Technik von Li et al. [LKF02] sowie die inkrementelle SPL-Model-Checking-Technik von Lochau et al. [LMBR14] führen eine schrittweise Exploration des Programmzustandsraums ähnlich zu CEGAR-Ansätzen aus. Dabei betrachten diese Ansätze jedoch nur eine einzelne Property in einem Schritt anstatt mehrere Properties (Testziele).

Weitere inkrementelle Model-Checking-Ansätze legen den Fokus auf eine schrittweise Test-Suite-Verfeinerung sowie der Re-Test-Auswahl ähnlich zum Regressionstesten [ER10, LSKL12]. Die Generierung von Testfällen ist jedoch kein bestandteil dieser Ansätze.

4.5 Zusammenfassung und Ausblick

In diesem Kapitel haben wir eine Technik zur automatisierten, familienbasierten Testgenerierung von Komponententests für SPLs präsentiert, die auf den Techniken aus Kapitel 3 basiert. Dabei wurde zum einen die Wiederverwendung von Testfällen zwischen Testzielen um die Wiederverwendung von Testfällen zwischen Produkten erweitert, sodass kleinere Test-Suiten generiert werden können und dabei weniger ARG-Konstruktionen durchgeführt werden können. Zum anderen wurde die familienbasierte Testgenerierung um die Testgenerierung mit Multi-Property-Checking erweitert, sodass mehrere Testziele einer SPL-Implementierung während einer ARG-Konstruktion gleichzeitig betrachtet werden können und somit implizit Teile des berechneten Programmzustandsraums zur Abdeckung dieser Testziele zwischen Produkten wiederverwendet werden können.

Die Ergebnisse der Evaluation haben gezeigt, dass die Kombination der Wiederverwendung von Testfällen zwischen Produkten und Testzielen während der Test-Suite-Generierung dazu führen kann, dass die Größe generierter Test-Suiten und die dafür benötigten Model-Checker-Aufrufe im Vergleich zur Test-Suite-Generierung ohne diese Wiederverwendung drastisch reduziert wird. Dabei können die familienbasierten Ansätze aufgrund der Komplexität der SPL-Implementierungen mehr CPU-Zeit für die Test-Suite-Generierung als produktweise Ansätze benötigen.

In zukünftigen Forschungsarbeiten sollte die in diesem Kapitel vorgestellte SPL-Test-Suite-Generierung noch detaillierter evaluiert werden. Insbesondere wäre es interessant, die Evaluation auf weitere Zielsysteme zu erweitern sowie die Auswirkungen von SPS und der Partitionierung der Testzielmenge auf die Test-Suite-Generierung für SPLs detailliert zu untersuchen.

Aus technischer Sicht ist die Partitionierung der Testzielmenge (vgl. Kapitel 3) für SPL-Implementierungen ein weiterer interessanter Aspekt, der untersucht werden sollte. Hierbei stellt sich die Frage, ob die Test-Suite-Generierung mit Multi-

Property-Checking für SPLs weiter verbessert werden kann, wenn der Aufbau von SPLs bei der Partitionierung der Testzielmenge berücksichtigt wird. Durch statische Analysen kann beispielsweise analysiert werden, welche Testziele in einer ähnlichen Produktmenge vorkommen. Dieses Wissen könnte danach für eine Partitionierung der Testzielmenge verwendet werden, die speziell die Charakteristika von Produktlinienimplementierungen in Betracht zieht und als Grundlage für ein Sampling-Kriterium zur Auswahl von repräsentativen Software-Produkten dienen auf denen Testfälle tatsächlich ausgeführt werden.

5

SPEZIFIKATION UND VALIDIERUNG DYNAMISCHER SOFTWARE-PRODUKTLINIEN

In Abschnitt 2.3 wurde die DCU-Fallstudie vorgestellt. Die DCU-Fallstudie umfasst die Software zur Kontrolle der Hardware-Komponenten eines Teilchenbeschleunigers im HIT für die Bestrahlungstherapie von Tumoren. Insbesondere wurde bei der Beschreibung der DCU-Fallstudie hervorgehoben, warum die DCU nicht nur eine SPL ist, sondern aufgrund ihrer dynamischen (Re-)Konfigurationseigenschaften eine DSPL ist. In diesem Kapitel werden Konzepte zur Spezifikation von DSPLs vorgestellt, welche existierende DSPL-Spezifikationsansätze erweitern (vgl. Forschungsherausforderung FH3.1), um die Rekonfigurationseigenschaften und -prozesse realer Systeme wie der DCU hinreichend beschreiben zu können. Dies umfasst insbesondere die Möglichkeit der Spezifikation von temporalen Abhängigkeiten zwischen Features während des (Re-)Konfigurationsprozesses einer DSPL. Für sicherheitskritische Systeme wie der DCU-Fallstudie ist eine Validierung ihrer Spezifikation essentiell, um eine sichere und fehlerlose Ausführung des Systems zum Schutz von Patienten zu garantieren. Deshalb werden Techniken zur automatisieren Validierung von DSPL-Spezifikationen vorgestellt (vgl. Forschungsherausforderung FH3.2), welche die Erweiterungen der existierenden DSPL-Spezifikationsansätze berücksichtigen. Zusätzlich zu den Ansätzen zur Validierung der Spezifikation von DSPLs werden wir basierend auf den Ansätzen aus den Kapiteln 3 und 4 beschreiben, wie DSPLs getestet werden können (vgl. Forschungsherausforderung FH3.3). Dabei wird insbesondere auf das Testen des Rekonfigurationsverhaltens von DSPLs eingegangen. Die Beiträge, die in diesem Kapitel vorgestellt werden, basieren größtenteils auf den Veröffentlichungen [BLL+14] und [LBHS17].

5.1 Spezifikation dynamischer Software-Produktlinien

In diesem Abschnitt werden durch die DCU-Fallstudie motivierte Konzepte zur Erweiterung bestehender Modellierungsansätze für DSPLs vorgestellt. Dazu wird mit einer allgemeinen Beschreibung der Architektur von DSPLs in Abschnitt 5.1.1 begonnen, wie sie in dieser Arbeit verwendet wird. Danach werden in Abschnitt 5.1.2 erweiterte Feature-Modelle eingeführt. Erweiterte Feature-Modelle dienen als Basis für einen stufenweisen Konfigurationsprozess für Software-Produkte, welcher in Abschnitt 5.1.3 beschrieben wird. Die stufenweise Konfiguration von Software-Produkten erlaubt es unter anderem die Produkt-Konfiguration auf mehrere Stakeholder mit verschiedenen Verantwortlichkeiten aufzuteilen. Dabei wird zwischen statischen und dynamischen Bindungszeiten für Features unterschieden und es können komplexe, temporale Restriktionen zwischen den Bindungszeiten von Fea-

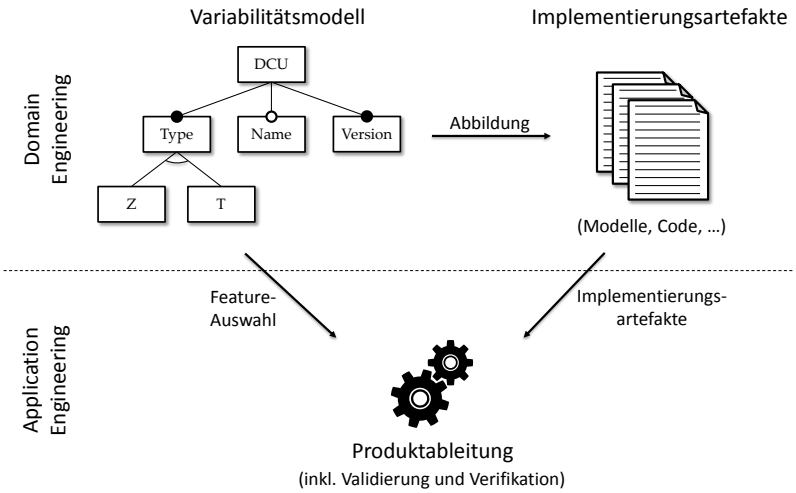


Abbildung 5.1: Artefakte einer SPL

tures und somit auch zwischen den Konfigurationsentscheidungen der Stakeholder formuliert werden. Zur Einschränkungen des erlaubten Rekonfigurationsverhaltens einer DSPL werden in Abschnitt 5.1.4 Rekonfigurationsautomaten beschrieben, welche die Spezifikation erlaubter Rekonfigurationen anhand sich ändernder Anwendungskontexte ermöglichen.

5.1.1 *Aufbau dynamischer Software-Produktlinien*

Beim traditionellen Software-Entwicklungsprozess wird ein einzelnes Software-System entworfen und entwickelt. SPLE betrachtet hingegen die Entwicklung einer ganzen Software-Familie, wobei die Wiederverwendung von Implementierungsartefakten (z. B. Code oder Modelle) den Schwerpunkt von SPLE bildet, wie in Abschnitt 2.3.2 dargestellt wurde. Zur Maximierung der Wiederverwendbarkeit von Implementierungsartefakten zwischen verschiedenen Software-Produkten einer Software-Familie werden die Implementierungsartefakte während des Domain Engineerings auf Features abgebildet, die in einem Variabilitätsmodell beschrieben werden. Danach werden die Implementierungsartefakte der SPL entsprechend einer Feature-Auswahl während der Produktableitung zu einem validen Produkt zusammengesetzt. Dies umfasst ebenfalls die Validierung und Verifikation des Produktes [ABKS13]. Somit besteht eine SPL mindestens aus einem Variabilitätsmodell und wiederverwendbaren Implementierungsartefakten sowie einer Abbildung zwischen den Features des Variabilitätsmodells und den Implementierungsartefakten [ABKS13], wie in Abbildung 5.1 dargestellt ist.

DSPLs erweitern SPLs um einen dynamischen Konfigurationsaspekt, der die Rekonfiguration zwischen Produkten zur Laufzeit erlaubt. Während einer Rekonfiguration wird die Feature-Auswahl eines Software-Produktes geändert, wodurch sich ebenfalls die aktiven Implementierungsartefakte des Software-Produktes ändern und somit ebenfalls das Verhalten des Software-Produktes. Um sicherzustellen, dass nur erlaubte Rekonfigurationen einer DSPL zur Laufzeit ausgeführt werden, reichen ein Variabilitätsmodell und Implementierungsartefakte einer SPL unter

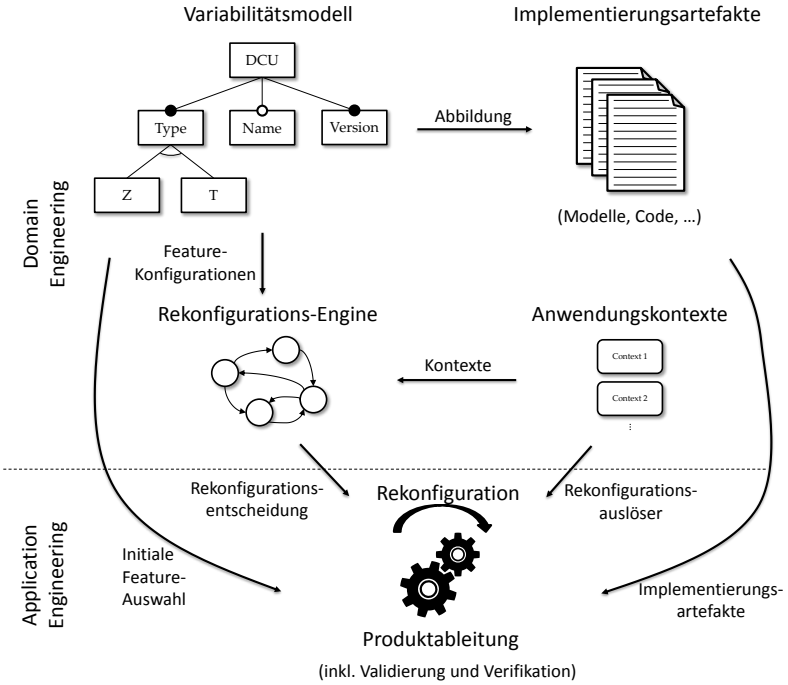


Abbildung 5.2: Artefakte einer DSPL

Umständen nicht aus. Inspiriert von (selbst-)adaptiven Systemen, können DSPLs deshalb um eine Rekonfigurations-Engine und ein Kontextmodell erweitert werden [HSSF06, LK06, Sal15]. DSPL-Ansätze implementieren dabei häufig eine MAPE-K-Schleife in ihren Rekonfigurations-Engines [HPS12, SLR13]. Die in dieser Arbeit vorgestellten Modellierungstechniken sind jedoch unabhängig von einer konkreten Implementierung mit oder ohne MAPE-K-Schleife, weshalb auf diesen Aspekt nicht näher eingegangen wird.

Um eine SPL zu einer DSPL zu erweitern, verwenden wir zum Variabilitätsmodell und den Implementierungsartefakten eine Rekonfigurations-Engine und Anwendungskontexte, ähnlich wie in [HSSF06, LK06, Sal15]. Die Rekonfigurations-Engine wird zur Steuerung der Rekonfigurationen einer DSPL verwendet, wobei die Anwendungskontexte die Rekonfigurationen auslösen, sodass sich das System entsprechend der aktuellen Anforderungen oder Kundenbedürfnisse adaptiert. Wie in Abbildung 5.2 zu sehen ist, werden somit in dieser Arbeit die folgenden Artefakte zur Spezifikation einer DSPL während des Domain Engineerings verwendet.

- **Variabilitätsmodell:** Ein Modell, welches die zulässigen Produkte der DSPL beschreibt. In der Literatur werden hierfür häufig FODA-Feature-Modelle verwendet, welche auch in dieser Arbeit verwendet werden (vgl. Abschnitt 2.3.3). Ein Beispiel für ein Variabilitätsmodell ist das DCU-Feature-Modell aus Abbildung 2.9.

- **Implementierungsartefakte:** Wiederverwendbare Implementierungsartefakte, die auf Basis einer Feature-Konfiguration zu einem Produkt komponiert werden. Beispiele für wiederverwendbare Implementierungsartefakte sind

die Verhaltenszustandsmaschine der DCU aus Abbildung 2.3 und der daraus generierte Code aus Abbildung 2.7.

- **Rekonfigurations-Engine:** Die Rekonfigurations-Engine ist für die Adaption der DSPL von einem Produkt zu einem anderen Produkt zur Laufzeit verantwortlich. Insbesondere setzt die Rekonfigurations-Engine Implementierungsartefakte auf Basis einer Änderung der aktuellen Feature-Auswahl neu zusammen. Weiterhin spezifiziert die Rekonfigurations-Engine, welche Rekonfigurations-Sequenzen während eines DSPL-Lebenszyklus erlaubt sind, d. h. von welcher Feature-Konfiguration zu welcher anderen Feature-Konfiguration zur Laufzeit gewechselt werden darf. In der Literatur wird die Rekonfigurations-Engine häufig auf Basis eines endlichen, deterministischen Automaten modelliert [DS11, Sal15]; dies werden wir in dieser Arbeit ebenfalls machen. Ein Beispiel für eine Rekonfigurations-Engine ist das Rekonfigurationsmodell der DCU aus Abbildung 2.13.

- **Anwendungskontexte:** Anwendungskontexte beschreiben die Bereiche/Umgebungen, in denen eine DSPL ausgeführt wird. Anwendungskontexte können durch wechselnde Anforderungen des Anwenders oder unterschiedliche Umgebungen, in denen eine DSPL läuft, definiert werden. Eine DSPL wird immer so konfiguriert, dass die DSPL die Anforderungen des aktuellen Anwendungskontextes möglichst gut erfüllen kann. Aus diesem Grund führt ein Wechsel des Anwendungskontextes üblicherweise zu einer Rekonfiguration einer DSPL.

Beim Application Engineering für DSPLs wird zuerst eine initiale Feature-Auswahl getroffen und auf Basis dieser Auswahl das initiale Produkt aus wiederverwendbaren Implementierungsartefakten abgeleitet, wie in Abbildung 5.2 dargestellt ist. Dieser Schritt entspricht dem Application Engineering von SPLs und geschieht statisch. Danach können dynamische Features, abhängig vom aktuellen Anwendungskontext, zur Laufzeit fortlaufend an- und abgewählt werden, d. h. der Wechsel von Anwendungskontexten löst üblicherweise die Rekonfiguration einer DSPL aus. Zwischen welchen Feature-Belegungen rekonfiguriert werden darf, wird durch die Rekonfigurations-Engine beschrieben, die zusätzlich die korrekte Ausführung der Rekonfiguration durchführt.

Wie in der Forschungsherausforderung FH3.1 beschrieben wurde, reichen die aus der Literatur bekannten Techniken und Artefakte zur Beschreibung einer DSPL nicht aus, um sicherheitskritische Systeme wie die DCU zu beschreiben. Sowohl Ansätze zur Spezifikation von DSPLs, die eine einfache Zerlegung der Feature-Menge in statische und dynamische Features vornehmen, als auch Ansätze, die eine feingranularere Unterteilung der Features auf Basis von stufenweiser Konfiguration erlauben, sind unzureichend zur Spezifikation eines realistischen Konfigurationsprozesses einer DCU während des Application Engineering sowie einer sicheren Rekonfiguration zur Laufzeit. Der Grund dafür ist, dass die Ansätze keine detaillierte Spezifikation von zeitlichen Restriktionen bei der Bindung von Features erlauben sowie keine Bindung von Features zu unterschiedlichen Zeiten ermöglichen. In diesem Abschnitt werden wir einen Ansatz vorschlagen, der aus der Literatur bekannte Ansätze zur Spezifikation von DSPLs um die Möglichkeiten erweitert,

zeitliche Restriktionen bei der Bindung von Features detailliert zu spezifizieren und Features zu unterschiedlichen Bindungszeiten zu konfigurieren. Dafür werden im nächsten Abschnitt erweiterte Feature-Modelle beschrieben, die zum einen nicht-Boole'sche Features und zum anderen komplexere Restriktionen zwischen Features als Require- und Exclude-Constraints erlauben.

5.1.2 *Erweiterte Feature-Modelle*

In Abschnitt 2.3.3 wurden die elementaren FODA-Feature-Modell-Elemente aus der Literatur beschrieben, um den Konfigurationsraum der DCU-Fallstudie mit Hilfe eines FODA-Feature-Modells zu spezifizieren. Hierbei wurden zwei wichtige Aspekte der DCU-Fallstudie außer Acht gelassen.

- Die DCU-Fallstudie enthält nicht-Boole'sche Konfigurationsoptionen, d. h. Konfigurationsoptionen, die nicht nur angewählt oder abgewählt werden können, sondern beispielsweise die Konfiguration eines ganzzahligen Wertes zwischen 0 und 100 erlauben.
- Den Features der DCU-Fallstudie werden vordefinierte Bindungszeiten in einem stufenweisen Konfigurationsprozess zugewiesen, die festlegen wann ein Feature während der Konfiguration eines Produktes gebunden werden darf. Eine Beschreibung von Bindungszeiten und stufenweiser Konfiguration folgt im nächsten Abschnitt.

Zur Modellierung dieser beiden Aspekte reichen die in Abschnitt 2.3.3 beschriebenen Feature-Modell-Elemente nicht aus. Aus diesem Grund verwenden wir in dieser Arbeit *nicht-Boole'sche Features*. Nicht-Boole'sche Features sind Features, die um zusätzliche getypte Attribute erweitert werden [PNX+11, KOD13]. Feature-Modelle mit nicht-Boole'sche Features werden in der Literatur als erweiterte Feature-Modelle bezeichnet [KOD13], wobei die Möglichkeit Features um Attribute zu erweitern bereits in der Machbarkeitsstudie zur feature-orientierten Domänenanalyse von Kang et al. erwähnt wurde [KCH+90].

> **Beispiel 5.1** (Nicht-Boole'sche Features)**.** Abbildung 5.3 zeigt einen Ausschnitt des erweiterten DCU-Feature-Modells mit nicht-Boole'schen Features, welche ganzzahlige Attribute haben. Beispielsweise hat das Feature *AddRtbDelayTime* ein Attribut *delay*, das eine positionsspezifische Synchronisation um einen Wert zwischen 0 ms und 100 ms verzögert.

In dieser Arbeit beschränken wir uns auf Attribute, die über endliche Wertedomänen getypt sind. Dadurch können erweiterte Feature-Modelle, die Feature-Attribute enthalten, mit Hilfe von Techniken analysiert werden, die bereits erfolgreich zur Analyse von Feature-Modellen verwendet wurden (z. B. SAT-Solver). Als Wertedomänen für Attribute verwerden wir Aufzählungen und ganzzahlige geschlossene Intervalle. Dabei wird ein Attribut a von einem Feature f im Folgenden als $f.a$ bezeichnet. Zur Spezifikation von nicht-Boole'schen Restriktionen zwischen Features und/oder Attributen, erweitern wir die Spezifikationssprache für Feature-Restriktionen gemäß der in Abbildung 5.4 dargestellten Grammatik.

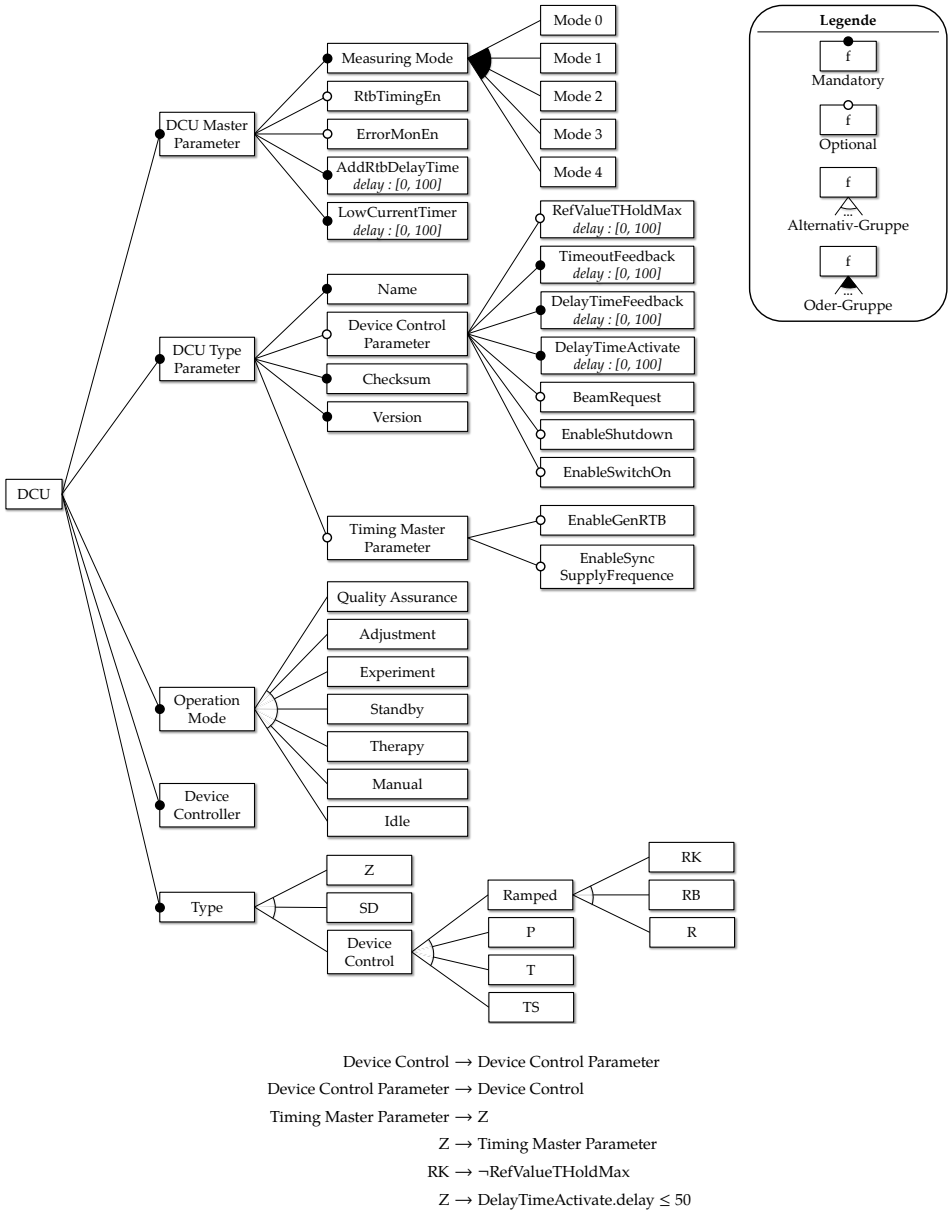


Abbildung 5.3: Ausschnitt des erweiterten Feature-Modells der DCU

Eine Restriktion besteht, gemäß der in Abbildung 5.4 dargestellten Grammatik, aus einem linken und einem rechten Term, wobei der linke Term den rechten Term impliziert oder sich beide Terme gegenseitig implizieren (vgl. Zeile (1)). Den linken Term bezeichnen wir im Folgenden mit *LHS* und den rechten Term mit *RHS*. Ein Term besteht entweder aus einem (negierten) Feature-Term *F* oder einem (negierten) Attribut-Term *A* (vgl. Zeile (2)). Ein Feature-Term *F* besteht aus einem Feature (vgl. Zeile (3)) und ein Attribut-Term *A* besteht aus dem Vergleich zwischen dem Wert eines Feature-Attributes und einer Konstanten oder dem Vergleich zwischen den Werten zweier Feature-Attribute (vgl. Zeile (4)). Für Vergleiche zwischen den Werten von Feature-Attributen und/oder Konstanten werden die Vergleichsoperatoren $=, \neq, \leq, \geq, <$ und $>$ verwendet (vgl. Zeile (5)). In der Grammatik wird nur die Implikation zwischen Termen explizit erlaubt, wodurch Require-Constraints of-

$$
\begin{array}{lll}
C & ::= & T \rightarrow T \mid T \leftrightarrow T \quad (1)\\
T & ::= & F \mid \neg F \mid A \mid \neg A \quad (2)\\
F & ::= & f \quad (3)\\
A & ::= & f.a\ R\ c \mid f.a\ R\ f.a \quad (4)\\
R & ::= & = \mid \neq \mid \leq \mid \geq \mid < \mid > \quad (5)
\end{array}
$$

f: Feature, a: Attribut, c: Konstante

Abbildung 5.4: Grammatik für nicht-Boole'sche Restriktionen zwischen Features und Attributen

fensichtlich spezifiziert werden können. Die Spezifikation von Exclude-Constraints ist nicht explizit, kann jedoch durch das Negieren der linken bzw. rechten Seite einer Implikation erreicht werden [Bat05]. Mit Hilfe der durch die Grammatik in Abbildung 5.4 definierten Restriktionen zwischen Features und Attributen können alle Restriktionen ausgedrückt werden, die in der DCU-Fallstudie enthalten sind.

Beispiel 5.2 (Nicht-Boole'sche Restriktionen)**.** Der Ausschnitt des erweiterten DCU-Feature-Modells aus Abbildung 5.3 enthält eine nicht-Boole'sche Feature-Restriktion zwischen dem Feature *Z* (Timing Master DCU) und dem Attribut *delay* des Features *DelayTimeActivate*, welches die Dauer der Verzögerung der Aktivierung einer DCU definiert. Mit Hilfe des Attributes *DelayTimeActivate.delay* kann im Allgemeinen eine Verzögerung der Aktivierung einer DCU um bis zu 100 ms spezifiziert werden. Für DCUs vom Typ *Z* darf die Verzögerung aus technischen Gründen jedoch maximal 50 ms betragen. Dies wird durch die Restriktion

$$Z \rightarrow DelayTimeActivate.delay \leq 50$$

spezifiziert. Die Spezifikation von Exclude-Constraints mit Hilfe von Negation wird beispielhaft durch die Restriktion

$$RK \rightarrow \neg RefValueTHoldMax$$

zwischen dem DCU-Typ *RK* und dem Feature *RefValueTHoldMax* dargestellt, die sich gegenseitig ausschließen.

Auf Basis des erweiterten Feature-Modells der DCU-Fallstudie können während des Application Engineering die verschiedenen DCUs abgeleitet werden. Dies umfasst das Konfigurieren einer DCU, d. h. die An- und Abwahl von Features und die Konfiguration der Attribute einer DCU, sowie das automatisierte Zusammensetzen von wiederverwendbaren Implementierungsartefakten basierend auf der gewählten Produkt-Konfiguration. Eine DSPL erlaubt dabei das wiederholte Neuzusammensetzen von Implementierungsartefakten zur Laufzeit auf Basis von Änderungen der Produkt-Konfiguration. Die (initiale) An- und Abwahl von Features zu einer Produkt-Konfiguration geschieht dabei nicht für alle Features in einem Schritt, sondern in einem schrittweisen Konfigurationsprozess [CHE04].

Beispiel 5.3 (Schrittweiser Konfigurationsprozess)**.** Insbesondere für Produkte größerer (D)SPLs ist es sinnvoll, die Produkte durch mehrere Experten konfigurieren zu lassen, wobei jeder Experte die Features entsprechend seiner Expertise konfiguriert. Die DCU umfasst beispielsweise verschiedene Aspekte, die von diversen Experten schrittweise konfiguriert werden. Im ersten Schritt konfiguriert ein Experte allgemeine DCU-Parameter, die für jede DCU konfiguriert werden müssen, wie den Typ einer DCU. Darauf aufbauend konfiguriert ein weiterer Experte Details wie das Zeitverhalten einer konkreten DCU. Dabei gibt es zwischen den einzelnen Konfigurationsschritten Abhängigkeiten, die berücksichtigt werden müssen. Beispielsweise ist das von dem zweiten Experten konfigurierte Zeitverhalten abhängig von der Konfiguration der allgemeinen DCU-Parameter, die im ersten Schritt festgelegt wurden (vgl. Beispiel 5.2).

Je umfangreicher eine (D)SPL ist, desto mehr Personen sind an der Entwicklung der (D)SPL und an der Ableitung von Produkten beteiligt. Zur Organisation des Ableitungsprozesses von Produkten bietet sich somit ein schrittweiser Konfigurationsprozess an.

Konfigurationsprozesse werden im Folgenden anhand der Konfiguration von Features beschrieben. Attribute werden im Gegensatz zu Features für eine übersichtlichere und verständlichere Darstellung der folgenden Beschreibungen ausgeklammert, können aber äquivalent zu Features konfiguriert werden. Zur Konfiguration von Features unterscheiden wir zwischen zwei möglichen Konfigurationsoperationen während eines Konfigurationsprozesses.

- Die Operation $\langle + f \rangle$ gibt die Anwahl eines Features $f \in F$ an.
- Die Operation $\langle - f \rangle$ gibt die Abwahl eines Features $f \in F$ an.

Die Menge aller mögliche Konfigurationsoperationen auf einer Feature-Menge F wird im Folgenden als

$$Op_F = \{\langle\, op\, f \rangle \mid op \in \{+, -\} \wedge f \in F\}$$

bezeichnet. Mit

$$F_{cp}^{op} \subseteq F$$

bezeichnen wir die Teilmenge von Features, die in einer Anwahl-/Abwahl-Operation innerhalb eines Konfigurationsprozesses auftritt. Ein Konfigurationsprozess $cp \in Op_F^*$ besteht aus einer Sequenz von Konfigurationsoperationen aus der Menge Op_F, wobei $*$ die Kleensche Hülle ist, d. h. die Menge aller Wörter über der Menge Op_F. In der Literatur gibt es mehrere Vorschläge für die Formalisierung der Semantik von Konfigurationsprozessen [CHH09a, CHH09b, WDSB09, MLSW14]. In dieser Arbeit wird die Notation für die Konfigurationsprozesssemantik von Feature-Modellen, wie oben beschrieben, informell gehalten.

Notation 5.4 (Valider Konfigurationsprozess und Semantik von Konfigurationsprozessen)**.** Sei FM ein Feature-Modell über der Feature-Menge F. Eine Sequenz

$$cp = \langle op_1\, f_1 \rangle, \langle op_2\, f_2 \rangle \ldots \langle op_n\, f_n \rangle \in Op_F^*$$

ist ein valider Konfigurationsprozess des Feature-Modells *FM*, wenn

- jedes Feature $f_i \in F$ in maximal einer Konfigurationsoperation $\langle op_i\, f_i \rangle$ in *cp* verwendet wird, d. h. $F_{cp}^{+} \cap F_{cp}^{-} = \emptyset$ und
- die Teilmenge von Features $F_{cp}^{+} \subseteq F$, die in einer Anwahloperation in *cp* auftritt, zu einer validen Produktkonfiguration erweitert werden kann, d. h. es gibt eine Produktkonfiguration $pc \in [\![FM]\!]$ mit $F_{cp}^{+} \subseteq pc$ und $F_{cp}^{-} \cap pc = \emptyset$.

Ein valider Konfigurationsprozess *cp* ist *vollständig*, wenn $F_{cp}^{+} \cup F_{cp}^{-} = F$ gilt, d. h. für jedes Feature wurde in *cp* eine Konfigurationsentscheidung getroffen. Andernfalls ist die Konfiguration *partiell*. Die Konfigurationsprozesssemantik von *FM*, d. h. die Menge aller validen Konfigurationsprozesse, wird angegeben als $[\![FM]\!]_{cp} \subseteq Op_F^*$.

Aus der Notation 5.4 folgt, dass $F_{cp}^{+} \in [\![FM]\!]_{cp}$ für jeden validen vollständigen Konfigurationsprozess *cp* gilt.

Beispiel 5.5 (Konfigurationsprozess)**.** Für das DCU-Feature-Modell ist die partielle Konfigurationssequenz

$$\langle + \mathit{Type} \rangle, \langle + P \rangle, \langle - T \rangle$$

valide. Die partielle Konfigurationssequenz

$$\langle + \mathit{Type} \rangle, \langle + P \rangle, \langle + T \rangle$$

ist hingegen nicht valide, da die Features *P* und *T* nicht zusammen in einem Produkt ausgewählt werden dürfen, da sie in einer gemeinsamen Alternativ-Gruppe sind und sich somit gegenseitig ausschließen.

Die in Notation 5.4 vorgestellte Konfigurationssemantik erlaubt es, Konfigurationsentscheidungen in einer beliebigen Reihenfolge zu einer validen Sequenz anzuordnen. Beispielsweise kann das mandatory Wurzel-Feature *DCU* des DCU-Feature-Modells aus Abbildung 5.3 als erstes, als letztes oder als Zwischenschritt eines Konfigurationsprozesses gebunden werden.

Ein typischer Konfigurationsprozess einer größeren (D)SPL wird oft in mehrere *Stufen* unterteilt, wobei in jeder Stufe eine bestimmte Teilmenge von Features vollständig konfiguriert wird. Die Einteilung der Konfigurationsschritte in Stufen kann beispielsweise aufgrund verschiedener Anforderungen oder Kompetenzen von Stakeholdern erfolgen, die am Konfigurationsprozess beteiligt sind. Die daraus resultierenden eingeschränkten Konfigurationsprozesse werden in der Literatur als stufenweise Konfigurationen (engl. Staged Configurations) bezeichnet [CHE04]. Für DSPLs kann die Einteilung von Features in Stufen des Konfigurationsprozesses anhand ihrer Bindungszeiten erfolgen, d. h. anhand der Zeitpunkte eines stufenweisen Konfigurationsprozesses zu denen die Features jeweils konfiguriert werden [HHPS08]. Im folgenden Abschnitt wird ein Konfigurationsprozess für DSPLs beschrieben, der stufenweise Konfigurationsprozesse aus der Literatur (z. B. den

Prozess von Hallsteinsen et al. [HHPS08]) um die Möglichkeiten erweitert, Features multiple Bindungzeiten zuzuweisen und komplexe Restriktionen zwischen Bindungszeiten zu spezifizieren.

5.1.3 *Stufenweise Konfiguration von Feature-Modellen*

Beim DSPL-Engineering wird zwischen statischen und dynamischen Bindungszeiten für Features unterschieden.

- **Statische Bindungszeiten:** Features mit statischen Bindungszeiten (auch *statische Features* genannt) bilden unveränderliche Konfigurationsentscheidungen ab, die vor der Zusammensetzung der Implementierungsartefakte beim Application Engineering genau einmal getroffen werden.
- **Dynamische Bindungszeiten:** Features mit dynamischen Bindungszeiten (auch *dynamische Features* genannt) sind für bereits abgeleitete Produktimplementierungen rekonfigurierbar und ermöglichen so vorgeplante Adaptionen eines Produktes über den gesamten Lebenszyklus. Die Rekonfiguration von Features bewirkt die entsprechende Anpassung der Produktimplementierung an die veränderte Feature-Auswahl.

Beispiel 5.6 (Statische und dynamische Bindungszeiten)**.** Die Typ-Features der DCU sind statische Features, die einmal konfiguriert werden und dann während des ganzen Lebenszyklus eines DCU-Produktes unverändert bleiben, da nicht nur die Software einer DCU von ihrem Typ abhängig ist, sondern auch die Hardware. Die *Delay*-Attribute einiger Features zur Konfiguration zeitlicher Verzögerungen während des Behandlungsprozesses sind hingegen vor jeder individuellen Behandlung (re-)konfigurierbar. Ebenso sind die Kind-Features des Features *Operation Mode* bis zu einem gewissen Grad rekonfigurierbar, sodass zwischen verschiedenen Betriebsmodi zur Laufzeit einer DCU gewechselt werden kann. Auf derartige Einschränkungen beim Wechseln zwischen Betriebsmodi und deren Modellierung wird im folgenden Abschnitt eingegangen.

Während des (Re-)Konfigurationsprozesses einer DSPL werden zuerst statische Features und danach dynamische Features konfiguriert. Dabei können die statische und die dynamische Bindungszeit in jeweils mehrere statische und dynamische Stufen unterteilt werden, die auch wieder einer festen, vordefinierten Ordnung unterliegen, wie es beispielsweise zur Modellierung von realen Systeme wie der DCU notwendig ist. Somit unterteilen wir den (Re-)Konfigurationsprozess von DSPLs in mehrere fortlaufende Stufen, die jeweils genau einer Bindungszeit zugeordnet sind und zwischen denen es Abhängigkeiten zwischen den Konfigurationsentscheidungen in den Stufen geben kann. Aus diesem Grund wird der (Re-)Konfigurationsprozess auch als stufenweise (Re-)Konfiguration bezeichnet. Im Folgenden werden Stufe und Bindungszeit synonym verwendet. Weiterhin konzentrieren wir uns in diesem Abschnitt auf die Restriktion von Bindungszeiten während der initialen Konfiguration eines Produktes. Darauf aufbauend wird die Rekonfiguration von Features im nächsten Abschnitt behandelt.

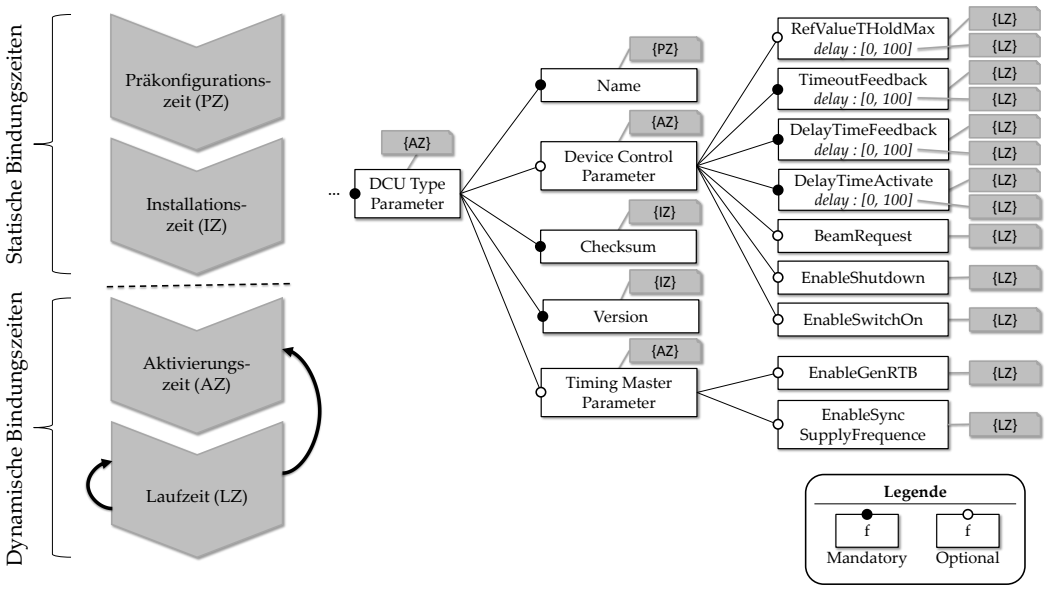


Abbildung 5.5: Ausschnitt aus dem DCU-Feature-Modell mit Bindungszeitannotationen

Beispiel 5.7 (Konfigurationsstufen)**.** Für die DCU-DSPL haben wir vier verschiedene Bindungszeiten, respektive Konfigurationsstufen, identifiziert, wie in Abbildung 5.5 dargestellt ist. Die ersten zwei Stufen sind statischen Bindungszeiten zugeordnet.

- **Präkonfigurationszeit (PZ)**: Die Präkonfigurationszeit umfasst grundsätzliche Konfigurationsentscheidungen, die für jede DCU getroffen werden müssen. Zum Beispiel umfasst die Präkonfigurationszeit die Konfiguration des *Typs* einer DCU.
- **Installationszeit (IZ)**: Die Installationszeit umfasst die Konfiguration der Master-Parameter einer DCU, z. B. den *Measuring Mode*, der die Sensitivität der Messungen von Kommunikationslatenzen angibt.

Im Gegensatz zu den ersten zwei Stufen, sind die anderen zwei Stufen dynamische Stufen.

- **Aktivierungszeit (AZ)**: Während der Aktivierungszeit wird der Betriebsmodus der DCU ausgewählt, z. B. der Betriebsmodus *Therapy* zur Behandlung von Patienten.
- **Laufzeit (LZ)**: Zur Laufzeit werden prozedur- und typspezifische Laufzeitparameter eingestellt. Ein Beispiel hierfür sind die zwei typspezifischen *Timing Master Parameter EnableGenRTBMasterClock* und *EnableSyncSupplyFrequence*.

In Abbildung 5.5 werden die Konfigurationsstufen pfeilartig dargestellt, um die Ordnung der Konfigurationsstufen anzudeuten. Die Ordnung von Konfigurationsstufen gibt dabei explizit eine Priorisierung von Konfigurationsentscheidungen an. Für die DCU ist diese Ordnung total, d. h. bevor in eine neue Konfigurationsstufe

gewechselt wird, muss sichergestellt sein, dass alle Features der vorherigen Stufe gebunden wurden. Dies kann sowohl technische als auch organisatorische Gründe haben. Allerdings kann diese Eigenschaft in manchen Fällen zu einer partiellen Ordnung abgeschwächt werden, z. B. im Falle von Konfigurationsstufen, die nicht voneinander abhängen.

Die Ordnung von Konfigurationsstufen beschränkt die Konfigurationsprozesssemantik eines Feature-Modells auf solche Konfigurationssequenzen *cp*, die der Ordnung entsprechen. Im einfachsten Fall werden bei der Einführung von Bindungszeiten in einem Feature-Modell alle Features einer Menge F in Partitionen $BT_F \subseteq 2^F$ unterteilt, wobei jede Teilmenge $F_S \in BT_F$ die Features darstellt, die einer Konfigurationsstufe S zugeordnet werden. Die totale Ordnung $<_{BT} \subseteq BT \times BT$ definiert die Reihenfolge der Konfigurationsstufen.

Beispiel 5.8 (Ordnung von Konfigurationsstufen)**.** Für die Konfigurationsstufen $BT = \{PZ, IZ, AZ, LZ\}$ der DCU aus Abbildung 5.5 gilt die Ordnung

$$PZ <_{BT} IZ <_{BT} AZ <_{BT} LZ.$$

Mit Hilfe der Ordnung der Konfigurationsstufen können wir die Menge der validen stufenweisen Konfigurationsprozesse einschränken. Intuitiv ist ein valider Konfigurationsprozess *cp* ein *valider stufenweiser Konfigurationsprozess*, wenn wir für jede Konfigurationsoperation $\langle op_j\, f_j \rangle$ in *cp* mit $f_j \in S_j$ und $S_j \in$ BT fordern, dass jedes Feature $f_i \in S_i$, mit $S_i <_{BT} S_j$, in einer Konfigurationsoperation $\langle op_i\, f_i \rangle$ vor jeder Konfigurationsoperation $\langle op_j\, f_j \rangle$ in *cp* ausgeführt wird. Weiterhin schränken wir die Konfigurationsprozesssemantik von Feature-Modellen auf Konfigurationssequenzen ein, die der stufenweisen Konfiguration wie oben beschrieben genügen.

Notation 5.9 (Stufenweise Konfigurationssemantik)**.** Seien *FM* ein Feature-Modell über einer Feature-Menge F und $(BT, <_{BT})$ Konfigurationsstufen über *FM*. Die stufenweise Konfigurationssemantik von *FM* wird mit $[\![FM]\!]_{sc} \subseteq [\![FM]\!]_{cp}$ bezeichnet, wobei $[\![FM]\!]_{cp}$ alle validen Konfigurationsprozesse für *FM* beschreibt. Eine Sequenz von Konfigurationsoperationen $sc \in Op_F^*$ ist eine valide stufenweise Konfiguration gdw. $sc \in [\![FM]\!]_{sc}$ gilt.

Zur vollständigen Spezifikation der DCU als DSPL werden neben der stufenweisen Konfigurationssemantik noch ausdrucksstärkere Modellierungskonzepte benötigt, um Restriktionen innerhalb des stufenweisen Konfigurationsprozesses zu beschreiben, die in der DCU-Fallstudie auftreten. Dies umfasst

1. die Möglichkeit Features und Feature-Attributen *multiple* Bindungszeiten zuzuweisen, d. h. das es möglich ist, Features und Feature-Attribute zu verschiedenen Bindungszeiten zu binden, wobei die Features und Feature-Attribute nur zu einer Bindungszeit tatsächlich gebunden werden, sowie

2. die Spezifikation von Restriktionen zwischen Feature-/Attribut-Konfigurationsentscheidungen und Bindungszeiten.

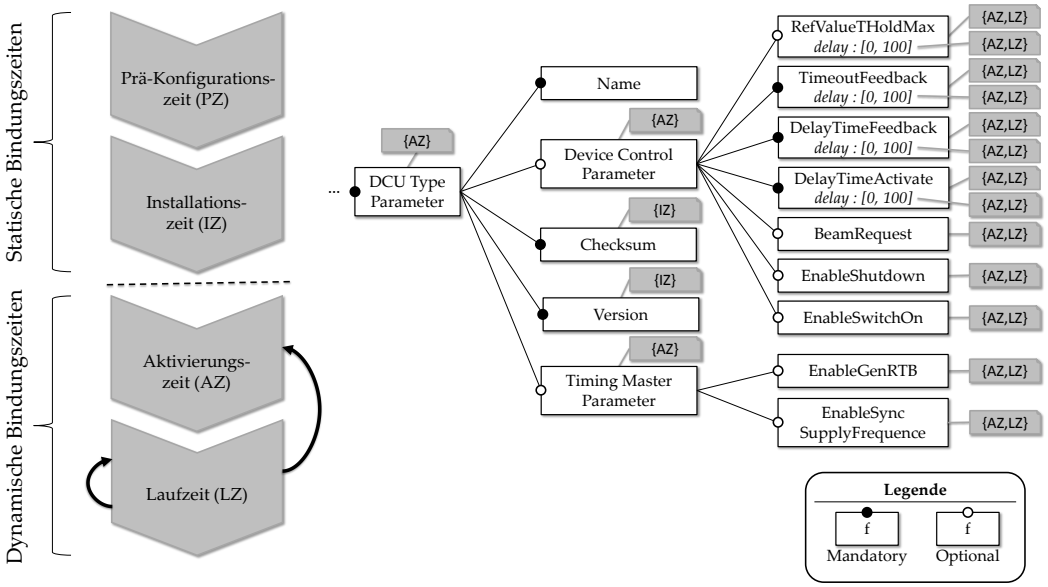


Abbildung 5.6: Ausschnitt aus dem DCU-Feature-Modells mit mehreren Bindungszeiten pro Feature/Attribut

Beispiel 5.10 (Multiple Bindungszeiten pro Feature/Attribut)**.** Das in Abbildung 5.6 dargestellte Feature-Modell verallgemeinert das Feature-Modell aus Abbildung 5.5 dadurch, dass einigen Features und Attributen multiple Bindungszeiten zugewiesen wurden zu denen die Features und Attribute gebunden werden können. Dem Feature *Version* ist immer noch nur eine Bindungszeit zugewiesen und es muss entsprechend zur *Installationszeit* konfiguriert werden. Im Gegensatz dazu, wird das Feature *RefValueTHoldMax* frühestens zur *Aktivierungszeit* gebunden, kann aber erst auch zur *Laufzeit* konfiguriert werden. Für den Fall, dass einem Feature/Attribut keine Bindungszeit zugewiesen wurde, nehmen wir an, dass es zu jeder Bindungszeit konfiguriert werden kann. Beispielsweise hat das Feature *Name* keine explizit aufgelistete Bindungszeit und kann somit entweder zur *Präkonfigurationszeit*, zur *Installationszeit*, zur *Aktivierungszeit* oder zur *Laufzeit* gebunden werden.

Um die stufenweise Konfigurationssemantik aus Notation 5.9 auf multiple Bindungszeiten zu verallgemeinern, kann die Notation $BT_F \subseteq 2^F$ so abgeschwächt werden, dass die Feature-Menge F nicht notwendigerweise in Partitionen unterteilt wird, sondern das jedes Feature $f \in F$ in mindestens einer Konfigurationsstufe $S \in BT$ mit $f \in S$ enthalten sein muss.

Beispiel 5.11 (Stufenweise Konfiguration mit multiplen Bindungszeiten)**.** Das Feature *RefValueTHoldMax* kann entweder zur *Aktivierungszeit* oder zur *Laufzeit* gebunden werden. Die stufenweise Konfigurationssemantik wurde entsprechend angepasst, weshalb Konfigurationsentscheidungen so verschoben werden können, dass $sc \in [\![FM]\!]_{sc}$ gilt, wobei eine Konfigurationsentscheidung

im Fall von *RefValueTHoldMax* spätestens zur *Laufzeit* getroffen werden muss. Daraus folgt auch, dass sowohl die Konfigurationssequenz

$$\langle + \textit{Device Control Parameter}\rangle, \langle + \textit{RefValueTHoldMax}\rangle$$

als auch die Konfigurationssequenz

$$\langle + \textit{RefValueTHoldMax}\rangle, \langle + \textit{Device Control Parameter}\rangle$$

valide ist. Wird jedoch angenommen, dass es noch ein weiteres Feature f mit der Bindungszeit *Laufzeit* gibt, dann ist die Konfigurationssequenz

$$\langle + \textit{RefValueTHoldMax}\rangle, \langle + f\rangle, \langle + \textit{Device Control Parameter}\rangle$$

nicht valide, da *Aktivierungszeit* $<_{BT}$ *Laufzeit* gilt.

Um anzugeben, dass eine Konfigurationsentscheidungen in einer späteren Konfigurationsstufe getroffen wird, werden wir im nächsten Abschnitt eine (nicht sichtbare) *skip*-Operation einführen.

Jede Konfigurationsoperation in einer stufenweisen Konfiguration hat potentiellen Einfluss auf eine nachfolgende Konfigurationsoperation in der gleichen oder in einer späteren Konfigurationsstufe. Ein Beispiel für den Einfluss einer Konfigurationsoperation auf eine nachfolgende Konfigurationsoperation ist, dass ein Feature aufgrund der Konfigurationentscheidung eines anderen Features in einer früheren Konfigurationsstufe in einer späteren Konfigurationsstufe angewählt werden muss. Im Lösungsraum einer DSPL kann es statische Features geben, bei deren Anwahl die zugehörigen Implementierungsartefakte in ein Produkt integriert werden müssen und bei deren Abwahl nicht. Im Gegensatz dazu können Implementierungsartefakte, die mit dynamischen Features assoziiert sind, unter Umständen immer in einem Produkt enthalten sein, z. B. durch Variabilitäts-Encoding mit Präprozessordirektiven [PS08]. Dadurch können die entsprechenden Implementierungsartefakte dynamisch zur Laufzeit (de-)aktiviert werden. Hieraus folgt auch, dass konventionelle Kontrollflussartefakte und Feature-Artefakte auf demselben Abstraktionslevel implementiert werden können, woraus sich logische und temporale Restriktionen zwischen den Artefakten und ihrer (Re-)Konfiguration ergeben können. Aktuelle DSPL-Modellierungsansätze erlauben keine explizite Spezifikation solcher Restriktionen, die Features, Attribute und Bindungszeiten enthalten. Deshalb werden Feature-Modelle in dieser Arbeit um komplexe Bindungszeitrestriktionen erweitert, welche den (Re-)Konfigurationsprozess für sicherheitsrelevante Systeme wie die DCU einschränken.

Beispiel 5.12 (Restriktionen zwischen Bindungszeiten von Features)**.** Die Bindungszeit des Features *Quality Assurance* beeinflusst die Bindungszeit des Features *DelayTimeActivate*, da *DelayTimeActivate* in der gleichen Konfigurationsstufe wie *Quality Assurance* konfiguriert werden muss, um die entsprechenden *RTB*-Ressourcen während der Aktivierung korrekt zu allozieren.

Um Restriktionen wie in Beispiel 5.12 zu spezifizieren, erweitern wir die Grammatik für nicht-Boole'sche Restriktionen zwischen Features und Attributen aus Ab-

C	::=	T → T | T ↔ T	(1)
T	::=	F | ¬F | A | ¬A | B | ¬B	(2)
F	::=	f	(3)
A	::=	f.a R c | f.a R f.a	(4)
B	::=	f.BT R bt | f.a.BT R bt | f.BT R f.BT | f.a.BT R f.a.BT | f.BT R f.a.BT	(5)
R	::=	= | ≠ | ≤ | ≥ | < | >	(6)

f: Feature, a: Attribut, c: Konstante, bt: Bindungszeit

Abbildung 5.7: Grammatik für nicht-Boole'sche Restriktionen zwischen Features, Attributen und Bindungszeiten

Tabelle 5.1: Verschiedene Typen von komplexen Bindungszeitrestriktionen

	Feature/Attribut	Bindungszeit
Feature/Attribut	1. Übliche Feature-/Attribut-Restriktionen	2. Konfigurationsentscheidung schränkt Bindungszeit ein
Bindungszeit	3. Bindungszeit schränkt Konfigurationsentscheidung ein	4. Abhängigkeiten zwischen Bindungszeiten

bildung 5.4 um Bindungszeiten. Dabei werden Bindungszeiten als Attribute von Features und Attributen behandelt, die über eine Aufzählung von Bindungszeiten *bt* getypt sind. Referenziert werden Bindungszeiten mit *f.BT* für ein Feature $f \in F$ bzw. *f.a.BT* für ein Attribut *a* des Features f. Die erweiterte Grammatik ist in Abbildung 5.7 dargestellt. Im Gegensatz zu der Grammatik aus Abbildung 5.4 enthält die Grammatik in Abbildung 5.7 zwei syntaktische Änderungen, um Restriktionen über Bindungszeiten zu spezifizieren. Zum einen kann ein Term T nun auch ein (negierter) Bindungszeitterm B sein (vgl. Zeile (2)). Zum anderen wurde durch Zeile (5) der Aufbau eines Bindungszeitterms ergänzt. Ein Bindungszeitterm besteht entweder aus einem Vergleich zwischen der Bindungszeit eines Features oder eines Attributes mit einer vordefinierten Bindungszeit oder es werden die Bindungszeiten zweier Features und/oder Attribute in Beziehung gesetzt. Vergleiche können mit den Operatoren $=, \neq, \leq, \geq, <$ und $>$ durchgeführt werden (vgl. Zeile (6)).

Auf Basis der Grammatik lassen sich vier Arten von Restriktionen identifizieren, welche Features/Attribute und Bindungszeiten enthalten. Die Arten von Restriktionen sind in Tabelle 5.1 aufgelistet. In der Tabelle 5.1 enthalten Zeilen die linke Seite einer Implikation einer Restriktion und Spalten die rechte Seite. Die Verwendung von Bindungszeitrestriktionen kann viele verschiedene Gründe haben. Zum Beispiel Sicherheitsanforderungen, technische Gründe, sowie rechtliche Anforderungen.

Beispiel 5.13 (Komplexe Bindungszeitrestriktionen)**.** Tabelle 5.1 listet die verschiedenen Typen von komplexen Bindungszeitrestriktionen auf. In der DCU-Fallstudie gibt es für jeden Typ mehrere Beispiele, wovon wir im Folgenden für jeden Typ von komplexen Bindungszeitrestriktionen ein Beispiel beschreiben.

1. Das Feature *Device Control* und somit auch eines seiner Kind-Features erfordert die Konfiguration der entsprechenden typspezifischen Parameter, welche zum Feature *Device Control Parameter* gehören. Dies kann mit der Restriktion

$$\mathit{Device\ Control} \leftrightarrow \mathit{Device\ Control\ Parameter}$$

   beschrieben werden.

2. Der Betriebsmodus *Therapy* erfordert, dass das Feature *EnableSyncSupplyFrequence* zur *Aktivierungszeit* gebunden wird, um vom Patienten Schaden durch eine Adaption während der Bestrahlung abzuhalten. Die entsprechende Restriktion ist

$$\mathit{Therapy} \rightarrow (\mathit{EnableSyncSupplyFrequence.BT} = \mathit{AZ}).$$

3. Die Konfiguration des Features *DelayTimeActivate* zur *Laufzeit* ist nur im Betriebsmodus *Experiment* erlaubt, was durch die Restriktion

$$(\mathit{DelayTimeActivate.BT} = \mathit{LZ}) \rightarrow \mathit{Experiment}$$

   ausgedrückt werden kann.

4. Die Konfiguration des Features *DelayTimeFeedback* zur *Aktivierungszeit* erfordert, dass das Feature *TimeoutFeedback* ebenfalls zur *Aktivierungszeit* konfiguriert wird und umgekehrt. Die entsprechende Restriktion ist

$$(\mathit{DelayTimeFeedback.BT} = \mathit{AZ}) \leftrightarrow (\mathit{TimeoutFeedback.BT} = \mathit{AZ}).$$

Die Einführung von Bindungszeitrestriktionen führt zu einer weiteren Einschränkung der stufenweisen Konfigurationssemantik. Somit müsste, damit ein stufenweiser Konfigurationsprozess $sc \in [\![FM]\!]_{sc}$ valide ist, die stufenweise Konfigurationsprozesssemantik fordern, dass alle Konfigurationsoperationen und ihre Ordnungen in *sc* alle Restriktionen, einschließlich der Bindungszeitrestriktionen, einhalten. Hierfür werden wir in einem späteren Abschnitt eine Modelltransformation vorschlagen, die implizit die stufenweise Konfigurationssemantik mit multiplen Bindungszeiten und komplexen Bindungszeitrestriktionen als SAT-Solving-Problem charakterisiert. Zuerst wird jedoch im folgenden Abschnitt eine Rekonfigurationssemantik für DSPLs beschrieben.

5.1.4 *Rekonfigurationsautomaten*

Wie bereits durch die Pfeile links und rechts der Bindungszeiten in Abbildung 5.6 angedeutet, sind Features mit dynamischen Bindungszeiten während des Lebenszyklus einer DSPL im Gegensatz zu statischen Features rekonfigurierbar. Statische Features werden nur genau einmal zur initialen Produktableitung konfiguriert und danach wird ihre Konfiguration nicht mehr geändert. Beispielsweise wird das statische Feature *Version* einmal zur Installationszeit konfiguriert und ist danach fix,

wohingegen das Feature *DelayTimeFeedback* während der Aktivierungszeit und zur Laufzeit (re-)konfiguriert werden darf.

Der Lebenszyklus eines DSPL-Produktes startet mit einer stufenweisen Konfiguration $sc_0 \in [\![FM]\!]_{sc}$, in der jedes statische und jedes dynamische Feature konfiguriert wird, was zu einer initialen Konfiguration $pc_0 = F^+_{sc} \in [\![FM]\!]_{sc}$ führt. Danach kann die initiale Konfiguration pc_0 zur Konfiguration pc_1 durch eine Sequenz von Rekonfigurationsoperationen $rp_1 \in Op^*_F$ rekonfiguriert werden. Somit enthält die neue Konfiguration

$$pc_1 = (pc_0 \backslash F^-_{rp_1}) \cup F^+_{rp_1}$$

alle Features, die bereits in pc_0 angewählt waren und nicht durch eine Rekonfigurationsoperation aus rp_1 abgewählt wurden, sowie die Features, die in rp_1 neu angewählt wurden. Damit eine Rekonfigurationssequenz rp_1 valide ist, muss die Rekonfigurationssequenz mit der folgenden intuitiven DSPL-Rekonfigurationssemantik übereinstimmen.

1. Die Rekonfiguration endet in einer validen Konfiguration $pc_1 \in [\![FM]\!]_{pc}$.
2. Die Feature-Mengen $F^+_{rp_1}$ und $F^-_{rp_1}$ enthalten ausschließlich dynamische Features.
3. Die Ordnung der Rekonfigurationsoperationen in rp_1 entspricht der Ordnung der (dynamischen) Konfigurationsstufen in $(BT, <_{BT})$.
4. Jedes Feature $f \in F$ taucht in höchstens einer Rekonfigurationsoperation in rp_1 auf und jede Operation führt eine tatsächliche Veränderung einer Konfiguration durch, d. h. $F^+_{rp_1} \cap pc_0 = \emptyset$ und $F^-_{rp_1} \subseteq pc_0$.

Im Gegensatz zu statischen Konfigurationsstufen können dynamische Konfigurationsstufen potentiell beliebig oft durchlaufen werden, wie in Abbildung 5.6 dargestellt ist. Eine Rekonfiguration startet deshalb immer in der ersten Konfigurationsstufe, in der ein Feature $f \in F$ rekonfiguriert wird. Danach verläuft der (Re-)Konfigurationsprozess wie die initiale Konfiguration.

> **Beispiel 5.14** (Beginn einer Rekonfiguration)**.** Ein Rekonfigurationsprozess in dem das Feature *Timing Master Parameter* rekonfiguriert wird, startet mit dem Betreten der Konfigurationsstufe *Aktivierungszeit* gefolgt von der Konfigurationsstufe *Laufzeit*, da *Timing Master Parameter* nur zur *Aktivierungszeit* geändert werden darf.

Für die Rekonfiguration eines Features $f \in F$ mit *multiplen* Bindungszeiten gibt es zwei mögliche Semantiken, die *beständigen* und die *rücksetzbaren* Bindungszeiten.

1. **Beständige Bindungszeiten:** Die Konfigurationsstufe, in der ein Feature f während der initialen Konfiguration gebunden wird, ist die Konfigurationsstufe in der das Feature f auch in allen folgenden Rekonfigurationen während des gesamten Lebenszyklus einer DSPL gebunden werden darf.
2. **Rücksetzbare Bindungszeiten:** Ein Feature f kann während des gesamten Lebenszyklus einer DSPL zu jeder Bindungszeit (re-)konfiguriert werden, die dem Feature zugewiesen wurde.

Beispiel 5.15 (Bindungszeitsemantiken)**.** Der Timing-Master-Parameter *TimeoutFeedback* bekommt, abhängig vom gewählten DCU-Typ, entweder beständige oder rücksetzbare Bindungszeiten zugewiesen. Wie in Abbildung 2.1b zu sehen ist, wird der gesamte Kontrollprozess des Teilchenbeschleunigers unter anderem von DCUs vom Typ *Z* kontrolliert, wohingegen andere DCUs, beispielsweise vom Typ *RK*, zweckgebundene Aufgaben erfüllen. Für DCUs vom Typ *Z* muss das Feature *TimeoutFeedback* deshalb statisch und zu einer beständigen Bindungszeit gebunden werden, um die Korrektheit des gesamten Kontrollprozesses sicherzustellen, wohingegen die Rekonfiguration von *TimeoutFeedback* zur Laufzeit für DCUs vom Typ *RK* unproblematisch ist.

Diese Unterscheidung von Semantiken multipler Bindungszeiten wird zur Spezifikation von DSPLs in vielen Domänen benötigt, z. B. für mobile Geräte [SLR13].

Der Lebenszyklus eines DSPL-Produktes besteht aus aufeinanderfolgenden Adaptionen in Rekonfigurationssequenzen

$$cp_0 \cdot rp_1 \cdot rp_2 \cdots \in Op_F^\omega,$$

wobei Op_F^ω die Menge aller Strings endlicher Länge über der Menge Op_F ist. Im Folgenden wird die stufenweise Konfigurationssemantik um Rekonfiguration erweitert, wobei die bereits beschrieben Eigenschaften weiterhin eingehalten werden.

Notation 5.16 (Stufenweise Rekonfigurationssemantik)**.** Seien *FM* ein Feature-Modell über der Feature-Menge *F* und $(BT, <_{BT})$ die Konfigurationsstufen von *FM*. Ein Rekonfigurationsprozess von *FM* wird als $[\![FM]\!]_{rp} \subseteq Op_F^\omega$ bezeichnet. Eine Rekonfigurationssequenz $rp \in Op_F^\omega$ ist ein valider Rekonfigurationsprozess gdw. $rp \in [\![FM]\!]_{rp}$ gilt.

Ein Rekonfigurationsprozess $rp \in [\![FM]\!]_{rp}$ ist eine Sequenz $pc_0, pc_1, pc_2, \ldots$ von evolvierenden Produktkonfigurationen. Jedoch sind nicht alle Konfigurationssequenzen, die dem Feature-Modell entsprechen, auch valide Konfigurationssequenzen der DSPL. Somit kann es vorkommen, dass ein Feature-Modell durch eine invalide Konfigurationssequenz konfiguriert wird und dies nicht durch die bisher eingeführten Techniken verhindert wird. Gerade in verschiedenen Anwendungskontexten einer DSPL kann das Rekonfigurationsverhalten unterschiedlichen Anforderungen unterworfen sein oder beispielsweise aufgrund von Sicherheitsaspekten eingeschränkt werden.

Beispiel 5.17 (Invalide Konfigurationssequenzen)**.** Aus Sicherheitsgründen ist die Rekonfiguration zwischen Betriebsmodi in bestimmten Anwendungskontexten stark eingeschränkt, um keinem Patienten zu schaden. Eine Therapie wird im *Idle*-Modus begonnen und folgt danach einer vordefinierten Abfolge von Betriebsmodi, die *immer* durchlaufen werden muss. Nach dem *Idle*-Modus wird der *Experiment*-Modus durchlaufen, gefolgt vom *Adjustment*-Modus, um die Parameter der Therapie für den aktuellen Patienten einzustellen und erst danach darf in den *Therapy*-Modus für die eigentliche Bestrahlung gewechselt werden. Im Gegensatz dazu, darf während der Wartung des Systems belie-

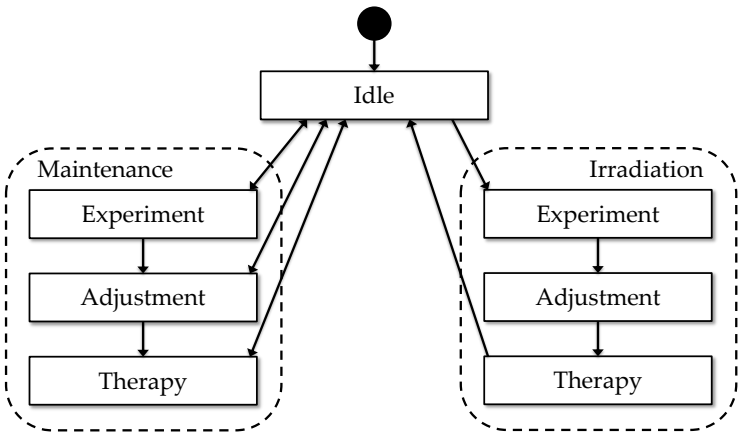


Abbildung 5.8: Ausschnitt des Rekonfigurationsautomaten der DCU-DSPL

big zwischen den einzelnen Betriebsmodi und dem *Idle*-Modus hin und her gewechselt werden.

Um diese Einschränkungen des Rekonfigurationsverhaltens einer DSPL zu spezifizieren, kann zusätzlich zum Feature-Modell, den Konfigurationsstufen und den Bindungszeitrestriktionen ein Rekonfigurationsmodell eingeführt werden, welches die erlaubten Rekonfigurationssequenzen präzise einschränkt. Helvensteijn [Hel12] und Damiani et al. [DS11] verwenden dafür Automaten als Spezifikation, deren Zustände (partielle) Konfigurationen und deren Transitionen erlaubte Rekonfigurationen darstellen. Die erlaubten Rekonfigurationen werden in dem Automaten zusätzlich durch vordefinierte Anwendungskontexte eingeschränkt.

Beispiel 5.18 (Rekonfigurationsautomat)**.** Abbildung 5.8 zeigt einen Ausschnitt des Rekonfigurationsautomaten der DCU-Fallstudie zum Rekonfigurieren der Betriebsmodi. Die Zustände beziehen sich auf die Betriebsmodus-Features der DCU, wobei in einem Zustand nicht aufgelistete Features abgewählt sind. Es sind zwei Anwendungskontexte spezifiziert, *Maintenance* und *Irradiation*. Während einer Bestrahlung (*Irradiation-Kontext*) muss eine feste Abfolge von Betriebsmodi eingehalten werden, um Patienten nicht zu schaden, wohingegen während der Wartung (*Maintenance-Kontext*) zwischen den Betriebsmodi und *Idle* beliebig gewechselt werden darf. Eine Rekonfigurationssequenz $rp \in [\![FM]\!]_{rp}$ ist nur valide, wenn die zugehörigen Konfigurationsschritte $pc_0, pc_1, pc_2, \ldots$ einem Pfad im Rekonfigurationsautomaten entspricht.

Rekonfigurationsautomaten schränken die möglichen Rekonfigurationen auf eine Menge erlaubter Rekonfigurationen ein und restriktieren somit die Rekonfigurationsmöglichkeiten einer DSPL. Die Spezifikation des Rekonfigurationsverhaltens einer DSPL durch Rekonfigurationsautomaten in Kombination mit verschiedenen Bindungszeiten und komplexen Bindungszeitrestriktionen kann für einen Entwickler unüberschaubar werden. Dadurch können unbeabsichtigt Fehler während der Spezifikation des Rekonfigurationsverhaltens einer DSPL gemacht werden, die zu unerwünschten Nebeneffekten wie Deadlocks während der Rekonfiguration führen können.

Beispiel 5.19 (Deadlocks während der Rekonfiguration)**.** Angenommen, das Feature *TimeoutFeedback* wird fehlerhafterweise im *Experiment*-Modus definiert und kann dadurch nicht mehr im *Therapy*-Mode konfiguriert werden. Der Rekonfigurationsprozess in Abbildung 5.8 kann nun stecken bleiben, wenn *TimeoutFeedback* eine beständige statische Bindungszeit erhält. Dieses Beispiel kann noch komplizierter werden, wenn beispielsweise die Bindungszeit von *TimeoutFeedback* Abhängigkeiten zu anderen Bindungszeiten von anderen Features hat.

Um solche (potentiellen) Deadlocks schon während der Spezifikation von DSPLs zu identifizieren, braucht es statische Analyse-Techniken, die über zustandsloses Constraint-Solving hinausgehen, wie beispielsweise Model Checking. Im folgenden Abschnitt 5.2 werden Analyse-Techniken zur statischen Validierung von DSPL-Spezifikationen vorgestellt.

5.2 Validierung dynamischer Software-Produktlinien

Während des SPL-Engineerings dient das Feature-Modell einer SPL einerseits der Modellierung des Variantenraumes und andererseits als Basis für den Konfigurationsprozess von Produkten. Aufgrund dieser zentralen Rolle des Feature-Modells ist eine akkurate Validierung des Feature-Modells essentiell für die Korrektheit der einzelnen Schritte des SPL-Engineerings. Die Validierung umfasst dabei das Überprüfen grundlegender Korrektheitseigenschaften wie *Erfüllbarkeit* und die Abwesenheit von *toten Features*. In der Literatur werden Feature-Modelle zur Analyse üblicherweise in entsprechende Constraint-Solving-Probleme übersetzt [BMC05, HCH09, Bat05, BSMC05].

Der DSPL-Modellierungsansatz aus Abschnitt 5.1 erweitert existierende Feature-Modellierungsansätze um multiple Bindungszeiten und komplexen Bindungszeitrestriktionen sowie um Rekonfigurationsautomaten zur Spezifikation valider Rekonfigurationsprozesse. Zur Berücksichtigung dieser neuen Modellierungskonzepte müssen existierende SPL-Validierungsansätze um die folgenden Aspekte erweitert werden.

1. Eine Erweiterung existierender Validitätseigenschaften für DSPLs. Dies umfasst beispielsweise eine Verallgemeinerung der Notationen von *Erfüllbarkeit* des Feature-Modells und *toter Features* unter Berücksichtigung von Konfigurationsstufen, multiplen Bindungszeiten und komplexen Bindungszeitrestriktionen.

2. Die Validierung neuer Validitätseigenschaften aufgrund der Charakteristiken von DSPLs, die in SPLs nicht auftreten. Dies umfasst im Speziellen die Validierung des Rekonfigurationsprozesses.

In diesem Abschnitt präsentieren wir ein Verfahren zur automatisierten Validierung von DSPL-Spezifikationen, dessen Aufbau in Abbildung 5.9 dargestellt ist. Im oberen Teil von Abbildung 5.9 wird die Validierung von erweiterten Feature-Modellen mit Bindungszeitannotationen dargestellt, die in Abschnitt 5.2.1 beschrieben wird. Hierfür wird das erweiterte Feature-Modell in eine Darstellung transformiert,

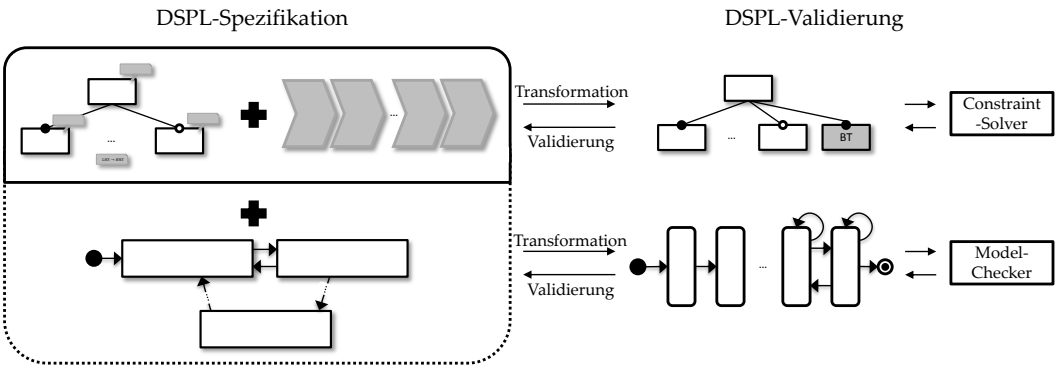


Abbildung 5.9: Automatisierte Validierung einer DSPL-Spezifikation

die ausschließlich FODA-Feature-Modell-Elemente enthält. Diese Darstellung wird im Folgenden *einfaches* Feature-Modell genannt und erlaubt die Verwendung von existierenden Feature-Modell-Analysetechniken auf Basis von Constraint-Solvern zur DSPL-Validierung. Der untere Teil von Abbildung 5.9 zeigt die Validierung der Eigenschaften der (Re-)Konfigurationsprozesse einer DSPL-Spezifikation aus Definition 5.9. Für diese Validierung sind Constraint-Solver nicht ausreichend genug, da temporale Eigenschaften wie Lebendigkeit überprüft werden müssen. Deswegen werden der Rekonfigurationsautomat und die Konfigurationsstufen der DSPL in eine Zustandsmaschine übersetzt, die mit Rekonfigurationsoperationen beschriftet wird. Auf Basis dieser Zustandsmaschine können Eigenschaften des (Re-)Konfigurationsprozesses der DSPL mit Hilfe von Model-Checking-Techniken validiert werden. Dies wird in Abschnitt 5.2.2 beschrieben.

5.2.1 *Validierung von Feature-Modellen mit komplexen Bindungszeitrestriktionen*

Im Folgenden wird eine Modelltransformation von einem erweiterten Feature-Modell mit Bindungszeiten und komplexen Bindungszeitrestriktionen zu einem einfachen Feature-Modell, das ausschließlich FODA-Feature-Modell-Elemente enthält, beschrieben. Die Verwendung des einfachen Feature-Modells erlaubt die Verwendung von Constraint-Solving-Techniken, die in der Literatur häufig für die Analyse von Feature-Modellen verwendet werden. Abbildung 5.10 zeigt den allgemeinen Ablauf der Transformation, der aus drei Schritten besteht:

1. der Transformation von Feature-Attributen in eine Menge von Features als Teil einer Alternativ-Gruppe, wobei jedes Feature einem möglichen Wert entspricht, der dem Feature-Attribut zugewiesen werden kann,

2. der Integration von Bindungszeiten als eine Menge von Features als Teil einer Alternativ-Gruppe für jedes Feature, wobei jedes Feature der Alternativ-Gruppe einer Bindungszeit entspricht, und

3. der Übersetzung von komplexen Restriktionen in Require- und Exclude-Constraints.

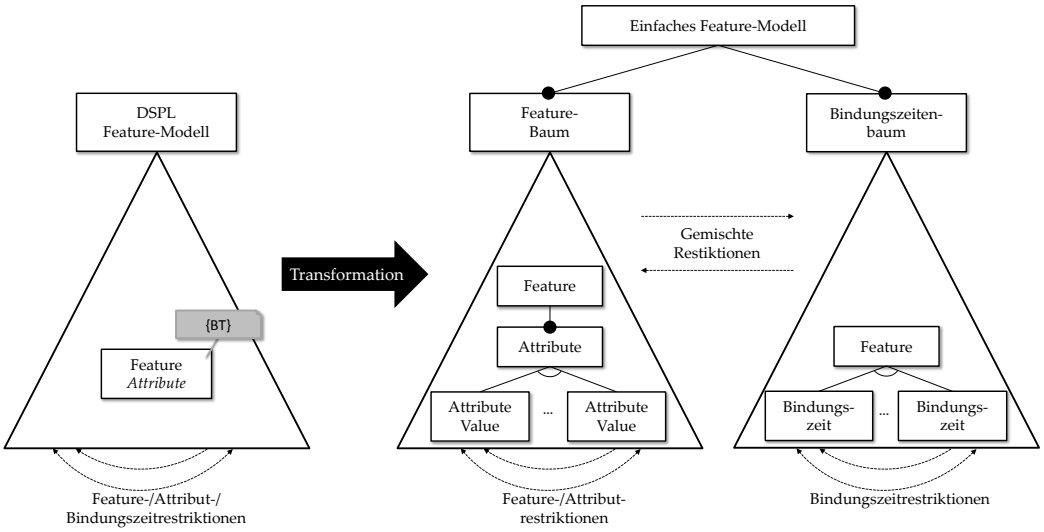


Abbildung 5.10: Transformation eines erweiterten Feature-Modells mit Bindungszeiten in ein einfaches Feature-Modell

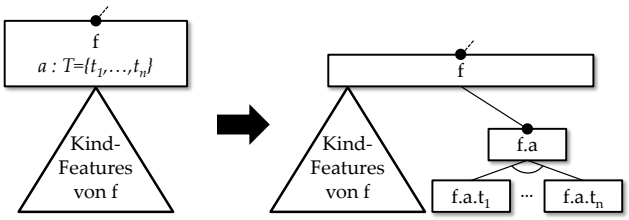
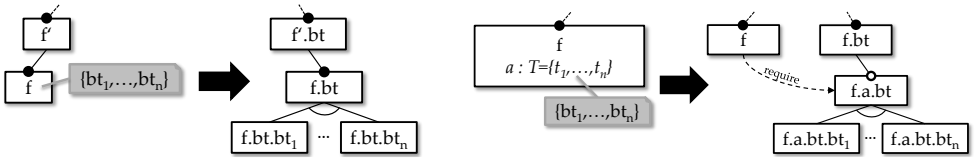


Abbildung 5.11: Transformation von Feature-Attributen

Das einfache Feature-Modell als Ergebnis der Transformation besteht aus zwei Teilbäumen: einem Feature-Baum, der die originalen Features sowie deren Attribute enthält, und einem Bindungszeitenbaum, der die Bindungszeitinformation von Features und Attributen enthält. Die Aufteilung des Feature-Modells in einen Feature-Baum und einen Bindungszeitenbaum hat als einziges Ziel eine bessere Lesbarkeit und ist technisch nicht notwendig.

Transformation von Feature-Attributen. Ein Feature-Attribut $f.a$ eines Typs T ist komplett zum Feature $f \in F$ zugehörig und muss deshalb immer einen Wert $v \in T$ zugewiesen bekommen, wenn f ausgewählt wird. Sollte f abgewählt werden, so werden alle Attribute $f.a$ von f ebenfalls abgewählt. Um dies zu modellieren, wird das Attribut $f.a$ von f als mandatory Kind-Feature von f angesehen. Da wir nur endliche Wertedomänen für Attribute betrachten, können wir alle Werte $v \in T$ in einer Alternativ-Gruppe unter $f.a$ auflisten. Eine Darstellung der Transformation von Attributen in ein einfaches Feature-Modell ist in Abbildung 5.11 zu sehen. Sie wird auf jedes Attribut $f.a_i$ von f angewendet.

Integration von Bindungszeiten. Das Schema der Transformation von Feature-Bindungszeiten ist in Abbildung 5.12a dargestellt. Grundsätzlich wird die



(a) Feature-Bindungszeiten **(b)** Attribut-Bindungszeiten

Abbildung 5.12: Integration von Bindungszeiten

Bindungszeit *f.BT* eines Features *f* wie ein Attribut behandelt, wobei es zwei Unterschiede gibt:

1. Die Werte von *f.BT* werden durch den stufenweisen Konfigurationsprozess gesetzt, abhängig davon in welcher Stufe *f* gebunden wird.

2. Bindungszeiten erhalten *immer* einen Wert, unabhängig davon, ob *f* angewählt wird oder nicht. Dabei gibt die angewählte Bindungszeit an, in welcher Konfigurationsstufe *f* konfiguriert wurde.

Bei der Transformation erweiterter Feature-Modelle in einfache Feature-Modelle wird ein Teilbaum mit Bindungszeiten eingeführt, wie in Abbildung 5.10 dargestellt ist. Innerhalb dieses Baumes wird für jedes Feature *f* ein mandatory Feature *f.BT* eingefügt, das als Bindungszeitparameter für *f* dient. Dabei dupliziert der Bindungszeitenbaum die Hierarchie der Features im Feature-Baum für eine bessere Lesbarkeit. Ähnlich zur Transformation von Feature-Attributen, wird unter das Feature *f.BT* eine Alternativ-Gruppe bestehend aus der Teilmenge der Bindungszeiten gehängt, zu denen *f* gebunden werden darf. Diese Transformation berücksichtigt keine Abhängigkeiten zwischen *f* und *f.BT*, da diese automatisch durch den stufenweisen Konfigurationsprozess erzwungen wird.

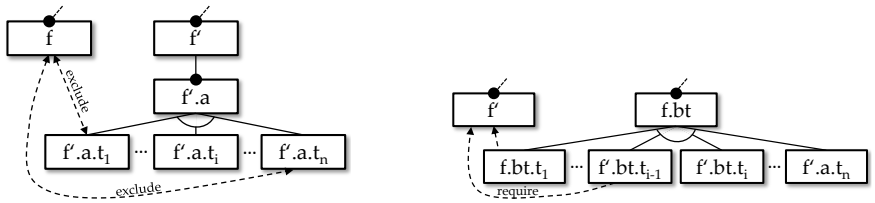
Die Semantik von Attribut-Bindungszeiten unterscheidet sich von der Semantik von Feature-Bindungszeiten aufgrund ihrer engen Beziehung zu ihren Features. Dies ist insbesondere der Fall, wenn ein Feature abgewählt wurde. In dieser Arbeit wird folgende Semantik verwendet, welche die intuitive Semantik von stufenweiser Konfiguration sinnvoll reflektiert.

1. Wenn *f* abgewählt wird bevor *f.a* konfiguriert wurde, hat *f.a* keine Bindungszeit in der endgültigen Konfiguration, da der Wert von *f.a* irrelevant für die endgültige Konfiguration ist, wenn *f* abgewählt wird.

2. Wenn *f* abgewählt wird nachdem *f.a* konfiguriert wurde, hat *f.a* eine Bindungszeit in der endgültigen Konfiguration.

Dies entspricht ebenfalls unserer Annahme, dass Restriktionen bezüglich Attributwerten (und ihrer Bindungszeiten) nur dann erzwungen werden, wenn auch das entsprechende Feature ausgewählt wurde.

Das Schema der Transformation von Attribut-Bindungszeiten gemäß oben beschriebener Semantik ist in Abbildung 5.12b zu sehen. Für jedes Attribut *f.a* eines Features *f* wird ein optionales Bindungszeit-Feature *f.a.BT* mit einer entsprechenden Alternativ-Gruppe, die alle möglichen Bindungszeiten von *f.a* enthält, unter das Feature *f.BT* eingefügt. Zusätzlich wird ein Require-Constraint von *f.a.BT* nach *f* eingefügt, um einen wohldefinierten Bindungszeitwert sicherzustellen.



(a) $f \Rightarrow (f'.a = t_i)$ **(b)** $(f.BT < bt_i) \Rightarrow f'$

Abbildung 5.13: Übersetzung von komplexen Restriktionen

Übersetzung von komplexen Restriktionen. Die Transformationsregeln in Abbildung 5.13 zeigen exemplarisch die Übersetzungsschemata für komplexe Restriktionen. Abbildung 5.13a zeigt den Fall, dass RHS eine Restriktion über Attribut- oder Bindungszeitwerte ist. Zur Übersetzung solcher Restriktionen werden Exclude-Constraints von LHS (z. B. f in Abbildung 5.13a) zu allen ungültigen Werten eingefügt, die durch RHS beschrieben werden. Wir verwenden keine Require-Constraints von f zu den gültigen Werten von f', da sonst die Anwahl von f die Anwahl von f' erzwingen würde, was jedoch nicht der gewünschten Semantik entspricht. In dem Fall, dass RHS aus einem Feature f' besteht, wie in Abbildung 5.13b, werden Require-Constraints von allen gültigen Werten zu f' eingefügt, die durch LHS beschrieben werden (z. B. alle Bindungszeiten von f, die vor bt_i kommen). Alle weiteren Kombinationen dieser Fälle von LHS und RHS werden entsprechend übersetzt.

Eine valide Konfiguration C eines transformierten Feature-Modells umfasst

- eine An-/Abwahlentscheidung für jedes Feature $f \in F$ ($\langle + F \rangle / \langle - F \rangle$) sowie seiner zugehörigen Bindungszeit ($f.BT := bt_i$) und
- für jedes Attribut der angewählten Features, eine Zuweisung eines Attributwertes ($\langle f.a := a.t_i \rangle$) sowie der zugehörigen Bindungszeit ($f.a.BT := bt_j$)

unter Einhaltung der Restriktionen des Feature-Modells. Die Detektion von Feature-Modell-Anomalien, wie unerfüllbare Restriktionen, während des Domain Engineerings ist wichtig, um mögliche Fehler während der Feature-Modell-Spezifikation zu identifizieren [BMC05]. Im Folgenden wird die Definition von Feature-Modell-Anomalien im Kontext von SPLs sowie ihre Detektion für DSPLs und der stufenweisen Konfigurationssemantik angepasst.

Validierung von Bindungszeitrestriktionen. Zwei Kernanforderungen an ein Feature-Modell sind, dass es erfüllbar ist, d. h. es gibt mindestens eine valide Produktkonfiguration, die alle Restriktionen erfüllt, und dass es keine unbeabsichtigten Kern-Features oder tote Features gibt, d. h. jedes Feature ist an- und abwählbar in mindestens einem stufenweisen Konfigurationsprozess. Die Adaption dieser Eigenschaften auf erweiterte Feature-Modelle mit Bindungszeiten erfolgt durch die zusätzliche Berücksichtigung von Bindungszeitrestriktionen. Das Ziel der Feature-Modell-Analyse ist somit der Ausschluss der folgenden unerwünschten DSPL-Spezifikationen:

- **Unerfüllbares Feature-Modell:** Es gibt keine valide Produktkonfiguration, die alle Feature- und Bindungszeitrestriktionen erfüllt, d. h. $[\![FM']\!]_{pc} = \emptyset$.

- **Kern-Features/-Attribute/-Bindungszeiten:** Durch Restriktionen über Features, Attribute und Bindungszeiten ist ein Feature, ein Attributwert oder ein Bindungszeitwert Teil jeder kompletten Produktkonfiguration $pc \in [\![FM']\!]_{pc}$, obwohl er nicht als mandatory definiert wurde.
- **Tote Features/Attribute/Bindungszeiten:** Tote Features, tote Attributwerte und tote Bindungszeitwerte sind niemals Teil einer validen Produktkonfiguration $pc \in [\![FM']\!]_{pc}$, da sie durch Restriktionen ausgeschlossen wurden.

Zum Ausschluss solcher unerwünschten DSPL-Spezifikationen werden Validierungsansätze benötigt, die Bindungszeiten berücksichtigen. Die Erfüllbarkeit des mit der oben beschriebenen Transformation erzeugten einfachen Feature-Modells FM' stellt unter anderem sicher, dass es mindestens einen stufenweisen Konfigurationsprozess $sc \in [\![FM']\!]_{sc}$ gibt, der zu einer validen Produktkonfiguration führt. Dies kann mit der Query

$$FM'$$

an einen Constraint-Solver überprüft werden. Die Übersetzung eines Feature-Modells FM' in einen entsprechenden logischen Ausdruck als Eingabe für einen Constraint-Solver kann anhand der Übersetzungsregeln aus Tabelle 2.1 geschehen.

Entsprechend dazu kann überprüft werden, ob es Kern-Features/-Attribute/-Bindungszeiten bzw. tote Features/Attribute/Bindungszeiten gibt, die immer bzw. niemals angewählt werden können. Beispielsweise kann mit der Query

$$FM' \rightarrow f'$$

überprüft werden, ob das Feature f' ein Kern-Feature ist. Wenn die Query erfüllbar ist, gibt es keine valide Konfiguration in der f' abgewählt ist und somit muss f' in jeder Konfiguration angewählt sein. Diese Query lässt sich ebenfalls zur Detektion von Kern-Attributen und Kern-Bindungszeiten verwenden. Äquivalent dazu prüft die Query

$$FM' \rightarrow \neg f',$$

ob es tote Features/Attribute/Bindungszeiten gibt, d. h. wenn die Query erfüllbar ist, gibt es in diesem Beispiel keine valide Konfiguration, in der f' angewählt ist.

Beispiel 5.20 (Reihenfolge von Konfigurationsentscheidungen)**.** In der DCU-Fallstudie ist die Konfigurationsreihenfolge der Attribute *TimeoutFeedback.timeout* und *DelayTimeFeedback.delay* sehr wichtig, um eine valide Konfiguration zu erhalten. Die Attribute und ihre zugehörigen Features können zwar jeweils zur *Aktivierungszeit* und zur *Laufzeit* gebunden werden, da aber der Wert von *TimeoutFeedback.timeout* von dem Wert von *DelayTimeFeedback.delay* abhängt, muss *DelayTimeFeedback.delay* zuerst gebunden werden.

Die bis hierher beschrieben Validierungstechniken stellen sicher, dass sinnvolle Konfigurationen aus dem DSPL-Feature-Modell abgeleitet werden können. Für die Attribute in Beispiel 5.20 bedeutet das, dass es eine Konfiguration gibt, die durch einen stufenweisen (Re-)Konfigurationsprozess abgeleitet werden kann, in der beide Attribute valide konfiguriert wurden. Jedoch wird damit nicht sichergestellt,

dass in jedem Schritt eines stufenweisen (Re-)Konfigurationsprozesses die logischen und temporalen Abhängigkeiten zwischen Konfigurationsentscheidungen die Möglichkeit offenlassen, dass in den folgenden Stufen noch eine vollständige, valide Konfiguration abgeleitet werden kann, wie in Beispiel 5.20 demonstriert wurde.

Stufenweise Konfiguration. Die Validierung während des Domain-Engineering stellt sicher, dass ein Feature-Modell *FM* erfüllbar ist und es mindestens eine valide, vollständige Konfiguration $pc \in [\![FM]\!]_{pc}$ gibt. Die Konfiguration *pc* wird während des Application-Engineering stufenweise in einer Sequenz von aufeinanderfolgenden partiellen Konfigurationen

$$pc_0, pc_1, \ldots, pc_n$$

vom Feature-Modell *FM* abgeleitet. Jede Konfiguration pc_i, $i = 1, \ldots, n$, verfeinert die vorherige Konfiguration pc_{i-1} entsprechend der Konfigurationsentscheidung, die in diesem Schritt getroffen wurde, wobei die Konfigurationsentscheidung konsistent bzgl. vorheriger Konfigurationsentscheidungen und der Restriktionen des Feature-Modells *FM* ist.

Beispielsweise führen die folgenden Konfigurationsentscheidungen $\langle +\ DCU \rangle$, $\langle +\ Device\ Controller \rangle$, $\langle +\ Type \rangle$, $\langle +\ Z \rangle, \ldots$ zu einer Sequenz von validen, partiellen Konfigurationen $pc_0, pc_1, pc_2, pc_3, \ldots$. Um sicherzustellen, dass diese Sequenz zu einer validen, kompletten Konfiguration $pc_n = pc$ führt, muss nach jedem Konfigurationsschritt überprüft werden, dass die Restriktionen des Feature-Modells eingehalten wurden [HCH09]. Weiterhin muss sichergestellt werden, dass jede Konfigurationsentscheidung der Ordnung der Bindungszeiten entspricht, d. h. dass die Sequenz $pc_0, pc_1, \ldots, pc_n$ so eingeschränkt wird, dass

1. die Sequenz der korrekten Ordnung von Bindungszeiten entspricht und
2. jede Konfigurationsentscheidung in einer der Konfigurationsentscheidung zugeordneten Konfigurationsstufe getroffen wird.

Um dies sicherzustellen, werden Bindungszeiten, in denen ein Feature/Attribut nicht konfiguriert wird, implizit während des stufenweisen Konfigurationsprozesses abgewählt.

> **Beispiel 5.21** (Abwahl von Bindungszeiten)**.** Angenommen die Konfiguration des Features *DCU Master Parameter* ist der erste Konfigurationsschritt pc_i während der *Installationszeit*, dann wird die Menge der verbleibenden Bindungszeiten für alle offenen Konfigurationsentscheidungen durch die Abwahl von der Bindungszeit *Pre-Konfigurationszeit* nach dem Konfigurationsschritt pc_i reduziert.

Zusammenfassend schließen wir auf folgende Korrektheitseigenschaft für die Transformation von DSPL-Spezifikationen in einfache Feature-Modelle unter Berücksichtigung der stufenweisen Konfigurationssemantik:

Eigenschaft 5.22. Seien FM ein Feature-Modell einer DSPL-Spezifikation über der Feature-Menge F und FM' das entsprechende transformierte einfache Feature-Modell. Dann gilt für jede Konfiguration $pc \in [\![FM]\!]_{pc}$ mit $pc' \in [\![FM']\!]_{pc}$ und $pc = (pc' \cap F)$, dass eine valide Konfigurationssequenz $sc \in [\![FM]\!]_{sc}$ mit $F^+_{sc} = pc$ existiert.

Der Beweis dieser Eigenschaft ergibt sich aus der Konstruktion des Transformationsschemas.

5.2.2 *Validierung von Rekonfigurationsprozessen*

Abbildung 5.9 zeigt die notwendigen Schritte für eine Validierung einer DSPL-Spezifikation. Dies umfasst neben der Überprüfung der Abwesenheit von Feature-Modell-Anomalien die Analyse wünschenswerter Eigenschaften des Rekonfigurationsprozesses, welche aus Definition 5.16 abgeleitet werden können. Als Basis für die Validierung eines Rekonfigurationsprozesses dient ein Rekonfigurationsautomat, da dieser das Rekonfigurationsverhalten während des gesamten Lebenszyklus eines DSPL-Produktes spezifiziert (vgl. Abschnitt 5.1.4). Beispielsweise spezifiziert der Rekonfigurationsautomat in Abbildung 5.8 einen Ausschnitt des Rekonfigurationsverhalten der DCU-Fallstudie. In dieser Arbeit wollen wir Rekonfigurationsautomaten gemäß der folgenden vier Eigenschaften validieren:

- **Korrekte Initialisierung:** Es gibt einen stufenweisen Konfigurationsprozess $sc \in [\![FM]\!]_{sc}$, der in einer initialen Konfiguration $pc_0 = F^+_{sc} \in [\![FM]\!]_{pc}$ endet, die konform mit der (partiellen) Konfiguration des initialen Zustands des Rekonfigurationsautomaten ist. Diese Eigenschaft erweitert die Notation der Erfüllbarkeit von Feature-Modellen für stufenweise Konfigurationen. Für den DCU-Rekonfigurationsautomaten wäre das eine Konfiguration, die das Feature *Idle* enthält.

- **Erreichbarkeit:** Jeder Zustand des Rekonfigurationsautomaten soll im Lebenszyklus einer DSPL erreichbar sein. Deswegen fordern wir, dass für jeden Zustand des Rekonfigurationsautomaten ein valider Rekonfigurationsprozess $rp \in [\![FM]\!]_{rp}$ mit einer entsprechenden Sequenz von Konfigurationen $pc_0, pc_1, \ldots, pc_k$ existiert, der einem Pfad im Rekonfigurationsautomaten entspricht und in einer Konfiguration endet, die konform mit dem betrachteten Zustand ist. Diese Eigenschaft erweitert die Notation von toten Features für stufenweise (Re-)Konfigurationen.

- **Fortschritt:** Von jeder Konfiguration $pc_i \in [\![FM]\!]_{pc}$, die über einen Rekonfigurationsprozess $rp \in [\![FM]\!]_{rp}$ mit einer entsprechenden Sequenz von Konfigurationen $pc_0, pc_1, \ldots, pc_i$, der einem Pfad im Rekonfigurationsautomaten entspricht, erreichbar ist, gibt es eine Sequenz rp_{i+1}, sodass $rprp_{i+1} \in [\![FM]\!]_{rp}$ und rp_{i+1} einem nachfolgenden Zustand im Rekonfigurationsautomaten entspricht. Diese Eigenschaft erweitert die Notation von Kern-Features für stufenweise (Re-)Konfigurationen.

- **Lebendigkeit:** Lebendigkeit fordert, dass von jeder Konfiguration, die über eine Rekonfigurationssequenz $rp \in [\![FM]\!]_{rp}$ erreichbar ist und zu einem Pfad im Rekonfigurationsautomaten konform ist, jeder andere Zustand des Rekonfigurationsautomaten über eine nachfolgende Rekonfigurationssequenz erreichbar ist.

Die ersten zwei Validitätseigenschaften beschränken sich auf (Re-)Konfigurationssequenzen endlicher Länge und könnten durch das wiederholte Anwenden der Constraint-Solving-Techniken aus dem vorherigen Abschnitt validiert werden. Die Validierung der letzten beiden Validitätseigenschaften erfordert hingegen das Validieren temporaler Eigenschaften, die aus dem Zusammenspiel von Feature- und Bindungszeitrestriktionen mit Rekonfigurationssequenzen entstehen. Somit ist Constraint-Solving zum Validieren dieser Eigenschaften im Allgemeinen nicht ausreichend, da Konfigurationsentscheidungen aus vorherigen, beliebigen (Re-)Konfigurationsprozessen spätere Konfigurationsschritte beeinflussen.

Solche Eigenschaften können mit Hilfe von Model-Checking verifiziert werden, da Model-Checking automatisierte Verifikationstechniken anbietet, um temporale Eigenschaften auf Verhaltensspezifikationen zu verifizieren [CGP99]. Zur Validierung von DSPL-Spezifikationen mit Hilfe von Model-Checking muss die DSPL-Spezifikation in ein entsprechendes Verhaltensmodell transformiert werden, das die Rekonfigurationssemantik aus Definition 5.16 berücksichtigt. Im Folgenden beschreiben wir ein Übersetzungsschema für eine DSPL-Spezifikation in eine äquivalente Zustandsmaschinenrepräsentation, die als Eingabe für die DSPL-Validierung verwendet werden kann.

Wir beginnen mit der Beschreibung der allgemeinen Struktur der Zustandsmaschine und ihrer Konfigurationsstufen, gefolgt von der Beschreibung der Transformation einzelner Features, ihrer einschränkenden Kontexte im Feature-Modell sowie ihrer Bindungszeitinformationen. Danach reichern wir die Zustandsmaschine um Rekonfigurationsverhalten an. Zur Illustration der Transformation verwenden die DCU-Feature-Modell-Ausschnitte aus Abbildung 5.3 und Abbildung 5.5.

Abbildung 5.14 zeigt im unteren Bereich eine Zustandsmaschine. Es wird zwischen drei Zustandstypen unterschieden: der Initialzustand wird durch einen schwarzen ausgefüllten Kreis dargestellt, der Endzustand wird durch einen schwarzen Kreis dargestellt, der einen ausgefüllten Kreis enthält und allgemeine Zustände werden durch ein Rechteck mit runden Ecken dargestellt. Jeder allgemeine Zustand kann eine Menge von parallelen Sub-Zustandsmaschinen enthalten, die durch Rechtecke mit spitzen Ecken dargestellt werden. Die Zustände der Zustandsmaschine sind durch Transitionen verbunden. Die Transitionsbeschriftungen bestehen aus drei optionalen Komponenten $E[C]/A$ entsprechend der Transitionsbeschriftungen von UML-Zustandsmaschinen, wobei

- E ein Event ist, das auftreten muss, um die Transition auszulösen,
- C eine Bedingung ist, die zusätzlich zum Auftreten des Events erfüllt sein muss, um die Transition auszulösen, und
- A eine Sequenz von atomaren Aktionen ist, welche ausgeführt werden, wenn die Transition ausgelöst wird.

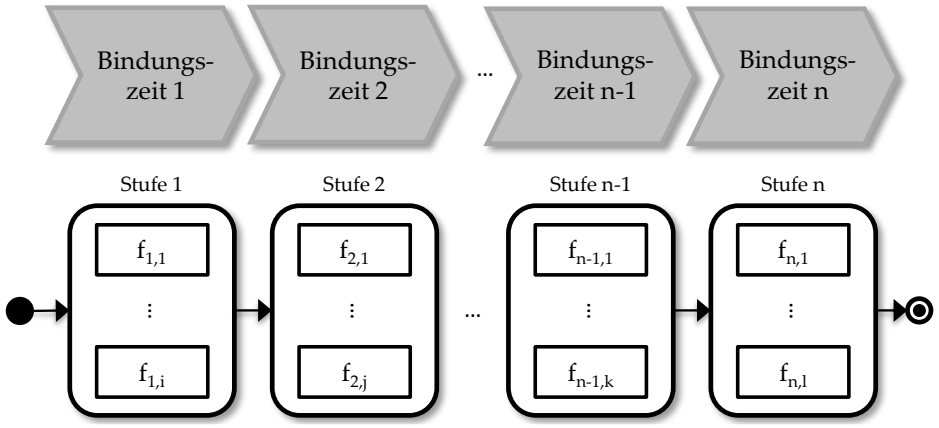


Abbildung 5.14: Transformation von Konfigurationsstufen

In Abbildung 5.14 wurden die Transitionsbeschriftungen weggelassen, da die Transitionsbeschriftungen erst später erläutert werden.

Transformation der Konfigurationsstufen

Abbildung 5.14 zeigt das Transformationsschema für eine DSPL-Spezifikation mit n Bindungszeiten. Jede Bindungszeit unterteilt den Konfigurationsprozess durch die Einführung einer Konfigurationsstufe als Teil der Konfigurationssemantik eines Feature-Modells FM. Dementsprechend wird für jede Bindungszeit $bt_i \in BT, 1 \leq i \leq n$, eine entsprechende Sub-Zustandsmaschine in die Zustandsmaschine eingefügt, die mit *Stufe i* beschriftet ist. Jede Sub-Zustandsmaschine dient als Container für das Encoding der Konfigurationsprozesssemantik der Teilmenge von Features $f_{i,j} \in bt_i, 1 \leq j \leq |bt_i|$, die der Stufe bt_i zugeordnet sind. Die Traversierungsreihenfolge der Sub-Maschinen entspricht der Reihenfolge der Konfigurationsstufen $<_{BT}$, d. h. erst wenn alle Konfigurationsentscheidungen in einer Stufe getroffen wurden, kann die nächste Stufe betreten werden. Weiterhin werden ein Initialzustand, der die erste Konfigurationsstufe markiert, und ein Endzustand, der die letzte Konfigurationsstufe markiert, zur Zustandsmaschine hinzugefügt.

Im nächsten Schritt wird für jedes Feature, das einer Konfigurationsstufe zugeordnet ist, eine weitere Sub-Zustandsmaschine in die Sub-Zustandsmaschine der Konfigurationsstufe eingefügt, welche die Konfigurationsoperationen des Features und entsprechende Restriktionen für das Feature enthält. Die Sub-Zustandsmaschinen zur Konfiguration der Features werden dabei parallel geschaltet. Da ein Feature multiple Bindungszeiten haben kann und es somit auch in verschiedenen Konfigurationsstufen gebunden werden kann, wird eine entsprechende Sub-Zustandsmaschine in jede Sub-Zustandsmaschine jeder Bindungszeit eingefügt, in der das Feature gebunden werden kann. Die Konfigurationslogik wird dabei für jede Konfigurationsstufe individuell angepasst, wobei ein Feature im Allgemeinen in einer Konfigurationsstufe angewählt oder abgewählt werden kann oder die Konfiguration des Features auf eine spätere Konfigurationsstufe verschoben werden kann.

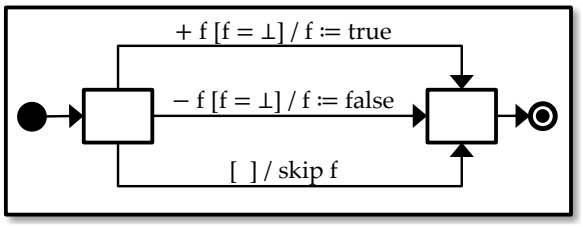


Abbildung 5.15: Transformation von Feature-Konfigurationen in Sub-Zustandsmaschinen

Beispiel 5.23 (Transformation von Konfigurationsstufen)**.** Für die DCU wird ein Skelett mit vier Sub-Zustandsmaschinen, die den vier Bindungszeiten *Präkonfigurationszeit*, *Installationszeit*, *Aktivierungszeit* und *Laufzeit* entsprechen, erstellt. In jeder Sub-Zustandsmaschine jeder Konfigurationsstufe werden parallele Sub-Zustandsmaschinen für die Features angelegt, die in der jeweiligen Konfigurationsstufe gebunden werden können. Beispielsweise wird für das Feature *DCU Type Parameter* eine Sub-Zustandsmaschine in die Sub-Zustandsmaschine für die Konfigurationsstufe *Aktivierungszeit* eingefügt. Da das Feature *RefValueTHoldMax* hingegen zur *Aktivierungszeit* und zur *Laufzeit* konfiguriert werden kann, werden entsprechende Sub-Zustandsmaschinen zur Konfiguration von *RefValueTHoldMax* in die Sub-Zustandsmaschinen beider Konfigurationsstufen eingefügt.

Transformation von Feature-Konfigurationsoperationen

Das Übersetzungsschema für Sub-Zustandsmaschinen für die Konfiguration eines Features f ist in Abbildung 5.15 dargestellt. Die Konfigurationsentscheidung für ein Feature f wird in einer zugehörigen Variablen mit dem gleichen Namen gespeichert. Die Werte dieser Feature-Variablen werden in dreiwertiger Logik angegeben, sodass auch das Aufschieben von Konfigurationsentscheidungen gespeichert werden kann. Somit ist der Wert einer Feature-Variablen während der stufenweisen Konfiguration entweder *true* (t), d. h. f ist angewählt, *false* (f), d. h. f ist abgewählt, oder $\bot$, d. h. es wurde für f noch keine Konfigurationsentscheidung getroffen. Wir nehmen die Auswertungssemantik der dreiwertigen Logikoperatoren an wie üblich und schreiben $f = \top$ für $f = \mathit{true} \vee f = \mathit{false}$. Die Transitionsbeschriftungen in den Sub-Zustandsmaschinen bestehen aus den drei optionalen Komponenten $E[C]/A$ entsprechend der Transitionsbeschriftungen von UML-Zustandsmaschinen. C und A werden über der Menge dreiwertiger Feature-Variablen wie zuvor beschrieben definiert, wohingegen die Elemente in E Konfigurationsoperationen $\langle \mathit{op}\ f \rangle$ entsprechen und deswegen extern ausgelöst werden. Die Entscheidung, ob ein Feature angewählt oder abgewählt wird, wird somit durch ein Event ausgelöst. Abhängig von der Konfigurationsentscheidung, die das Event auslöst, wird die zugehörige Feature-Variable auf *true* oder *false* gesetzt. Die oberste Transition der Sub-Zustandsmaschine in Abbildung 5.15 ist für die Auswahl eines Feature f in einer Konfigurationsstufe und die mittlere für die Abwahl zuständig. Zusätzlich führen wir eine Konfigurationsoperation $\langle \mathit{skip}\ f \rangle$ ein, um die Konfigurationsentscheidung eines Features f auf eine spätere Konfigurationsstufe zu verschieben. In

Tabelle 5.2: An- und Abwahlbedingungen für Features in bestimmten Kontexten

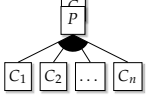
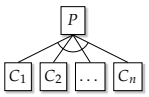
		Anwahlbedingung	Abwahlbedingung
	P	$\neg(C = f)$	$\neg(C = t)$
	C	$\neg(P = f)$	$\neg(P = t)$
	P	–	$\neg(C = t)$
	C	$\neg(P = f)$	–
	P	$\bigvee \neg(C_i = f), i = 1, ..., n$	$\bigwedge \neg(C_i = t), i = 1, ..., n$
	$C_1, \ldots, C_n$	$\neg(P = f)$	$For\ C_i, i = 1, \ldots, n : \neg(P = t) \vee \left(P = t \wedge \left(\bigwedge \neg(C_j = f), j = 1, \ldots, n, j \neq i\right)\right)$
	P	$\bigvee \neg(C_i = f), i = 1, ..., n$	$\bigwedge \neg(C_i = t), i = 1, ..., n$
	$C_1, \ldots, C_n$	$For\ C_i, i = 1, \ldots, n : \neg P = f \wedge \left(\bigwedge \neg(C_j = t), j = 1, \ldots, n, j \neq i\right)$	$For\ C_i, i = 1, \ldots, n : \neg(P = t) \vee \left(P = t \wedge \left(\bigwedge \neg(C_j = f), j = 1, \ldots, n, j \neq i\right)\right)$
	A	$\neg(B = f)$	–
	B	–	$\neg(A = t)$
	A	$\neg(B = t)$	–
	B	$\neg(A = t)$	–

Abbildung 5.15 entspricht die unterste Transition der Skip-Operation, die einerseits dem Aufschieben der Konfiguration von f in eine spätere Stufe dient und andererseits dem Überspringen der Konfiguration von f, falls f bereits in einer früheren Stufe konfiguriert wurde. Das Aufschieben einer Konfigurationsentscheidung darf in der letzten Konfigurationsstufe, in der f konfiguriert werden kann, nicht mehr möglich sein. Um dies sicherzustellen fügen wir der Skip-Transition die Bedingung $[f = \top]$ in der letzten Konfigurationsstufe von f hinzu. In diesem Fall kann die Konfiguration von f nicht übersprungen werden, wenn f noch nicht konfiguriert wurde. Dies ist beispielsweise für die Sub-Zustandsmaschine von *RefValueTHoldMax* in der Stufe *Laufzeit* der Fall. Zusammen mit den Bedingungen der oberen beiden Transitionen wird somit sichergestellt, dass jedes Feature genau einmal im Konfigurationsprozess konfiguriert wird.

Die Konfiguration von Feature-Attributen kann entsprechend übersetzt werden, indem beispielsweise Variablen, die über Integer getypt werden, verwendet werden. Beispielsweise hat *RefValueTHoldMax* ein Integer-Attribut mit einem Wertebereich von 0 bis 100. Hier könnte -1 verwendet werden, um anzuzeigen, dass das Attribut ungebunden ist.

Bisher berücksichtigt die Transformation der DSPL-Spezifikation in eine Zustandsmaschine noch keine Restriktionen zwischen Features gemäß des Feature-Modells und die Zustandsmaschine erlaubt somit invalide Konfigurationssequenzen. Um dem zu entgegnen werden im Folgenden die Anwahl- und Abwahlbedingungen von Features entsprechend angepasst.

Transformation von Feature-Restriktionen

Die verschiedenen Konfigurationsrestriktionsarten eines Feature-Modells, d. h. Eltern-Kind-Beziehungen, Feature-Modalitäten, Feature-Gruppen und Cross-Tree-Constraints, schränken die validen Kombinationen von Features in einer Produkt-

Konfiguration ein. Jedes Feature $f \in F$ hat einen eigenen Kontext in einem Feature-Modell, der unter anderem durch sein Eltern-Feature und seine Geschwister-Features in einer Feature-Gruppe definiert wird und die An- und Abwahlbedingungen von f in einer stufenweisen Konfiguration beeinflusst. Tabelle 5.2 gibt eine Übersicht über die möglichen Kontexte eines Features. Die Kontexte werden im Folgenden dafür genutzt, die Bedingungen an den bereits eingeführten Transitionen für die An- und Abwahl von Features zu verfeinern.

Die erste Regel in Tabelle 5.2 beschreibt die An- und Abwahlbedingungen eines Features P mit einem *mandatory* Kind-Feature C.

Beispiel 5.24 (An-/Abwahlbedingungen für mandatory Features)**.** Das Feature *DCU Type Parameter* hat ein mandatory Kind-Feature *Name*. *DCU Type Parameter* darf nur angewählt werden, wenn *Name* (noch) nicht abgewählt wurde und darf nur abgewählt werden, wenn *Name* (noch) nicht angewählt wurde. Die An- und Abwahl von *Name* funktioniert nach dem gleichen Muster, da *Name* nur ausgewählt werden darf, wenn *DCU Type Parameter* angewählt wird.

Die Bedingungen aus Beispiel 5.24 können mit Hilfe von dreiwertiger Logik ausgedrückt werden, da beispielsweise $\neg(P = f)$ äquivalent zu $(P = t \vee P = \bot)$ ist. Somit ist *DCU Type Parameter* gemäß der ersten Zeile in Tabelle 5.2 nur anwählbar, wenn *Name* bereits angewählt oder noch nicht konfiguriert wurde. Die An- und Abwahl von *Name* funktioniert analog.

Damit *optionale* Features angewählt werden dürfen, müssen ihre Eltern-Features ausgewählt sein, wohingegen es in einer Eltern-Kind-Beziehung keine Bedingung für ihre Abwahl gibt. Das Eltern-Feature eines optionalen Features darf nur abgewählt werden, wenn das optionale Kind-Feature noch nicht abgewählt wurde.

Beispiel 5.25 (An-/Abwahlbedingungen für optionale Features)**.** Das optionale Feature *Device Control Parameter* darf in unserem Beispiel immer abgewählt werden. Im Gegensatz dazu, darf es jedoch nur angewählt werden, wenn sein Eltern-Feature (noch) nicht abgewählt wurde. Entsprechend darf das Feature *DCU Type Parameter* nur abgewählt werden, wenn das Kind-Feature *Device Control Parameter* (noch) nicht angewählt ist. Es gibt jedoch keine Restriktionen für die Anwahl von *DCU Type Parameter*, wohingegen die Abwahl in unserem Beispiel nicht möglich ist, da es ein mandatory Kind-Feature des Wurzel-Features ist.

Eine *Oder-Gruppe* erfordert, dass mindestens ein Gruppen-Feature angewählt wird, wenn das Eltern-Feature der Gruppe ausgewählt wird.

Beispiel 5.26 (An-/Abwahlbedingungen für die Features einer Oder-Gruppe)**.** *MeasuringMode* darf als Eltern-Feature einer Oder-Gruppe nur angewählt werden, wenn mindestens eines seiner Kind-Features (noch) nicht abgewählt wurde, und darf nur abgewählt werden, wenn keines seiner Kind-Features bereits angewählt wurde. Das stellt sicher, dass wenn das Feature *MeasuringMode* angewählt wird, mindestens eines seiner Kind-Features noch angewählt werden kann. Entsprechend dürfen Kind-Features der Oder-Gruppe nur ange-

wählt werden, wenn das Eltern-Feature (noch) nicht abgewählt wurde und die Kind-Features dürfen nur abgewählt werden, wenn das Eltern-Feature (noch) nicht angewählt wurde oder das Eltern-Feature angewählt wurde und mindestens ein Geschwister-Feature noch nicht abgewählt wurde. Diese Konstruktion stellt sicher, dass mindestens ein Gruppen-Feature angewählt werden muss, wenn *MeasuringMode* angewählt ist.

Alternativ-Gruppen fordern, dass genau eine Gruppen-Feature angewählt wird, wenn das Eltern-Feature angewählt wird.

Beispiel 5.27 (An-/Abwahlbedingungen für die Features einer Alternativ-Gruppe)**.** Die An- und Abwahlbedingungen für das Eltern-Feature einer Alternativ-Gruppe sind gleich zu denen des Eltern-Features einer Order-Gruppe. Dementsprechend darf *Type* als Eltern-Feature einer Alternativ-Gruppe nur angewählt werden, wenn mindestens eines seiner Kind-Features (noch) nicht abgewählt wurde, und darf nur abgewählt werden, wenn keines seiner Kind-Features bereits angewählt wurde. Somit kann mindestens ein Kind-Feature von *Type* noch angewählt werden, wenn *Type* angewählt wurde. Die Anwahlbedingung von Kind-Features unterscheidet sich jedoch bei Oder- und Alternativ-Gruppen. Die Kind-Features von *Type*, z. B. *Z* oder *SD*, dürfen nur angewählt werden, wenn ihr Eltern-Feature (noch) nicht abgewählt wurde und kein anderes Kind-Feature bereits angewählt wurde. Die Abwahlbedingung von Kind-Features ist wieder gleich zu der Abwahlbedingung von Kind-Features einer Oder-Gruppe.

Require-Constraints geben an, dass die Anwahl eines Features die Anwahl eines anderen Features fordert. Das führt zu denselben Bedingungen wie die für die An-/Abwahl von optionalen Features und ihren Eltern-Features.

Beispiel 5.28 (An-/Abwahlbedingungen für die Features eines Require-Constraints)**.** Das Feature *Z* erfordert das Feature *Timing Master Parameter*. Deswegen darf *Z* nur angewählt werden, wenn *Timing Master Parameter* nicht abgewählt wurde, und immer abgewählt werden. *Timing Master Parameter* hingegen darf immer angewählt werden, aber nur abgewählt werden, wenn *Z* nicht angewählt wurde.

Exclude-Constraints zwischen zwei Features spezifizieren, dass Features nicht zusammen in einer Konfiguration angewählt werden können.

Beispiel 5.29 (An-/Abwahlbedingungen für die Features eines Exclude-Constraints)**.** Die Features *RK* und *RefValueTHoldMax* schließen sich gegenseitig aus. Deswegen können die Features nicht angewählt werden, wenn das jeweils andere Feature bereits angewählt wurde. Die Features können jedoch immer abgewählt werden.

Die vorgestellten Regeln sind komponierbar, da die Regeln nur die An- und Abwahl eines Features, unabhängig von Features außerhalb des lokalen Kontextes, einschränken.

Transformation von Bindungszeiten und Bindungszeitrestriktionen

Für die Integration von Bindungszeitrestriktionen führen wir zum einen eine Kontrollvariable *curBt* ein, welche die Bindungszeit der aktuellen Konfigurationsstufe speichert, und zum anderen neue Bindungszeitvariablen *f.BT* für jedes Feature, um zu speichern, zu welcher Bindungszeit das Feature f gebunden wurde.

Im Allgemeinen können Restriktionen Features, Attribute und Bindungszeiten enthalten. In diesem Abschnitt konzentrieren wir uns auf Require-Constraints zwischen Features und Bindungszeiten, wobei komplexe Restriktionen über Attribute und Exclude-Constraints in ähnlicher Weise behandelt werden können. Tabelle 5.1 zeigt die vier Arten von Restriktionen zwischen denen wir unterscheiden, wobei wir *Fall 1*, d. h. die Behandlung von Restriktionen zwischen Features und Attributen, bereits beschrieben haben.

In *Fall 2* schränkt eine Konfigurationsentscheidung eine Bindungszeit ein, d. h. wir haben eine Restriktion gemäß des Musters $f \rightarrow f'.BT <op> bt$ mit $op \in OP, bt \in BT$. In diesem Fall darf f immer abgewählt werden, jedoch nur angewählt werden, wenn

$$(f' = \bot) \vee (f' = \top \wedge f'.BT <op> bt)$$

gilt, d. h. f' ungebunden ist oder in einer Stufe gebunden wurde, welche die Bindungszeitrestriktion $f'.BT <op> bt$ erfüllt. f' darf hingegen nur dann gebunden werden, wenn

$$(f = \bot \vee f = false) \vee (f = true \wedge f'bt <op> bt)$$

gilt, d. h. f ungebunden oder abgewählt ist; oder wenn f bereits angewählt ist und f' zu einer Bindungszeit gebunden wird, welche die gegebene Bindungszeitrestriktion erfüllt.

Beispiel 5.30 (Konfigurationsentscheidung schränkt Bindungszeit ein)**.** Die Restriktion

$$Therapy \rightarrow (EnableSynSupplyFrequence.BT = AT)$$

aus Beispiel 5.13 schränkt die Bindungszeit von *EnableSynSupplyFrequence* auf Basis der Konfigurationsentscheidung von *Therapy* ein. Gemäß der oben beschriebenen Schemata ist die Anwahlbedingung für *Therapy*

$$\begin{aligned}&(EnableSynSupplyFrequence = \bot)\\ \vee\ &(EnableSynSupplyFrequence = \top \wedge EnableSynSupplyFrequence.BT = AT)\end{aligned}$$

und die Bedingung für die Bindung von *EnableSynSupplyFrequence*

$$(Therapy = \bot) \vee (Therapy = false) \vee (Therapy = true \wedge curBt = AT).$$

Fall 3 beschreibt Restriktionen, in denen das Binden eines Features f zu einer bestimmten Bindungszeit bt die Konfigurationsentscheidung eines Features f' einschränkt, d. h. wir haben Restriktionen der Form $f <op> bt \rightarrow f'$ mit $op \in OP, bt \in BT$. Das Feature f darf in diesem Fall nur gebunden werden, wenn

$$(f' = \bot \vee f' = true) \vee (f' = false \wedge \neg(curBT <op> bt))$$

gilt. f' hingegen darf zwar jederzeit angewählt werden, aber nur abgewählt werden, wenn die Bedingung

$$(f = \bot) \vee (f = \top \wedge \neg(f'.BT <op> bt))$$

erfüllt ist.

Beispiel 5.31 (Bindungszeit schränkt Konfigurationsentscheidung ein)**.** Gegeben ist die Restriktion

$$(DelayTimeActivate.BT = LZ) \rightarrow Experiment$$

aus Beispiel 5.13. Für das Einschränken der Bindung von *DelayTimeActivate* verwenden wir die Bedingung

$$(Experiment = \bot) \vee (Experiment = true) \vee (Experiment = false \wedge \neg(curBt = LZ)),$$

wohingegen die Anwahl von *Experiment* durch die Bedingung

$$(DelayTimeActivate = \bot) \vee (DelayTimeActivate = \top \wedge \neg(curBt = LZ))$$

beschränkt wird.

Im letzten Fall, *Fall 4*, schränkt die Bindungszeit eines Features f die Bindungszeit eines anderen Features f' ein, d. h. wir haben Restriktionen der Form $f <op> bt \rightarrow f'.BT <op'> bt'$ mit $op, op' \in OP, bt, bt' \in BT$. Die Bindung von f wird hier durch die folgende Bedingung

$$(f' = \bot) \vee (f' = \top \wedge ((curBt <op> bt \wedge f'.BT <op'> bt') \vee \neg(curBt <op> bt)))$$

eingeschränkt. Im Gegensatz dazu wird die Bindung von f' durch die Restriktion

$$(f = \bot) \vee (f = \top \wedge ((curBt <op'> bt') \vee (\neg(curBt <op'> bt') \wedge \neg(f.BT <op> bt))))$$

eingeschränkt.

Beispiel 5.32 (Bindungszeit schränkt Konfigurationsentscheidung ein)**.** Gegeben ist die Restriktion

$$(DelayTimeFeedback.BT = AT) \leftrightarrow (TimeoutFeedback.BT = AT)$$

aus Beispiel 5.13. Um sicherzustellen, dass *TimeoutFeedback* nicht zur *Aktivierungszeit* gebunden wird, wenn *DelayTimeFeedback* nicht zur *Aktivierungszeit* gebunden wurde, fordern wir als Bindungsbedingung für *DelayTimeFeedback*

$$(TimeoutFeedback = \bot) \vee \Big(TimeoutFeedback = \top \\ \wedge ((curBt = AT \wedge TimeoutFeedback.BT = AT) \vee \neg(curBt = AT))\Big).$$

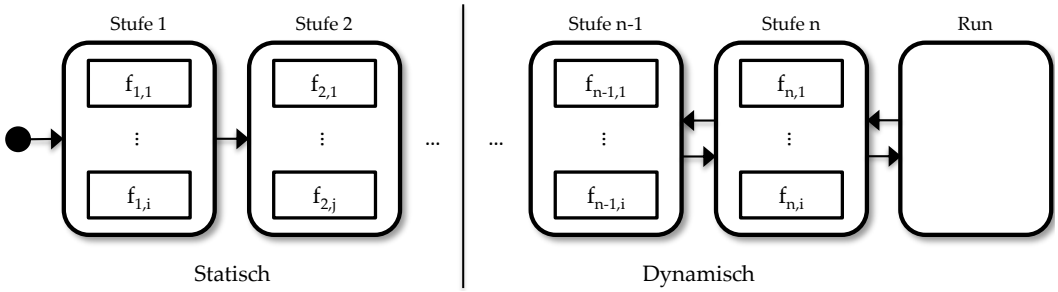


Abbildung 5.16: Zustandsmaschine mit Rekonfigurationstransitionen

Als Bindungsbedingung für *TimeoutFeedback* fordern wir hingegen

$$(DelayTimeFeedback = \bot) \vee \Big(DelayTimeFeedback = \top \\ \wedge ((curBt = AT \wedge DelayTimeFeedback.BT = AT) \vee \neg(curBt = AT))\Big).$$

Anhand der Korrektheit der Konstruktion aus der oben beschriebenen Transformation leiten wir die folgende Eigenschaft ab.

Eigenschaft 5.33. Sei FM ein Feature-Modell über der Feature-Menge F und den Konfigurationsstufen $(BT, <_{BT})$. Dann gilt, dass jede Sequenz $sc \in [\![FM]\!]_{sc}$ zu einem kompletten Lauf der entsprechenden Zustandsmaschine gehört und umgekehrt.

Somit entspricht die Zustandsmaschine exakt der Semantik der (vollständigen) stufenweisen Konfigurationen gemäß des Feature-Modells.

Transformation von Rekonfigurationsoperationen

Der letzte Schritt der Transformation umfasst die Einführung von Rekonfigurationsoperationen durch das Einfügen von *Rekonfigurationstransitionen* für das Zurücksetzen von Konfigurationsentscheidungen. Wie in Abbildung 5.16 dargestellt, werden diese Transitionen jeweils zwischen zwei aufeinanderfolgenden, dynamischen Konfigurationsstufen in umgekehrter Richtung eingefügt. Die Aktionen der Rückwärtstransitionen enthalten eine Operation $f := \bot$, um jedes Feature f aus der Zielkonfigurationsstufe wieder auf ungebunden zu setzen und es somit (re-)konfigurierbar zu machen. Somit kann eine komplette Konfiguration einen Schritt zurück in eine frühere Konfigurationsstufe gehen, um die dort getroffenen Konfigurationsentscheidungen zu ändern. Die Rekonfiguration wird dabei durch ein externes Event ausgelöst, z. B. durch einen Stakeholder. Features mit einer beständigen Bindungszeit können nur in der Konfigurationsstufe rekonfiguriert werden, in der sie zuerst konfiguriert wurden. Deshalb werden Feature mit einer statischen oder beständigen Bindungszeit nicht durch die Aktionen der Rückwärtstransitionen auf ungebunden gesetzt, da die Features nicht rekonfiguriert werden können. Weiterhin ersetzen wir den Endzustand durch einen *Run*-Zustand, der uns im Folgenden dabei hilft zu überprüfen, ob ein Konfigurationsprozess abgeschlossen wurde.

Zusammenfassend formulieren wir die folgende Eigenschaft, deren Korrektheit wieder auf dem unterliegenden Transformationsschema basiert.

Eigenschaft 5.34. Sei FM ein Feature-Modell über der Feature-Menge F und den Konfigurationsstufen $(BT, <_{BT})$. Dann gilt, dass jede Sequenz $rp \in [\![FM]\!]_{rp}$ zu einem vollständigen Lauf der entsprechenden Zustandsmaschine mit Rekonfigurationstransitionen gehört und umgekehrt.

Somit stellt die konstruierte Zustandsmaschine eine valide Basis für die Validierung einer DSPL-Spezifikation dar.

Validierung von DSPL-Spezifikationen

Die Konfigurationssequenzen, die durch die Zustandsmaschine akzeptiert werden, sind genau die, die durch die stufenweise Konfigurationssemantik der DSPL-Spezifikation definiert sind. Aufbauend auf der Zustandsmaschine kann mit Hilfe von Model-Checking die DSPL-Spezifikation bzgl. der vorher definierten Eigenschaften validiert werden. Die Eingabe für den Model-Checker sind dabei die System-Spezifikation sowie eine temporale Eigenschaft, die als LTL-Formel angegeben wird. Auf Basis der Eingabe kann der Model-Checker die Eigenschaft automatisiert für das System validieren.

Um die *korrekte Initialisierung* zu zeigen, muss überprüft werden, dass es einen kompletten stufenweisen Konfigurationsprozess $sc \in [\![FM]\!]_{sc}$ gibt, der zu einer initialen Konfiguration pc_0 führt, die konform mit der (partiellen) Konfiguration pc_{S_0} des Initialzustandes S_0 des Rekonfigurationsautomaten (vgl. Rekonfigurationsautomat der DCU aus Abbildung 5.8) ist. Dies kann mit der LTL-Query

$$\Diamond(Run \wedge pc_{S_0})$$

überprüft werden, wobei $\Diamond$ für *eventually* steht.

Beispiel 5.35 (Korrekte Initialisierung der DCU)**.** Zum Validieren der korrekten Initialisierung der DCU müssen wir überprüfen, ob der Zustand *Run* mit der partiellen Konfiguration *Idle* erreicht werden kann. Dazu verwenden wir die LTL-Query $\Diamond(Run \wedge Idle == true)$.

Für die Validierung der Eigenschaft *Erreichbarkeit* müssen wir für jeden Zustand S des Rekonfigurationsautomaten und seine (partielle) Konfiguration pc_S prüfen, ob *Run* vom Initialzustand S_0 aus erreichbar ist, wobei in *Run* die (partielle) Konfiguration pc_S gelten muss. Ein Template für eine entsprechende LTL-Query, das wir auf jeden Zustand anwenden müssen, ist

$$\Diamond((Run \wedge pc_{S_0}) \wedge X\Diamond(Run \wedge pc_S)),$$

wobei X für *next* steht.

Beispiel 5.36 (Validierung der Erreichbarkeit im DCU-Rekonfigurationsautomaten)**.** Um beispielsweise die Erreichbarkeit von *Adjustment* zu validieren, können wie die LTL-Query

$$\Diamond((Run \wedge Idle == true) \wedge X\Diamond(Run \wedge Adjustment == true)),$$

verwenden.

Zur Validierung der Eigenschaften *Fortschritt* und *Lebendigkeit* der Rekonfigurationsprozesssemantik benötigen wir komplexere Anfragen. Um *Fortschritt* zu überprüfen, müssen wir für jeden Zustand S_1 des Rekonfigurationsautomaten überprüfen, dass immer wenn S_1 erreicht wird, von dort aus jeder andere Zustand S_2 mit $S_1 \neq S_2$ erreicht werden kann. Ein Template für eine LTL-Query ist

$$\Box((Run \wedge pc_{S_1}) \wedge X\Diamond(Run \wedge pc_{S_2})),$$

wobei $\Box$ für *always* steht.

Beispiel 5.37 (Validierung von Fortschritt bzgl. der Rekonfigurationsprozesssemantik)**.** Beispielsweise müssen wir überprüfen, dass immer dann, wenn die partielle Konfiguration *Therapy* erreicht wird, wir durch eine Rekonfiguration zu einer partiellen Konfiguration *Idle* wechseln können. Dies kann durch die LTL-Query

$$\Box((Run \wedge Therapy == true) \wedge X\Diamond(Run \wedge Idle == true)),$$

überprüft werden.

Um *Lebendigkeit* zu überprüfen, muss Fortschritt für alle Zustandspaare in beide Richtungen gezeigt werden, d. h. dass sie von sich gegenseitig aus erreichbar sind.

5.3 Testen dynamischer Software-Produktlinien

DSPLs erweitern SPLs um den Aspekt der Rekonfiguration von Features zur Laufzeit, sodass dynamisch zwischen Produktkonfigurationen gewechselt werden kann. Während der Rekonfiguration einer DSPL von einer Produktkonfiguration zu einer anderen kann es zu Fehlverhalten kommen. Zum Beispiel, wenn sich die DSPL während der Rekonfiguration in einem transienten Zustand befindet, in dem die zur erfolgreichen Rekonfiguration notwendige An- bzw. Abwahl von Features noch nicht vollständig abgeschlossen ist. Weiterhin muss überprüft werden, dass sich eine DSPL nach einer Rekonfiguration gemäß des Verhaltens des Zielproduktes der Rekonfiguration verhält. Somit muss beim Testen von DSPLs nicht nur das Verhalten der einzelnen Produkte, welche die DSPL umfasst, getestet werden, wie es bei SPLs der Fall ist, sondern es muss auch das Rekonfigurationsverhalten von DSPLs getestet werden. In diesem Abschnitt beschreiben wir, wie die Testgenerierungsansätze aus den Kapiteln 3 und 4 auf das Testen von DSPLs angewendet werden können.

5.3.1 *Testen des Verhaltens dynamischer Software-Produktlinien*

Zum Testen des Verhaltens einer DSPL, d. h. des Verhaltens der Produkte einer DSPL, können Test-Suiten familienbasiert, wie in Kapitel 4, generiert werden und auf der Implementierung einer DSPL ausgeführt werden.

Beispiel 5.38 (Testen der Verhaltensspezifikation der DCU)**.** Zum Testen des Verhaltens der DCU-DSPL können Testfälle gemäß der Techniken aus Kapitel 4 generiert werden. Die Ausgangsbasis für die Testgenerierung ist eine Verhaltenszustandsmaschine, die das Verhalten der DCU spezifiziert. Ein Ausschnitt der Verhaltenszustandsmaschine der DCU ist in Abbildung 2.3 dargestellt. Wie in Beispiel 2.5 demonstriert wurde, kann aus der Verhaltenszustandsmaschine automatisiert C-Code erzeugt werden, der als Eingabe für den White-Box Testgenerierungsansatz aus Kapitel 4 verwendet werden kann. Der C-Code, der für den Ausschnitt der Verhaltenszustandsmaschine der DCU erzeugt wurde, ist in Abbildung 2.7 dargestellt. Zur Spezifikation von Abdeckungskriterien, basierend auf den Elementen der Verhaltenszustandsmaschine, können die in den C-Code generierten C-Labels verwendet werden, welche die Elemente der Verhaltenszustandsmaschine referenzieren (vgl. Beispiel 2.5).

Zum Testen des Rekonfigurationsverhaltens der DCU ist das alleinige Berücksichtigen der funktionellen Verhaltensspezifikation der DCU unzureichend. Deshalb muss auch die Spezifikation des Rekonfigurationsverhaltens der DCU in Form des Rekonfigurationsautomaten berücksichtigt werden.

5.3.2 *Testen der Rekonfiguration dynamischer Software-Produktlinien*

In dieser Arbeit werden zwei Ziele beschrieben, die beim Testen des Rekonfigurationsverhaltens einer DSPLs verfolgt werden können:

1. Das Testen des Verhaltens der DSPL während der Rekonfiguration.
2. Das Testen des Verhaltens nach der Rekonfiguration von DSPLs.

Für das erste Ziel wird getestet, ob während der Rekonfiguration einer DSPL von einer Produktkonfiguration zu einer anderen Produktkonfiguration ein Fehlverhalten auftritt. Dies ist insbesondere deshalb relevant, da es während der Rekonfiguration einen Zeitpunkt geben kann, in dem noch nicht alle zu rekonfigurierenden Features an- bzw. abgewählt worden sind und sich eine DSPL dementsprechend in einer invaliden Produktkonfiguration befinden kann. Für das zweite Ziel wird getestet, ob eine Rekonfiguration erfolgreich war, d. h. sich eine DSPL nach der Rekonfiguration entsprechend der Zielproduktkonfiguration der Rekonfiguration verhält.

Zur Berücksichtigung des Rekonfigurationsverhaltens beim Testen ist es notwendig, die erlaubten Rekonfigurationen in den Testgenerierungsprozess mit einzubinden. Als Spezifikation des erlaubten Rekonfigurationsverhaltens dient der Rekonfigurationsautomat einer DSPL. Aus einem Rekonfigurationsautomaten kann,

```
1  ...
2  if (state == Idle) {
3      STATE_IDLE:
4      if (rc == Idle_Experiment_Irradiation) {
5          TRANSITION_IDLE_EXPERIMENT_IRRADIATION:
6          f_Idle = false;
7          f_Experiment = true;
8          f_Irradiation = true;
9          state = Experiment_Irradiation;
10     } else if (rc == Idle_Experiment_Maintenance) {
11         TRANSITION_IDLE_EXPERIMENT_MAINTENANCE:
12         f_Idle = false;
13         f_Experiment = true;
14         f_Maintenance = true;
15         state = Experiment_Maintenance;
16     }
17     ...
18 } else if (state == Experiment_Irradiation) {
19     STATE_EXPERIMENT_IRRADIATION: ...
20 } else if (state == Experiment_Maintenance) {
21     STATE_EXPERIMENT_MAINTENANCE: ...
22 } else if (state == Adjustment_Irradiation) {
23     STATE_ADJUSTMENT_IRRADIATION: ...
24 }
25 ...
```

Abbildung 5.17: Aus dem Rekonfigurationsautomaten der DCU generierter Code

wie aus Verhaltensmodellen, Code generiert werden, der zur Rekonfiguration einer DSPL in die DSPL-Implementierung verwoben werden kann. Dies kann beispielsweise an bestimmten Stellen im Code passieren, sodass sich die Ausführung der Implementierung des Verhaltens einer DSPL mit der Ausführung der Rekonfigurationslogik abwechselt (vgl. Beispiel 2.5). Dadurch wird sichergestellt, dass Rekonfigurationen nur dann ausgeführt werden können, wenn keine kritischen Berechnungen durchgeführt werden, die durch die Rekonfiguration zu fehlerhaftem Verhalten führen können.

Beispiel 5.39 (C-Code für den Rekonfigurationsautomaten der DCU)**.** Abbildung 5.8 zeigt einen Ausschnitt des Rekonfigurationsautomaten der DCU, der das Rekonfigurationsverhalten der DCU spezifiziert. Eine beispielhafte Implementierung, die aus dem Rekonfigurationsautomaten der DCU generiert wurde, ist in Abbildung 5.17 zu sehen. Für jeden Zustand des Rekonfigurationsautomaten wird ein If-Block generiert, der während einer Rekonfiguration ausgeführt wurde, wenn der Zustand gerade aktiv ist (vgl. Zeile 2–18 für den Zustand *Idle*). Für jede erlaubte Rekonfiguration, die in einem Zustand beginnen kann, wird ein entsprechender If-Block generiert, der die zugehörige Rekonfiguration von Features enthält. Beispielsweise enthält der If-Block in den Zeilen 4–10 die Rekonfiguration vom Zustand *Idle* zum Zustand *Experiment* während einer Bestrahlung. Zur Identifikation der Zustände und Transitionen

des Rekonfigurationsautomaten im Code werden wie in Beispiel 2.5 C-Labels verwendet.

Zum Erreichen des ersten Ziels ist es wichtig, dass die erlaubten Rekonfigurationen hinreichend getestet werden, d. h. möglichst alle Rekonfigurationen getestet werden oder mindestens eine repräsentative Auswahl. Um dies sicherzustellen können die Elemente eines Rekonfigurationsautomaten zur Spezifikation des Abdeckungskriteriums verwendet werden, wie es auch für Verhaltensmodellen beim Black-Box-Testen gemacht werden kann (vgl. Beispiel 2.5). So können beispielsweise alle Transitionen des Rekonfigurationsautomaten als Testziele spezifiziert werden, wodurch jede erlaubte Rekonfiguration durch mindestens einen generierten Testfall ausgeführt wird.

Beispiel 5.40 (Rekonfigurationen als Testziele)**.** Zum Testen der erlaubten Rekonfigurationen der DCU können die Rekonfigurationen als Testziele basierend auf dem Rekonfigurationsautomaten der DCU spezifiziert werden. Die Transitionen können dabei durch C-Labels im Code identifiziert werden, der aus dem Rekonfigurationsautomaten der DCU generiert wurde (vgl. Beispiel 5.39). Damit beispielsweise ein Testfall generiert wird, der die Rekonfiguration vom Zustand *Idle* zum Zustand *Experiment* zur Bestrahlung ausführt, kann das C-Label *TRANSITION_IDLE_EXPERIMENT_IRRADIATION* als abzudeckendes Testziel spezifiziert werden. Um eine Test-Suite zu generieren, die alle Transitionen des Rekonfigurationsautomaten ausführt, müssen alle Transitions-Labels im generierten Code durch Testfälle abgedeckt werden.

Das zweite Ziel kann entsprechend erreicht werden, indem jedes Software-Produkt nach einer Rekonfiguration, wie die Software-Produkte in Kapitel 4, getestet wird. Dadurch kann sichergestellt werden, dass jedes Software-Produkt nach einer Rekonfiguration gemäß seiner Spezifikation funktioniert.

Beispiel 5.41 (Verhalten nach einer Rekonfiguration testen)**.** In Abbildung 2.7 ist der Code dargestellt, der aus der Verhaltenszustandsmaschine der DCU generiert wurde. Zum Testen des Verhaltens der DCU nach einer Rekonfiguration, können Testziele erstellt werden, die auf dem Rekonfigurationsautomaten *und* auf der Verhaltenszustandsmaschine spezifiziert werden. Beispielsweise kann ein Testfall generiert werden, der die zwei Testziele *TRANSITION_IDLE_EXPERIMENT_MAINTENANCE* und *TRANSITION_DEV_STOP* nacheinander traversiert. Die Transition *TRANSITION_IDLE_EXPERIMENT_MAINTENANCE* ist dabei eine Transition aus dem Rekonfigurationsautomaten und die Transition *TRANSITION_DEV_STOP* ist eine Transition aus der Verhaltenszustandsmaschine. Getestet wird dabei, ob die DCU nach einer Rekonfiguration vom Zustand *Idle* des Rekonfigurationsautomaten zum Zustand *Experiment* während einer Wartung korrekt gestoppt werden kann, d. h. in den *Idle*-Zustand der Verhaltenszustandsmaschine versetzt werden kann.

5.4 Experimentelle Evaluation

In diesem Abschnitt evaluieren wir die beiden DSPL-Validierungsansätze aus Abschnitt 5.2 anhand der DCU-Fallstudie, d. h. den Constraint-Solver-basierten Ansatz und den Modell-Checker-basierten Ansatz. Zu diesem Zweck haben wir sowohl die Modelltransformation eines erweiterten Feature-Modells mit Bindungszeitrestriktionen in ein einfaches Feature-Modell implementiert (vgl. Abschnitt 5.2.1) als auch die Modelltransformation einer DSPL-Spezifikation in eine Zustandsmaschine (vgl. Abschnitt 5.2.2).

Forschungsfragen

Das Ziel der Evaluation ist es, die vier Forschungsfragen **RQ1.1**, **RQ1.2**, **RQ2.1** und **RQ2.2** zu beantworten. Dabei beziehen sich die ersten zwei Forschungsfragen auf die Analyse von erweiterten Feature-Modellen und dem damit einhergehenden stufenweisen Konfigurationsprozess und die letzten beiden Forschungsfragen auf die Analyse von DSPL-Spezifikationen.

- **RQ1.1 (Effizienz):** Können mit dem Constraint-Solver-basierten Ansatz aus Abschnitt 5.2.1 effizient Anomalien in Feature-Modellen mit Bindungszeiten detektiert werden?
- **RQ1.2 (Effektivität):** Kann der Constraint-Solver-basierte Ansatz aus Abschnitt 5.2.1 zur Validierung von stufenweisen Konfigurationsszenarien angewendet werden?

Die Forschungsfrage **RQ1.1** zielt darauf ab, zu untersuchen, wie effektiv Anomalien in erweiterten Feature-Modellen mit Bindungszeiten detektiert werden können. Da die Anomaliedetektion auf SAT-Anfragen basiert, wird zur Beantwortung der Forschungsfrage **RQ1.1** die abhängige Variable *durchschnittliche CPU-Zeit für eine SAT-Anfrage* verwendet.

Die Forschungsfrage **RQ1.2** zielt auf die Analyse der Ausführungszeiten von SAT-Anfragen bzgl. des Feature-Modells nach jedem Konfigurationsschritt eines stufenweisen Konfigurationsprozesses ab. Somit ist auch hier die abhängige Variable wieder die *durchschnittliche Laufzeit für eine SAT-Anfrage*.

RQ1.1 und **RQ1.2** beschäftigen sich mit der Evaluation des Constraint-Solver-basierten Analyseansatzes für DSPL-Spezifikationen. Im Gegensatz dazu beschäftigen **RQ2.1** und **RQ2.2** mit der Evaluation des Model-Checker-basierten Analyseansatzes.

- **RQ2.1 (Korrekte Initialisierung/Erreichbarkeit):** Können mit dem Modell-Checker-basierten Ansatz aus Abschnitt 5.2.2 die Eigenschaften korrekte Initialisierung und Erreichbarkeit eines Rekonfigurationsautomaten effizient validiert werden?
- **RQ2.2 (Fortschritt/Lebendigkeit):** Können mit dem Modell-Checker-basierten Ansatz aus Abschnitt 5.2.2 die Eigenschaften Fortschritt und Lebendigkeit von Rekonfigurationsszenarien effizient validiert werden?

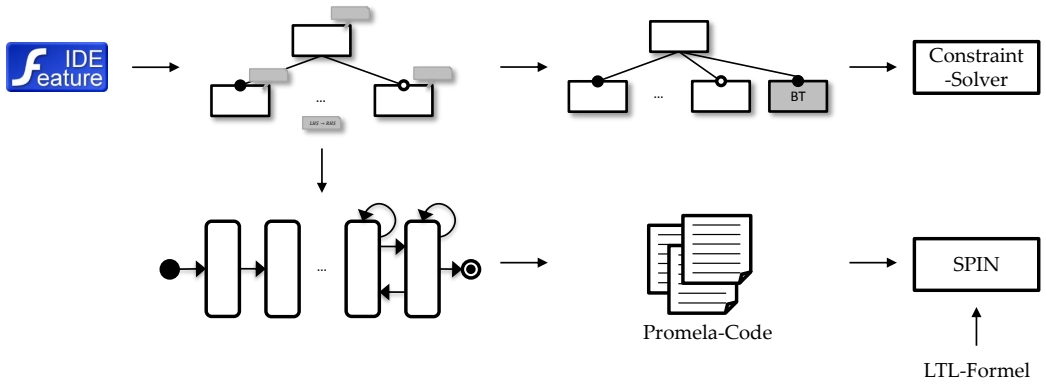


Abbildung 5.18: Architekturdiagramm für die DSPL-Validierungsstrategien

Die Forschungsfragen **RQ2.1** und **RQ2.2** zielen auf die Analyse von Eigenschaften des Rekonfigurationsprozesses einer DSPL-Spezifikation gemäß Abschnitt 5.2.2 ab. Die Analyse der Eigenschaften erfolgt durch Model-Checker-Anfragen. Somit ergibt sich als abhängige Variable zur Beantwortung der Forschungsfragen die *durchschnittliche Laufzeit einer Model-Checker-Anfrage*.

Experimentaufbau

Fallstudien. Für die Evaluation haben wir einen geeigneten Ausschnitt aus dem Feature-Modell der DCU-Fallstudie gewählt, der 47 Features und 55 Restriktionen umfasst, einschließlich Feature- und Bindungszeitrestriktionen. Die DCU-Fallstudie erlaubt es uns, Schlüsse bzgl. der Anwendbarkeit und der Skalierbarkeit unserer Validierungsstrategien für größe Systeme zu ziehen, da die DCU-DSPL weit mehr als 100.000 Produkte umfasst.

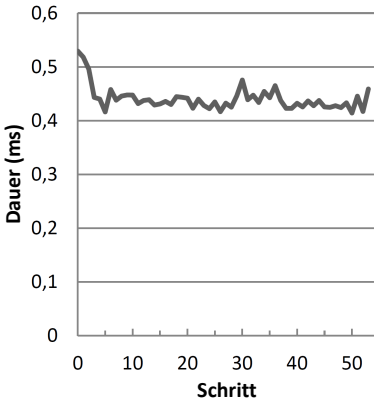
Tool-Unterstützung. Die Abbildung 5.18 zeigt die Architektur der Hauptkomponenten der Evaluationsimplementierung einschließlich der Datenflüsse zwischen diesen Komponenten. Für die Erzeugung und Bearbeitung von Feature-Modellen verwenden wir FeatureIDE[1]. Die Feature-Modelle werden aus dem FeatureIDE-Format in unser eigenes Meta-Modell-Format für die spätere Analyse übertragen. Diese Meta-Modell-Format reichert Feature-Modelle um Feature-Attribute, Bindungszeiten und entsprechende Restriktionen an (vgl. Abschnitt 5.1). Für die Analysen des Feature-Modells (**RQ1.1** und **RQ1.2**) verwenden wir einen effizienten SAT-Solver, der bereits in anderen Arbeiten zu laufzeitadaptiven Systemen, die als DSPL modelliert wurden, zum Einsatz kam [SLR13].

Für die Transformation einer DSPL-Spezifikation in ein Zustandsmaschinenmodell haben wir ebenfalls ein Meta-Modell definiert. Als Model-Checker für die Analyse der Rekonfigurationseigenschaften kam der Model-Checker SPIN[2] zum Einsatz. Da SPIN als Eingabe nur Promela-Code akzeptiert, haben wir eine Transformation von Zustandsmaschinenmodellen in äquivalenten Promela-Code implementiert.

[1] `https://featureide.github.io/`
[2] `http://spinroot.com/`

	FM	Einfaches FM
#Features	47	783
#Restriktionen	55	61
#Kern-Features		6
#Kern-BT		39
#Tote Features		0
#Tote BT		0
SAT-Anfrage		0,46ms

(a) Domain-Engineering-Analyse



(b) Stufenweise Konfiguration

Abbildung 5.19: Ergebnisse der Validitätsüberprüfungen des DCU-Feature-Modells basierend auf SAT-Anfragen

Aufbau der Messungen und Datensammlung. Um Skalierbarkeit zu zeigen, haben wir die benötigte Zeit für die Validierung des (Re-)Konfigurationsprozesses unter Anwendung der verschiedenen Validierungsstrategien gemessen. Dabei haben wir jeden Validierungsschritt 100 mal wiederholt und geben am Ende die Durchschnittszeit aller Validierungsschritte an. Neben der Analysezeit geben wir ebenfalls einige Statistiken zum Feature-Modell der DCU und dem zugehörigen einfachen Feature-Modell an. Die Evaluation wurde auf einem Computer mit einem Intel i5-3230M-Prozessor (2,6GHz) mit 16GB RAM ausgeführt.

Ergebnisse und Diskussion

Bezüglich der Forschungsfragen **RQ1.1** und **RQ1.2** haben wir zuerst den Ausschnitt des DCU-Feature-Modells in ein flaches Feature-Modell transformiert und dann mit Hilfe eines SAT-Solvers die Erfüllbarkeit des flachen Feature-Modells überprüft. Die Ergebnisse sind in der Tabelle in Abbildung 5.19a dargestellt. Als Erstes ist festzustellen, dass das einfache Feature-Modell aufgrund der Transformation von Feature-Attributen und Bindungszeiten sehr viel mehr Features als das originale Feature-Modell enthält. Beispielsweise wird ein Feature-Attribut mit einem Wertebereich von 0 bis 100 in entsprechend viele neue Features im einfachen Feature-Modell umgewandelt.

Bezüglich der Forschungsfrage **RQ1.1** haben wir 6 Kern-Features und 39 Kern-Bindungszeiten gefunden. Dies zeigt, dass es unser Ansatz ermöglicht, Anomalien in einem DSPL-Feature-Modell zu finden. Die durchschnittliche Zeit für eine SAT-Anfrage betrug dabei 0,46ms.

Weiterhin haben wir einen beispielhaften stufenweisen Konfigurationsprozess durchgespielt, um die Forschungsfrage **RQ1.2** zu beantworten. Nach jedem Konfigurationsschritt haben wir die Erfüllbarkeit des teilweise konfigurierten Feature-Modells durch eine SAT-Anfrage geprüft, um mögliche Deadlocks in nachfolgenden Stufen zu detektieren. Die Ergebnisse sind in Abbildung 5.19b dargestellt. Jede SAT-Anfrage dauerte ungefähr 0,45ms und der gesamte Konfigurationsprozess bestehend aus 53 Konfigurationsschritten benötigte 23,5ms. Bei weiteren Experi-

menten haben wir unter anderem künstliche Fehler zufällig über die Menge von Features verteilt und dabei ähnliche Ergebnisse bzgl. der Dauer der SAT-Anfragen erhalten. Zusammenfassend stellen sich die SAT-Anfragen als sinnvolles Mittel zur Validitätsüberprüfung von DSPL-Feature-Modellen in industriellen Kontexten dar.

Zur Beantwortung der Forschungsfragen **RQ2.1** und **RQ2.2** haben wir unseren Ansatz auf den oben genannten DCU-Feature-Modellausschnitt und den Rekonfigurationsautomaten aus Abbildung 5.8 angewendet. Zuerst haben wir mit Hilfe der Transformation aus Abschnitt 5.2.2 aus dem DCU-Feature-Modell ein Zustandsmaschinenmodell abgeleitet, welches die Rekonfigurationsprozesse der DCU-DSPL beschreibt. Danach haben wir Model-Checker-Anfragen zur Überprüfung verschiedener Eigenschaften des Rekonfigurationsprozesses der DCU-DSPL ausgeführt, wie es in Abschnitt 5.2.2 beschrieben wurde.

Bezüglich der Forschungsfrage **RQ1.2** haben wir zuerst die Eigenschaft *korrekte Initialisierung* überprüft, d. h. die Erreichbarkeit des Zustands *Run* im Zustandsmaschinenmodell. Weiterhin haben wir die Erreichbarkeit aller anderen Zustände des Rekonfigurationsautomaten überprüft (*Idle*, *Therapy*, *Experiment* und *Adjustment* sowohl im Therapiemodus als auch im Wartungsmodus). Alle diese Überprüfungen waren erfolgreich und benötigten im Durchschnitt 16,9s, wobei jede Überprüfung zwischen 15,6s und 18,1s benötigte. Der gesamte Prozess dauerte ungefähr 135,2s. Auch hier wurden wieder fehlschlagende Anfragen gestellt, welche ungefähr die gleiche Zeit benötigten.

Die Forschungsfrage **RQ2.2** bezieht sich auf die Eigenschaften Fortschritt und Lebendigkeit von Rekonfigurationsprozessen. Zur Beantwortung dieser Frage evaluierten wir eine schwache Form der Lebendigkeit, indem wir für jeweils zwei Zustände s1 und s2 des Rekonfigurationsautomaten überprüft haben, ob es möglich ist, zuerst den Zustand s1 und dann den Zustand s2 zu erreichen, z. B. der Wechsel zwischen *Adjustment* und *Therapy*. Die daraus resultierenden 42 Anfragen dauerten insgesamt 730s, wobei eine Anfrage im Durchschnitt 17,4s dauert. Eine einzelne Anfrage benötigte dabei zwischen 15,3s und 19,2s. Im Gegensatz dazu hat die Überprüfung von starker Lebendigkeit, wie sie in Abschnitt 5.2.2 beschrieben wurde, keine Ergebnisse geliefert. Der Grund liegt wahrscheinlich in der Größe der Fallstudie.

Die Evaluationsergebnisse zeigen, dass unsere DSPL-Validierungsstrategien in fast allen Punkten auf ein reales Systeme angewendet werden können und skalieren. Nur bei der Überprüfung von strenger Lebendigkeit des DSPL-Spezifikation gab es keine Ergebnisse. Wie bereits erwähnt sind die Ergebnisse verallgemeinerbar auf Systeme, die ähnlich zu unserer Fallstudie sind. Für Aussagen bzgl. einer umfassenderen Verallgemeinerbarkeit müssen weitere Fallstudien betrachtet werden.

Um die strenge Lebendigkeit für DSPL-Spezifikationen zu zeigen, haben wir zwei Ideen. Die erste Idee ist es, die Parametrisierung von SPIN besser an unser Szenario anzupassen oder sogar eine andere Model-Checking-Technologie zu verwenden. Die zweite Idee ist es, das generierte Zustandsmaschinenmodell um weitere Einschränkungen bzgl. des Treffens/Überspringens von Konfigurationsentscheidungen anzureichern. Dies könnte zu einer Verringerung der benötigten Validierungszeit führen.

Gefährdung der Validität

Gefährdung der Konstruktvalidität. Wir haben die DCU als DSPL basierend auf Dokumenten aufbereitet, die uns von der Eckelmann AG zur Verfügung gestellt wurden. Somit könnte die Konstruktvalidität durch unsere Interpretation der Spezifikation der DCU bzw. aufgrund von fehlenden Daten gefährdet sein. Um Fehler bei der Aufbereitung der DCU-Fallstudie zu vermeiden, haben wir die DCU-Systemexperten der Eckelmann AG intensiv interviewt, sowohl vor als auch nach der Aufbereitung der DCU-Fallstudie.

Die Validierung des Rekonfigurationsprozesses wurde mit Hilfe eines Model-Checkers auf Basis von Promela-Code durchgeführt, der aus einem Zustandsmaschinenmodell generiert wurde. Somit ist ein wichtiger Schritt für die Evaluation, dass das Zustandsmaschinenmodell tatsächlich der DSPL-Spezifikation entspricht. Um dies sicherzustellen, haben wir die Transformation ausführlich getestet und bereits erfolgreich in anderen Kontexten wie der Test-Suite-Generierung eingesetzt. Somit sind wir sicher, dass es keine Gefährdung der Konstruktvalidität bzgl. unseres Ansatzes gibt.

Gefährdung der internen Validität. Wir konnten keine Gefährdung der internen Validität aufgrund unserer Fallstudie identifizieren. Um einen Bias bzgl. der Messung der Ausführungszeit unserer Analysen zu vermindern, haben wir sämtlich Messungen 100 mal durchgeführt und dann den Durchschnitt aus allen Messergebnissen berechnet.

Gefährdung der externen Validität. Eine Gefährdung der externen Validität ist, dass nur ein System in der Evaluation betrachtet wurde. Bis zu einem gewissen Grad können unsere Ergebnisse jedoch auf ähnliche Systeme aus unterschiedlichen Domänen verallgemeinert werden. Für eine umfassende Verallgemeinerung müssen die Validierungsansätze jedoch auf weitere Systeme angewendet werden. Unseres Wissens nach sind DSPLs aus der Industrie für Evaluation jedoch nicht verfügbar.

5.5 Verwandete Arbeiten

In diesem Abschnitt diskutieren wir zu den in diesem Kapitel behandelten Ansätzen und Techniken verwandte Arbeiten. Der Fokus liegt dabei auf der Modellierung von DSPLs sowie der Analyse ihrer stufenweisen Konfiguration und Rekonfigurationssemantiken.

Modellierung von DSPLs

In der Literatur wurden bereits verschiedene Ansätze zur Modellierung von DSPLs beschrieben. Initial wurden DSPLs von Hallsteinsen et al. in [HHPS08] eingeführt, wobei die Idee, Features zu verschiedenen Bindungszeiten zu konfigurieren, in der Arbeit von Hallsteinsen et al. bereits beschrieben wurde. Hinchey et al. haben das Konzept von DSPLs in [HPS12] weiter verfeinert und die grundsätzlichen Eigenschaften von DSPLs sowie die Unterschiede zu SPLs beschrieben. Die Brücke

zwischen DSPLs und adaptiven Systemen haben Bencomo et al. geschlagen, indem sie einen Modellierungsansatz für DSPLs entwickelten, der DSPLs als adaptive Systeme betrachtet [BSBG08, BHdA12]. Mei et al. präsentieren in [MZG03] einen feature-orientierten Domänenmodellierungsansatz, der ebenfalls Bindungszeiten berücksichtigt, den Fokus aber auf die Modellierung legt und keine Rekonfiguration einer SPL zulässt.

Zur Modellierung des Rekonfigurationsverhaltens von DSPLs werden in der Literatur häufig Automatenmodelle verwendet, wie es auch in dieser Arbeit geschehen ist. Damiani und Schaefer modellieren das erlaubte Rekonfigurationsverhalten von DSPLs mit Hilfe von Mealy-Automaten, deren Transitionen mit Änderungen an der Implementierung beschriftet sind, die bei einer Rekonfiguration durchgeführt werden müssen [DS11]. Sowohl Helvensteijn als auch Saller et al. verwenden Zustands-Transitions-Graphen zur Modellierung von validen DSPL-Rekonfigurationen, wobei die Transitionen mit Features beschriftet sind, die beim Konfigurationswechsel an- bzw. abgewählt werden [Hel12, SOS+12, SLR13]. Die Zustände repräsentieren dabei partielle Konfigurationen. Das in dieser Arbeit verwendete Rekonfigurationsmodell ist ähnlich zu den Modellen von Helvensteijn und Saller et al., da in unseren Modellen ebenfalls die Zustände partielle Konfigurationen repräsentieren und die Transitionen implizit mit an- bzw. abgewählten Features beschriftet werden. Im Gegensatz zu allen obengenannten DSPL-Modellierungsansätzen bieten wir jedoch die Möglichkeiten an, mit multiplen Bindungszeiten pro Feature/Attribut umzugehen und komplexe Bindungszeitrestriktionen über Features, Attributen und Bindungszeiten zu spezifizieren.

Modellierung und Analyse stufenweiser Konfigurationsprozesse

Als erste haben Czarnecki et al. einen stufenweisen Konfigurationsprozess für Feature-Modelle eingeführt, um schrittweise eine Konfiguration eines Produktes während des Application Engineering zu verfeinern [CHE04]. Zur Definition der verschiedenen Konfigurationsstufen einer SPL werden hauptsächlich *Multi-Views* auf Feature-Modelle [CHE05] oder Bindungszeiten für Features [HHPS08, MZG03] verwendet, wobei sich der erste Ansatz eher auf den Problemraum bezieht und der zweite auf den Lösungsraum.

In der Literatur wurden verschiedene Multi-View-Ansätze vorgeschlagen. Unter anderem zur Modellierung von Concerns [ACLF12, SMM+12] (z. B. organisatorische Strukturen und Implementierungsdetails), zur Modellierung von Konfigurationsabläufen [HCH09, HHSD10, HHS+13] und für Cloud-basierte Anwendungen [SLW12]. Darauf aufbauend wurden formale Semantiken [CHH09b, MLSW14] und entsprechende Analysetechniken [CHH09a, WDSB09, MCdO07] für die stufenweise Konfiguration mit Multi-Views vorgeschlagen. Weiterhin gibt es ein breites Angebot an Werkzeugen für stufenweise Konfiguration mit Multi-Views [AHH11, BNP+07, MMR+15]. Das in dieser Arbeit beschriebene Framework für stufenweise Konfiguration basiert auf den Notationen und Konzepten aus der Literatur und erweitert diese um die Möglichkeit komplexe Bindungszeitrestriktionen zu definieren und zu validieren. Dies erlaubt keiner der genannten Multi-View-Ansätze.

Bezüglich der Spezifikation von stufenweiser Konfiguration mit Hilfe von Bindungszeiten haben Mei et al. in [MZG03] und Svahnberg et al. in [SvGB05] ver-

schiedene Bindungszeiten dokumentiert und kategorisiert. Dahingegen haben sich Rosenmüller et al. hauptsächlich mit Implementierungstechniken für die Unterstützung von statischen und dynamischen Bindungszeiten für Features in DSPLs beschäftigt [RSSA08, RSPA11, RSAS11]. Keiner dieser Ansätze hat jedoch im Detail die Herausforderungen, die aus den Möglichkeiten multiple Bindungszeiten und komplexe Bindungszeitrestriktionen zu spezifizieren hervorgehen, analysiert, wie es in dieser Arbeit getan wurde.

Automatisierte Domänenanalyse

Neuere Ansätze zur Analyse von Variabilitätsmodellen legen häufig den Fokus auf die weit verbreiteten FODA-artigen Feature-Modelle [KCH+90]. Diese Ansätze lassen sich grob in Ansätze basierend auf Constraint-Solver und algebraische Ansätze unterteilen. Ansätze, die Constraint-Solver verwenden, übersetzen Variabilitätsmodelle üblicherweise in verschiedene Arten von Constraint-Problemen zur Validierung der Modelle und zur Detektion von Anomalien. Beispiele hierfür sind die Übersetzung in SAT-Probleme [MWC09, Bat05] oder CSPs [BMC05, BSMC05]. Eine zusammenfassende Übersicht über solche Ansätze kann in [BSRC10] gefunden werden. Weiterhin haben Karataş et al. eine Übersetzung von erweiterten Feature-Modellen auf CSP vorgeschlagen, die, wie unser Ansatz, die Modellierung von Feature-Attribute, welche auf endliche Wertedomänen beschränkt sind, unterstützt [KOD13].

Algebraische Ansätze zur Analyse von Variabilitätsmodellen verwenden Mengenlehre, um die Semantik von Feature-Modellen zu charakterisieren. Zum Beispiel schlagen Schobbens et al. eine generische Formalisierung der Syntax und Semantik von Feature-Diagrammen mit Hilfe von algebraischer Semantiken vor [SHT06, HST+08]. Dabei unterstützen sie jedoch weder erweiterte Feature-Modelle noch Rekonfigurationssemantiken.

Zusammenfassend basiert unser Ansatz auf den Techniken aus dem Bereich der Analysen von Variabilitätsmodellen mit Constraint-Solvern. Jedoch erweitert unser Ansatz die existierenden Ansätze um die Möglichkeit Bindungszeiten sowohl syntaktisch als auch semantisch in Feature-Modelle zu integrieren. Dies wird von keinem der obengenannten Ansätze unterstützt.

5.6 Zusammenfassung und Ausblick

In diesem Kapitel haben wir einerseits einen Ansatz zur Modellierung und automatischen Validierung von DSPLs während der Domänenanalyse vorgestellt und andererseits einen Testgenerierungsansatz für DSPLs. Zur Spezifikation von DSPLs wurden Feature-Modelle um Bindungszeitinformationen erweitert, sodass die Semantik von stufenweisen Rekonfigurationsprozessen unter Berücksichtigung von komplexen logischen und zeitlichen Restriktionen präzise formuliert werden kann. Die Erweiterungen wurden dabei durch die DCU-Fallstudie motiviert, ein sicherheitskritisches, adaptives Kontrollsystem aus der Medizintechnik. Zur Validierung von DSPL-Spezifikationen wurden Ansätze basierend auf Constraint-Solvern und Model-Checkern beschrieben. Unter Verwendung von Constraint-Solvern können dabei Feature-Modell-Anomalien wie Unerfüllbarkeit und unerwünschte Kern-Fea-

tures ausgeschlossen werden. Da mit Hilfe von Constraint-Solvern nicht das (potentiell unendliche) Rekonfigurationsverhalten von DSPLs während des gesamten Produktlebenszyklus analysiert werden kann, haben wir eine Transformation von DSPL-Spezifikationen in Zustandsmaschinen vorgeschlagen, die als Eingabe für einen Model-Checker dienen. Dadurch können alle gewünschten Eigenschaften an den Rekonfigurationsprozess von DSPLs validiert werden, z. B. Fortschritt und Lebendigkeit. Zum Testen des (Rekonfiguration-)Verhaltens von DSPLs wurde weiterhin beschrieben, wie die Testgenerierungsansätze aus den Kapiteln 3 und 4 zur Generierung von Testfällen für DSPLs verwendet werden können.

Die Ergebnisse der Evaluation zeigen, dass die in diesem Kapitel vorgestellten DSPL-Validierungsstrategien in fast allen Punkten auf reale Systeme, die ähnlich zur DCU sind, angewendet werden können und skalieren. Nur bei der Überprüfung von strenger Lebendigkeit des DSPL-Spezifikation gab es keine Ergebnisse, da das zugrundeliegende Model-Checking-Problem ein bekanntes schweres Problem ist. Die erlangten Ergebnisse sind verallgemeinerbar auf Systeme, die ähnlich zu unserer Fallstudie sind. Für Aussagen bzgl. einer umfassenderen Verallgemeinerbarkeit müssen jedoch weitere Fallstudien betrachtet werden.

In zukünftigen Arbeiten ist ein interessanter zu untersuchender Aspekt die Erweiterung der Ausdrucksmächtigkeit der Grammatik für nicht-Boole'sche Restriktionen zwischen Features, Attributen und Bindungszeiten. Hierbei könnten beispielsweise mehr Sprachkonstrukte wie das logische Oder, weitere Wertedomänen oder Berechnung auf Attributwerten unterstützt werden. Ein Beispiel dafür wäre die Restriktion

$$\mathit{TimeoutFeedback} \Rightarrow \mathit{DelayTimeActive} + \mathit{DelayTimeFeedback} \leq \mathit{TimeoutFeedback}$$

der DCU-Fallstudie, die fordert, dass wenn das Attribut *TimeoutFeedback* ausgewählt wurde, der Wert des Attributs *TimeoutFeedback* größer als die Summe der Werte der zeitlichen Verzögerungen *DelayTimeActive* und *DelayTimeFeedback* sein muss. Insbesondere die Unterstützung von (potentiell) unendlichen Wertedomänen für Features und Attribute würde dabei ebenfalls neue Analysetechniken erfordern.

Ein weiteres zukünftiges Forschungsfeld könnte die Synthese von Rekonfigurationsautomaten sein. Die Rekonfigurationsautomaten für DSPLs werden in aktuellen Arbeiten manuell von Entwicklern erstellt. Gerade für umfangreiche Systeme wie der DCU kann dies schnell Rekonfigurationsautomaten mit hunderten oder tausenden von Zuständen führen, wenn mehr Rekonfigurationsverhalten als nur die Rekonfiguration der Betriebsmodi berücksichtigt wird. Aufgrund der größe solcher Rekonfigurationsautomaten könnten Entwickler die Übersicht über das spezifizierte Rekonfigurationsverhalten verlieren und Fehler einbauen. Ein alternativer Ansatz könnte es deshalb sein Rekonfigurationsautomaten aus einem Regelsatz, der beispielsweise in Hennessy-Milner-Logik [vHRF13] spezifiziert ist, automatisiert durch Synthese abzuleiten.

Zusätzlich ist es sinnvoll die Anwendbarkeit der vorgestellten Ansätze auf weitere Domänen zu prüfen, z. B. mobile Geräte [SLR13]. Weiterhin wäre es wünschenswert Bindungszeitrestriktionen automatisiert von DSPL-Artefakten und Implementierungs-Code ableiten zu können.

6

ZUSAMMENFASSUNG UND AUSBLICK

Das Ziel dieser Arbeit war die Entwicklung eines ganzheitlichen Ansatzes zur Spezifikation, Validierung und Verifikation von rekonfigurierbaren Software-Systemen. Als Basis für die Umsetzung dieses Ansatzes wurde das Paradigma der dynamischen Software-Produktlinien (DSPL) verwendet, da es die Konzepte der Variabilität im Raum und der Variabilität über die Zeit in sich vereint und somit eine valide Grundlage für die Entwicklung von Ansätzen zur Spezifikation, Validierung und Verifikation rekonfigurierbarer Software-Systeme darstellt.

In Abschnitt 2.1.1 wurde ein hochkonfigurierbares laufzeitadaptives medizintechnisches System aus der Industrie eingeführt, welches als motivierendes Beispiel für die in dieser Arbeit behandelten Forschungsherausforderungen verwendet wurde. Diese auf Abschnitt 1.1 basierenden Forschungsherausforderungen wurden am Ende der einzelnen Abschnitte in Kapitel 2 beschrieben und werden im Folgenden kurz wiederholt sowie zur Strukturierung der Zusammenfassung der in dieser Arbeit entwickelten wissenschaftlichen Beträge verwendet.

6.1 Zusammenfassung und Erkenntnisse

Die Forschungsherausforderung **FH3.1** dreht sich um die Entwicklung eines stufenweisen Konfigurationsprozesses für DSPLs, der die präzise Spezifikation von temporalen Abhängigkeiten zwischen den Konfigurationsentscheidungen für Features erlaubt. In Abschnitt 5.1 haben wir einen Ansatz zur Modellierung von DSPLs vorgestellt, welcher Feature-Modelle um Bindungszeitinformationen erweitert, sodass die Semantik von stufenweisen Rekonfigurationsprozessen unter Berücksichtigung von komplexen logischen und zeitlichen Restriktionen präzise formuliert werden kann. Weiterhin haben wir Rekonfigurationsautomaten verwendet, um die erlaubten Rekonfigurationen einer DSPL auf eine valide Teilmenge aller Rekonfigurationen einzuschränken.

Aufbauend auf der Forschungsherausforderung **FH3.1** beschäftigt sich die Forschungsherausforderung **FH3.2** mit der Frage nach Strategien zur automatisierten Validierung von (Re-)Konfigurationsprozessen von DSPLs inklusive der stufenweisen (Re-)Konfigurationen. Diesbezüglich wurden in Abschnitt 5.2 Ansätze zur Validierung von DSPL-Spezifikationen basierend auf Constraint-Solvern und Model-Checkern beschrieben. Unter Verwendung von Constraint-Solvern können dabei Feature-Modell-Anomalien wie Unerfüllbarkeit und unerwünschte Kern-Features, d. h. optionale Features, die aufgrund von Feature-Restriktionen in jeder Konfiguration enthalten sind, ausgeschlossen werden. Da mit Hilfe von Constraint-Solvern nicht das (potentiell unendliche) Rekonfigurationsverhalten von DSPLs während

des gesamten Produktlebenszyklus analysiert werden kann, haben wir eine Transformation von DSPL-Spezifikationen in Zustandsmaschinen vorgeschlagen, die als Eingabe für einen Model-Checker dienen. Dadurch können alle gewünschten Eigenschaften des Rekonfigurationsprozesses von DSPLs validiert werden, z. B. Fortschritt und Lebendigkeit.

Die Spezifikations- und Validierungsansätze für DSPLs wurden prototypisch implementiert. Für die Spezifikation der komplexen logischen und zeitlichen Restriktionen zwischen Features, Bindungszeiten und Attributen wurde eine Xtext-Grammatik verwendet. Zur Implementierung der Validierungstechniken kamen der SMT-Solver Z3 als Constraint-Solver und SPIN als Model-Checker zum Einsatz. Die Implementierung wurde auf eine reale DSPL aus der Industrie zu Evaluationszwecken angewendet. Die daraus entstandenen Ergebnisse zeigen, dass die oben beschriebenen Ansätze grundsätzlich zur präzisen Spezifikation realer DSPLs und zur automatisierten Validierung dieser Spezifikation geeignet sind und die Implementierung skaliert.

Die Forschungsherausforderungen **FH1**, **FH2.1**, **FH2.2** und **FH3.3** beschäftigen sich allesamt mit dem Testen von (D)SPLs, welches die in dieser Arbeit betrachtete Verifikationsstrategie für DSPL-Implementierungen ist. Der Fokus lag dabei auf der automatisierten und effizienten Generierung von Testfällen zum Testen des funktionalen Verhaltens und des Rekonfigurationsverhaltens von (D)SPLs. Dabei drehen sich die Forschungsherausforderungen insbesondere um die Erweiterung und Kombination existierender Testgenerierungsansätze mit dem Ziel der effizienten Wiederverwendung von Zwischenergebnissen der Testgenerierung zwischen Testzielen (**FH1**) und Produkten (**FH2.1**) sowie der Ausnutzung von potentiellen Synergien während der Kombination dieser Wiederverwendungsstrategien (**FH2.2**). Darauf aufbauend beschäftigte sich die Forschungsherausforderung **FH3.3** mit der Erweiterung dieser Testgenerierungsansätze um die Berücksichtigung des Rekonfigurationsverhaltens von DSPLs.

Um dies zu erreichen wurden in Kapitel 3 die Techniken zur Testgenerierung basierend auf Model-Checking von Beyer et al. [BCH+04, BHTV13] und Holzer [Hol13] mit (1) Techniken aus dem Bereich Multi-Property-Checking kombiniert und (2) die Vereinigung von Programmzuständen im Programmzustandsraum verbessert. (1) Multi-Property-Checking ist eine aktuelle Model-Checking-Technik, welche die parallele Betrachtung mehrerer zu überprüfender Programmeigenschaften erlaubt. Durch die Kombination von Testgenerierungstechniken mit Multi-Property-Checking ist es möglich, während eines Model-Checker-Laufs (d. h. während der Konstruktion eines abstrakten Erreichbarkeitsgraphen eines Programms) für mehrere Testziele eine Test-Suite zu generieren und dadurch eine systematische Wiederverwendung von Testgenerierungsergebnissen zwischen Testzielen zu ermöglichen. (2) Mit Hilfe der Verbesserung der Vereinigung von Programmzuständen im Programmzustandsraum kann die Größe des zu berechnenden Programmzustandsraums während der Testgenerierung reduziert werden, indem ähnliche Programmzustände häufiger zusammengefasst werden können. Bisher führte die Vereinigung von Programmzuständen häufig zu einem Verlust von Pfadsensitivität, wodurch für einen Programmpfad und somit für einen Testfall nach der Programmzustandsvereinigung nicht entschieden werden konnte, welche Testziele der

Testfall tatsächlich abdeckt. Um diesem Verlust an Pfadsensitivität entgegen zu wirken, der durch das vermehrte Vereinigen von Programmzuständen entstehen kann, haben wir das Konzept der symbolischen Pfadsensitivität verwendet. Symbolische Pfadsensitivität erlaubt es, Pfade im Programmzustandsraum eindeutig Testzielen zuzuordnen, sodass trotz der Vereinigung von Programmzuständen korrekte Test-Suiten generiert werden.

Die Techniken aus Kapitel 3 wurden in Kapitel 4 um die Berücksichtigung von Variabilität im Raum und somit um die Wiederverwendung von Testfällen zwischen Produkten einer (D)SPL erweitert. Durch die Verwendung von familienbasierten Testgenerierungsansätzen, bei denen alle Produkte einer SPL in einer einzigen SPL-Implementierung zusammen betrachtet werden, und der daraus resultierenden Wiederverwendung von Testfällen zwischen Testzielen und Produkten können kleinere Test-Suiten generiert und dabei die Anzahl der für die Testgenerierung benötigten Programmanalysen weiter reduziert werden. Aufbauend auf den Techniken zur Testgenerierung aus den Kapiteln 3 und 4 wurde in Kapitel 5.3 ebenfalls die Variabilität über die Zeit berücksichtigt, wodurch das (Rekonfigurations-)Verhalten von DSPLs während der Testgenerierung berücksichtigt werden konnte.

Die in dieser Arbeit präsentierten Techniken zur effizienten Testgenerierung wurden basierend auf dem Model-Checking-Framework CPAchecker implementiert und evaluiert. Die Ergebnisse der Evaluation zeigen, dass die Testgenerierung basierend auf Multi-Property-Checking mit symbolischer Pfadsensitivität durch die Betrachtung mehrerer Testziele in einem Lauf im Allgemeinen zu einer besseren Testeffizienz (kürzere Generierungsdauer und kleinere Test-Suite-Größen) bei einer ähnlichen Testeffektivität der generierten Test-Suiten (höhere Abdeckungsraten und Fehlerdetektionsraten) führt. Jedoch können nicht einfach alle Testziele gleichzeitig betrachtet werden, da es eine kritische Anzahl von Testzielen gibt, ab der die Vorteile der Testgenerierung mit Multi-Property-Checking durch zusätzlichen Berechnungsaufwand beeinträchtigt werden, der durch die gleichzeitige Betrachtung sehr vieler Testziele entsteht. Die Kombination der Wiederverwendung von Testfällen zwischen Produkten und Testzielen während der Testgenerierung für (D)SPLs führt ebenfalls dazu, dass die Größe generierter Test-Suiten und die dafür benötigten Model-Checker-Aufrufe im Vergleich zur Testgenerierung ohne diese Wiederverwendung drastisch reduziert werden. Dabei können die Ansätze, die eine komplette Familie von Software-Produkten gleichzeitig betrachten, aufgrund der Komplexität der SPL-Implementierungen mehr CPU-Zeit für die Test-Suite-Generierung als produktweise Ansätze benötigen.

6.2 Ausblick

Am jeweiligen Ende der Kapitel 3, 4 und 5 wurden wichtige Ideen für zukünftige Arbeiten diskutiert, welche wie folgt zusammengefasst werden können.

Bezüglich der Spezifikation von DSPLs ist ein interessanter zu untersuchender Aspekt die Erweiterung der Ausdrucksmächtigkeit der Grammatik für nicht-Boole'sche Restriktionen zwischen Features, Attributen und Bindungszeiten. Hierbei könnten beispielsweise mehr Sprachkonstrukte wie das logische Oder, weitere Wertedomänen oder Berechnung auf Attributwerten unterstützt werden. Insbesondere

die Unterstützung von (potentiell) unendlichen Wertedomänen für Features und Attribute würde dabei ebenfalls neue Analysetechniken für die Validierung der DSPL-Spezifikation erfordern. Ein weiteres zukünftiges Forschungsfeld könnte die Synthese von Rekonfigurationsautomaten sein, bei der Rekonfigurationsautomaten für DSPLs aus einem Regelsatz, der beispielsweise in Hennessy-Milner-Logik [vHRF13] spezifiziert ist, automatisiert abgeleitet werden. Dies ist insbesondere deshalb interessant, da Rekonfigurationsautomaten für umfangreiche Systeme (z. B. der DCU) schnell hunderte oder tausende von Zuständen enthalten können, die für einen Entwickler schnell unübersichtlich in der Handhabung werden können.

Bezüglich der Testgenerierung für DSPLs ist einer der interessantesten Aspekte die Partitionierung der Testzielmenge, wenn mit Hilfe von Multi-Property-Checking mehrere Testziele parallel betrachtet werden. Aktuell wird die Testzielmenge zufällig auf Partitionen einer bestimmten Größe verteilt. Zukünftig könnte eine systematische Partitionierung der Testzielmenge die Effizienz der Test-Suite-Generierung weiter verbessern. Beispielsweise könnten Testziele gemäß dem Kontrollfluss im CFA partitioniert werden, sodass Testziele, die auf den gleichen Programmpfaden oder in gleichen Produkten liegen, gemeinsam betrachtet werden. Weiterhin könnte die Partitionierung dynamisch erfolgen, sodass Testziele basierend auf bereits erlangten Informationen über die IUT des aktuellen Test-Suite-Generierungsprozesses partitioniert werden.

Wie oben erwähnt, dient das konfigurierbare Model-Checking-Framework CPAchecker, welches hunderte Konfigurationsoptionen enthält, als Basis für die Implementierung. Mit Hilfe dieser Konfigurationsoptionen kann die Test-Suite-Generierung potenziell weiter optimiert und an die Anforderungen unterschiedlicher Zielsysteme angepasst werden. Dies umfasst ebenfalls die Verwendung verschiedener Model-Checking-Technologien, wie explizites Model-Checking oder Bounded Model-Checking. In zukünftigen Forschungsarbeiten können die Auswirkungen der verschiedenen Konfigurationsoptionen und Model-Checking-Techniken des CPAchecker-Frameworks auf die Test-Suite-Generierung untersucht werden, um die Test-Suite-Generierung weiter zu verbessern.

[ABB+ar] Sven Apel, Dirk Beyer, Johannes Bürdek, Malte Lochau, and Andreas Stahlbauer. Configurable Test-Goal Set Partitioning for Multi-Goal Test-Suite Generation. Technical report, TU Darmstadt, 2018 (to appear). (Cited on page 78, 81 und 85.)

[ABF+13] Sven Apel, Dirk Beyer, Karlheinz Friedberger, Franco Raimondi, and Alexander von Rhein. Domain types: Abstract-domain selection based on variable usage. In *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, pages 262–278, 2013. (Cited on page 153.)

[ABKS13] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013. (Cited on page 25, 27, 30 und 158.)

[ABM+16] Sven Apel, Dirk Beyer, Vitaly O. Mordan, Vadim S. Mutilin, and Andreas Stahlbauer. On-the-fly decomposition of specifications in software model checking. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 349–361, 2016. (Cited on page 6, 46, 84, 85, 107, 108 und 154.)

[ACLF12] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Separation of concerns in feature modeling: support and applications. In *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012*, pages 1–12, 2012. (Cited on page 203.)

[AF14] Andrea Arcuri and Gordon Fraser. On the effectiveness of whole test suite generation. In *Search-Based Software Engineering - 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings*, pages 1–15, 2014. (Cited on page 109.)

[AHH11] Ebrahim Khalil Abbasi, Arnaud Hubaux, and Patrick Heymans. A toolset for feature-based configuration workflows. In *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, pages 65–69, 2011. (Cited on page 203.)

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009. (Cited on page 1 und 28.)

[AL13] Murat Aksu and Charlie Li. World quality report, 4th edition 2012-13. 2013. (Cited on page 3.)

[AtBFG11] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. A model-checking tool for families of services. In *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, pages 44–58, 2011. (Cited on page 4 und 113.)

[AvRW+13] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 482–491, 2013. (Cited on page 150 und 153.)

[Bat05] Don Batory. *Feature Models, Grammars, and Propositional Formulas*, pages 7–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. (Cited on page 31, 116, 163, 176 und 204.)

[BCH+04] Dirk Beyer, Adam Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 326–335, 2004. (Cited on page 3, 4, 24, 45, 46, 47, 59, 108, 110 und 208.)

[BCM+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 428–439, 1990. (Cited on page 47, 52 und 76.)

[Bey12] Dirk Beyer. Competition on software verification - (SV-COMP). In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 504–524, 2012. (Cited on page 101.)

[BGM13] Ella Bounimova, Patrice Godefroid, and David A. Molnar. Billions and billions of constraints: whitebox fuzz testing in production. In *35th International Conference on Software Engineering, IC-*

SE '13, San Francisco, CA, USA, May 18-26, 2013, pages 122–131, 2013. (Cited on page 3 und 45.)

[BHdA12] Nelly Bencomo, Svein O. Hallsteinsen, and Eduardo Santana de Almeida. A view of the dynamic software product line landscape. *IEEE Computer*, 45(10):36–41, 2012. (Cited on page 2, 7 und 203.)

[BHJP04] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. Specifying and generating test cases using observer automata. In *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, pages 125–139, 2004. (Cited on page 108.)

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 504–518, 2007. (Cited on page 62, 99, 101 und 150.)

[BHT08] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 29–38, 2008. (Cited on page 53, 54, 62, 63, 64, 69, 70 und 71.)

[BHTV13] Dirk Beyer, Andreas Holzer, Michael Tautschnig, and Helmut Veith. Information reuse for multi-goal reachability analyses. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 472–491, 2013. (Cited on page 6, 59, 108, 109, 110 und 208.)

[BK11] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 184–190, 2011. (Cited on page 99, 101 und 150.)

[BLB+15] Johannes Bürdek, Malte Lochau, Stefan Bauregger, Andreas Holzer, Alexander von Rhein, Sven Apel, and Dirk Beyer. Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines. In *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 84–99, 2015. (Cited on page 4, 48, 50, 109, 113, 114, 116, 123, 124, 132 und 133.)

[BLL+14] Johannes Bürdek, Sascha Lity, Malte Lochau, Markus Berens, Ursula Goltz, and Andy Schürr. Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints. In *The Eighth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '14, Sophia Antipolis, France, January 22-24, 2014*, pages 16:1–16:8, 2014. (Cited on page 157.)

[BMC05] David Benavides, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. Automated reasoning on feature models. In *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, pages 491–503, 2005. (Cited on page 2, 24, 176, 180 und 204.)

[BNP+07] Goetz Botterweck, Daren Nestor, André Preußner, Ciarán Cawley, and Steffen Thiel. Towards supporting feature configuration by interactive visualisation. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*, pages 125–131, 2007. (Cited on page 203.)

[BNRS08] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 3–14, 2008. (Cited on page 108.)

[BSBG08] Nelly Bencomo, Peter Sawyer, Gordon S. Blair, and Paul Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, pages 23–32, 2008. (Cited on page 2 und 203.)

[BSMC05] David Benavides, Sergio Segura, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. Using java CSP solvers in the automated analyses of feature models. In *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, pages 399–408, 2005. (Cited on page 176 und 204.)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, September 2010. (Cited on page 2 und 204.)

[BTL+15] Fabian Benduhn, Thomas Thüm, Malte Lochau, Thomas Leich, and Gunter Saake. A survey on modeling techniques for formal

behavioral verification of software product lines. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '15, Hildesheim, Germany, January 21-23, 2015*, page 80, 2015. (Cited on page 34.)

[CBT+14] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz Cortés, and Mike Hinchey. An overview of dynamic software product line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, 91:3–23, 2014. (Cited on page 2.)

[CCP+12] Maxime Cordy, Andreas Classen, Gilles Perrouin, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Simulation-based abstractions for software product-line model checking. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 672–682, 2012. (Cited on page 4, 113 und 154.)

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. (Cited on page 1 und 29.)

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999. (Cited on page 184.)

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*, pages 173–182, 2012. (Cited on page 2 und 30.)

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration using feature models. In *Software Product Lines, Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004, Proceedings*, pages 266–283, 2004. (Cited on page 2, 163, 165 und 203.)

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005. (Cited on page 203.)

[CHH09a] Andreas Classen, Arnaud Hubaux, and Patrick Heymans. Analysis of feature configuration workflows. In *RE 2009, 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31 - September 4, 2009*, pages 381–382, 2009. (Cited on page 164 und 203.)

[CHH09b] Andreas Classen, Arnaud Hubaux, and Patrick Heymans. A formal semantics for multi-level staged configuration. In *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, 2009. Proceedings*, pages 51–60, 2009. (Cited on page 164 und 203.)

[CHSL11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 321–330, 2011. (Cited on page 4, 113 und 154.)

[CJW15] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. Just test what you cannot verify! In *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 100–114, 2015. (Cited on page 108.)

[Cla03] Edmund M. Clarke. Counterexample-guided abstraction refinement. In *10th International Symposium on Temporal Representation and Reasoning / 4th International Conference on Temporal Logic (TIME-ICTL 2003), 8-10 July 2003, Cairns, Queensland, Australia*, page 7, 2003. (Cited on page 56.)

[CM94] John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994. (Cited on page 18 und 24.)

[CMW16] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 144–155, 2016. (Cited on page 108.)

[CN01] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Eng. Addison-Wesley, 2001. (Cited on page 2, 25, 26, 27 und 113.)

[COLS11] Harald Cichos, Sebastian Oster, Malte Lochau, and Andy Schürr. Model-based coverage-driven test suite generation for software product lines. In *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, pages 425–439, 2011. (Cited on page 4, 7, 39, 113, 121, 123, 128 und 153.)

[CS05] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: combining static checking and testing. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 422–431, 2005. (Cited on page 108.)

[CS06] Christoph Csallner and Yannis Smaragdakis. Dsd-crasher: a hybrid analysis tool for bug finding. In *Proceedings of the ACM/-SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 245–254, 2006. (Cited on page 108.)

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013. (Cited on page 3 und 45.)

[DGH16] Przemyslaw Daca, Ashutosh Gupta, and Thomas A. Henzinger. Abstraction-driven concolic testing. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, pages 328–347, 2016. (Cited on page 108.)

[dMSNdCMM+11] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. A systematic mapping study of software product lines testing. *Information & Software Technology*, 53(5):407–423, 2011. (Cited on page 34.)

[DPL+14] Xavier Devroey, Gilles Perrouin, Axel Legay, Maxime Cordy, Pierre-Yves Schobbens, and Patrick Heymans. Coverage criteria for behavioural testing of software product lines. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, pages 336–350, 2014. (Cited on page 7 und 153.)

[DPS14] Xavier Devroey, Gilles Perrouin, and Pierre-Yves Schobbens. Abstract test case generation for behavioural testing of software product lines. In *18th International Software Product Lines Conference - Companion Volume for Workshop, Tools and Demo papers, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 86–93, 2014. (Cited on page 7 und 153.)

[DS11] Ferruccio Damiani and Ina Schaefer. Dynamic delta-oriented programming. In *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011. Workshop Proceedings (Volume 2)*, page 34, 2011. (Cited on page 7, 160, 175 und 203.)

[ER10] Emelie Engström and Per Runeson. A qualitative survey of regression testing practices. In *Product-Focused Software Pro-*

cess Improvement, 11th International Conference, PROFES 2010, Limerick, Ireland, June 21-23, 2010. Proceedings, pages 3–16, 2010. (Cited on page 36 und 154.)

[ER11] Emelie Engström and Per Runeson. Software product line testing - A systematic mapping study. *Information & Software Technology*, 53(1):2–13, 2011. (Cited on page 34.)

[FA13] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Trans. Software Eng.*, 39(2):276–291, 2013. (Cited on page 109.)

[FDG08] Roman Froschauer, Deepak Dhungana, and Paul Grünbacher. Managing the life-cycle of industrial automation systems with product line variability models. In *34th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2008, September 3-5, 2008, Parma, Italy*, pages 35–42, 2008. (Cited on page 1.)

[Fer11] Anura Fernando. Software verification and validation: The role of IEC 60601-1. *Biomedical Instrumentation & Technology*, 45(5):370–372, sep 2011. (Cited on page 11.)

[FFVH12] Stefan Feldmann, J Fuchs, and Birgit Vogel-Heuser. Modularity, variant and version management in plant automation - future challenges and state of the art. In *Proceedings of International Design Conference, DESIGN*, pages 1689–1698, 05 2012. (Cited on page 1.)

[GHK+06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, pages 117–127, 2006. (Cited on page 108.)

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005. (Cited on page 108.)

[GLM08a] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Active property checking. In *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 207–216, 2008. (Cited on page 108.)

[GLM08b] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and*

Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008, 2008. (Cited on page 108.)

[GLM12] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012. (Cited on page 3 und 45.)

[GNRT10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai-Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 43–56, 2010. (Cited on page 108.)

[GSC+04] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent, and John Crupi. *Software Factories: Assembling Applications with Patterns, models, Frameworks and Tools*. Wiley, 2004. (Cited on page 29.)

[Gus07] Thomas Gustafsson. An approach for selecting software product line instances for testing. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*, pages 81–86, 2007. (Cited on page 36.)

[Hal05] Robert J. Hall. Fundamental nonmodularity in electronic mail. *Autom. Softw. Eng.*, 12(1):41–79, 2005. (Cited on page 150.)

[HCH09] Arnaud Hubaux, Andreas Classen, and Patrick Heymans. Formal modelling of feature configuration workflows. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 221–230, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. (Cited on page 2, 176, 182 und 203.)

[HdMR04] Grégoire Hamon, Leonardo Mendonça de Moura, and John M. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, pages 261–270, 2004. (Cited on page 4, 45 und 108.)

[Hel12] Michiel Helvensteijn. Dynamic delta modeling. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 127–134, New York, NY, USA, 2012. ACM. (Cited on page 175 und 203.)

[HHPS08] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, April 2008. (Cited on page 2, 7, 165, 166, 202 und 203.)

[HHS+13] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Khalil Abbasi. Supporting multiple perspectives in feature-based configuration. *Softw. Syst. Model.*, 12(3):641–663, July 2013. (Cited on page 203.)

[HHSD10] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, and Dirk Deridder. Towards multi-view feature-based configuration. In *Requirements Engineering: Foundation for Software Quality, 16th International Working Conference, REFSQ 2010, Essen, Germany, June 30 - July 2, 2010. Proceedings*, pages 106–112, 2010. (Cited on page 203.)

[HKL+10] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*, pages 182–191, 2010. (Cited on page 4, 45 und 109.)

[Hol13] Andreas Holzer. *Query-Based Test Case Generation*. PhD thesis, Technische Universität Wien, 2013. (Cited on page 6, 46, 47, 56, 57, 67, 69, 71, 74, 75, 78, 109, 110 und 208.)

[HPS12] Mike Hinchey, Sooyong Park, and Klaus Schmid. Building dynamic software product lines. *IEEE Computer*, 45(10):22–26, 2012. (Cited on page 7, 39, 159 und 202.)

[HSH+00] Pei-Hsin Ho, Thomas R. Shiple, Kevin Harer, James H. Kukula, Robert F. Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. Smart simulation using collaborative formal and simulation engines. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design, 2000, San Jose, California, USA, November 5-9, 2000*, pages 120–126, 2000. (Cited on page 108.)

[HSSF06] Svein O. Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. Using product line techniques to build adaptive systems. In *Software Product Lines, 10th International Conference, SPLC 2006, Baltimore, Maryland, USA, August 21-24, 2006, Proceedings*, pages 141–150, 2006. (Cited on page 39 und 159.)

[HST+08] Patrick Heymans, Pierre-Yves Schobbens, Jean-Christophe Trigaux, Yves Bontemps, Raimundas Matulevicius, and Andreas Classen. Evaluating formal properties of feature diagram languages. *IET Software*, 2(3):281–302, 2008. (Cited on page 204.)

[HSTV08] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Fshell: Systematic test case generation for dynamic analysis and measurement. In *Computer Aided Verificati-*

on, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings, pages 209–213, 2008. (Cited on page 109.)

[HSTV09] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Query-driven program testing. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, pages 151–166, 2009. (Cited on page 46, 47 und 109.)

[HSTV10] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. How did you specify your test suite. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 407–416, 2010. (Cited on page 56, 57, 58, 101, 108 und 150.)

[HTSV10] Andreas Holzer, Michael Tautschnig, Christian Schallhart, and Helmut Veith. An introduction to test specification in FQL. In *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers*, pages 9–22, 2010. (Cited on page 56, 57, 101 und 150.)

[IEE90] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990. (Cited on page 15.)

[JHF11] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. A survey of empirics of strategies for software product line testing. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, pages 266–269, 2011. (Cited on page 34.)

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990. (Cited on page 2, 30, 31, 161 und 204.)

[KGR+11] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 805–824, 2011. (Cited on page 37.)

[KMSL83] J. Kramer, J. Magee, M. Sloman, and A. Lister. Conic: an integrated approach to distributed computer control systems. *IEE*

Proceedings E - Computers and Digital Techniques, 130(1):1–, January 1983. (Cited on page 150.)

[KOD13] Ahmet Serkan Karatas, Halit Oguztüzün, and Ali H. Dogru. From extended feature models to constraint logic programming. *Sci. Comput. Program.*, 78(12):2295–2312, 2013. (Cited on page 161 und 204.)

[LBHS17] Malte Lochau, Johannes Bürdek, Stefan Hölzle, and Andy Schürr. Specification and automated validation of staged reconfiguration processes for dynamic software product lines. *Software and System Modeling*, 16(1):125–152, 2017. (Cited on page 10, 11 und 157.)

[LBL+14] Malte Lochau, Johannes Bürdek, Sascha Lity, Matthias Hagner, Christoph Legat, Ursula Goltz, and Andy Schürr. Applying Model-based Software Product Line Testing Approaches to the Automation Engineering Domain. *Automatisierungstechnik*, 62(11):771–780, 2014. (Cited on page 1.)

[Lig02] Peter Liggesmeyer. *Software-Qualität - testen, analysieren und verifizieren von Software*. Spektrum Akadem. Verl., 2002. (Cited on page 45.)

[Liu11] L. Liu. The application of mass customization in the automobile manufacturing industry. In *2011 International Conference on Consumer Electronics, Communications and Networks (CECNet)*, pages 808–811, April 2011. (Cited on page 24.)

[LK06] Jaejoon Lee and Kyo Chul Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *Software Product Lines, 10th International Conference, SPLC 2006, Baltimore, Maryland, USA, August 21-24, 2006, Proceedings*, pages 131–140, 2006. (Cited on page 39, 40 und 159.)

[LKF02] Harry C. Li, Shriram Krishnamurthi, and Kathi Fisler. Interfaces for modular feature verification. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*, pages 195–204, 2002. (Cited on page 154.)

[LKL12] Jihyun Lee, Sungwon Kang, and Danhyung Lee. A survey on software product line testing. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*, pages 31–40, 2012. (Cited on page 34.)

[LMBR14] Malte Lochau, Stephan Mennicke, Hauke Baller, and Lars Ribbeck. Deltaccs: A core calculus for behavioral change. In *Leveraging Applications of Formal Methods, Verification and Validation.*

Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I, pages 320–335, 2014. (Cited on page 154.)

[Loc13] Malte Lochau. *Model-Based Conformance Testing of Software Product Lines*. PhD thesis, University of Braunschweig - Institute of Technology, 2013. (Cited on page 35.)

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997. (Cited on page 108.)

[LSBL17] Lars Luthmann, Andreas Stephan, Johannes Bürdek, and Malte Lochau. Modeling and testing product lines with unbounded parametric real-time constraints. In *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A, Sevilla, Spain, September 25-29, 2017*, pages 104–113, 2017. (Cited on page 153.)

[LSKL12] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental model-based testing of delta-oriented software product lines. In *Tests and Proofs - 6th International Conference, TAP 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*, pages 67–82, 2012. (Cited on page 36 und 154.)

[MC09] Prabhat Mishra and Mingsong Chen. Efficient techniques for directed test generation using incremental satisfiability. In *VLSI Design 2009: Improving Productivity through Higher Abstraction, The 22nd International Conference on VLSI Design, New Delhi, India, 5-9 January 2009*, pages 65–70, 2009. (Cited on page 109.)

[MCdO07] Marcílio Mendonça, Donald D. Cowan, and Toacy Cavalcante de Oliveira. A process-centric approach for coordinating product configuration decisions. In *40th Hawaii International International Conference on Systems Science (HICSS-40 2007), CD-ROM / Abstracts Proceedings, 3-6 January 2007, Waikoloa, Big Island, HI, USA*, page 283, 2007. (Cited on page 203.)

[McG01] John McGregor. Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2001. (Cited on page 4, 39 und 113.)

[McM10] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 104–118, 2010. (Cited on page 108.)

[MJ10] C. R. Maga and N. Jazdi. An approach for modeling variants of industrial automation systems. In *2010 IEEE International*

Conference on Automation, Quality and Testing, Robotics (AQTR), volume 1, pages 1–6, May 2010. (Cited on page 1.)

[MLSW14] Stephan Mennicke, Malte Lochau, Julia Schroeter, and Tim Winkelmann. Automated verification of feature model configuration processes based on workflow petri nets. In *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 62–71, 2014. (Cited on page 164 und 203.)

[MMR+15] Raúl Mazo, Juan C. Muñoz-Fernández, Luisa Rincón, Camille Salinesi, and Gabriel Tamura. Variamos: an extensible tool for engineering (dynamic) product lines. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 374–379, 2015. (Cited on page 203.)

[MS07] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 416–426, 2007. (Cited on page 108.)

[MWC09] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 231–240, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. (Cited on page 2 und 204.)

[MZG03] Hong Mei, Wei Zhang, and Fang Gu. A feature oriented approach to modeling and reusing requirements of software product lines. In *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, pages 250–256, Nov 2003. (Cited on page 7 und 203.)

[OMR10] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, pages 196–210, 2010. (Cited on page 36.)

[OWES11] Sebastian Oster, Andreas Wübbeke, Gregor Engels, and Andy Schürr. Model-based testing for embedded systems. *Model-Based Testing for Embedded Systems*, pages 338–381, 2011. (Cited on page 34.)

[OZML11] Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. Moso-polite: tool support for pairwise and model-based software product line testing. In *Fifth International Workshop on Variability Modelling of Software-Intensive Systems, Namur, Belgium, January 27-29, 2011. Proceedings*, pages 79–82, 2011. (Cited on page 36.)

[Par76] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976. (Cited on page 24.)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1 edition, 2005. (Cited on page 1, 25, 27 und 30.)

[PNX+11] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. A study of non-boolean constraints in variability models of an embedded operating system. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 2:1–2:8, New York, NY, USA, 2011. ACM. (Cited on page 161.)

[PS08] Hendrik Post and Carsten Sinz. Configuration lifting: Verification meets software configuration. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 347–350, 2008. (Cited on page 4, 113 und 170.)

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 459–468, 2010. (Cited on page 36.)

[QCM10] Xiaoke Qin, Mingsong Chen, and Prabhat Mishra. Synchronized generation of directed tests using satisfiability solving. In *VLSI Design 2010: 23rd International Conference on VLSI Design, 9th International Conference on Embedded Systems, Bangalore, India, 3-7 January 2010*, pages 351–356, 2010. (Cited on page 109.)

[RSAS11] Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible feature binding in software product lines. *Automated Software Engineering*, 18(2):163–197, 2011. (Cited on page 204.)

[RSPA11] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Combining runtime adaptation and static binding in dynamic software product lines. *Technical Report 02, Otto-von-Guericke-Universität Magdeburg*, 2011. (Cited on page 204.)

[RSSA08] Marko Rosenmüller, Norbert Siegmund, Gunter Saake, and Sven Apel. Code generation to support static and dynamic composition of software product lines. In *Proceedings of the 7th*

International Conference on Generative Programming and Component Engineering, GPCE '08, pages 3–12, New York, NY, USA, 2008. ACM. (Cited on page 204.)

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 12–27, 1988. (Cited on page 52.)

[Sal15] Karsten Saller. *Model-based runtime adaption of resource constrained devices*. PhD thesis, Darmstadt University of Technology, Germany, 2015. (Cited on page 159 und 160.)

[SH11] Ina Schaefer and Reiner Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, 2011. (Cited on page 4 und 113.)

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148, Sept 2006. (Cited on page 204.)

[SL12] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest - Aus- und Weiterbildung zum Certified Tester, Foundation Level nach ISTQB-Standard (5. Aufl.)*. dpunkt.verlag, 2012. (Cited on page 15, 16, 18, 36, 45 und 101.)

[SLR13] Karsten Saller, Malte Lochau, and Ingo Reimund. Context-aware dspls: Model-based runtime adaptation for resource-constrained systems. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC '13 Workshops, pages 106–113, New York, NY, USA, 2013. ACM. (Cited on page 7, 159, 174, 199, 203 und 205.)

[SLW12] Julia Schroeter, Malte Lochau, and Tim Winkelmann. Multi-perspectives on feature models. In *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, pages 252–268, 2012. (Cited on page 203.)

[SMM+12] Julia Schroeter, Peter Mucha, Marcel Muth, Kay Jugel, and Malte Lochau. Dynamic configuration management of cloud-based applications. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 2*, pages 171–178, 2012. (Cited on page 203.)

[Som07] Ian Sommerville. *Software engineering*. International computer science series. Addison-Wesley, 8th edition, 2007. (Cited on page 15.)

[SOS+12] Karsten Saller, Sebastian Oster, Andy Schürr, Julia Schroeter, and Malte Lochau. Reducing feature models to improve runtime adaptivity on resource limited devices. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 2*, pages 135–142, 2012. (Cited on page 203.)

[SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A comparison of decision modeling approaches in product lines. In *Fifth International Workshop on Variability Modelling of Software-Intensive Systems, Namur, Belgium, January 27-29, 2011. Proceedings*, pages 119–126, 2011. (Cited on page 30.)

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Softw., Pract. Exper.*, 35(8):705–754, 2005. (Cited on page 41 und 203.)

[TAK+15] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. Analysis strategies for software product lines: A classification and survey. In *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, Dresden, Germany*, pages 57–58, 2015. (Cited on page 7 und 34.)

[TJ01] Mitchell M. Tseng and Jianxin Jiao. *Mass Customization*, pages 684–709. Wiley, 2001. (Cited on page 24.)

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007. (Cited on page 19, 20 und 23.)

[VBM15] Mahsa Varshosaz, Harsh Beohar, and Mohammad Reza Mousavi. Delta-oriented fsm-based testing. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, pages 366–381, 2015. (Cited on page 153.)

[vdLSR07] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action - the best industrial practice in product line engineering*. Springer, 2007. (Cited on page 27 und 28.)

[vHRF13] A. C. van Hulst, Michel A. Reniers, and Wan Fokkink. Maximal synthesis for hennessy-milner logic. In *13th International Conference on Application of Concurrency to System Design, ACSD 2013, Barcelona, Spain, 8-10 July, 2013*, pages 1–10, 2013. (Cited on page 205 und 210.)

[VHRF+16] Birgit Vogel-Heuser, Susanne Rösch, Juliane Fischer, Thomas Simon, Sebastian Ulewicz, and Jens Folmer. Fault handling

in PLC-based industry 4.0 automated production systems as a basis for restart and self-configuration and its evaluation. *Journal of Software Engineering and Applications*, 09(01):1–43, 2016. (Cited on page 39.)

[VPK04] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, pages 97–107, 2004. (Cited on page 108.)

[WCKK06] David M. Weiss, Paul C. Clements, Kyo Kang, and Charles Krueger. Software product line hall of fame. In *Proceedings of the 10th International on Software Product Line Conference*, SPLC '06, pages 237–, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on page 1 und 113.)

[WDSB09] Jules White, Brian Dougherty, Doulas C. Schmidt, and David Benavides. Automated reasoning for multi-step feature model configuration problems. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 11–20, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. (Cited on page 164 und 203.)

[Wen17] Philipp Wendler. *Towards Practical Predicate Analysis*. PhD thesis, University of Passau, 2017. (Cited on page 65.)

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. (Cited on page 27.)