Komponentenbasierte Synthese von Simulationsmodellen

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Fadil Kallat

Dortmund

2022

Tag der mündlichen Prüfung: 24. Mai 2023

Dekan: Prof. Dr.-Ing. Gernot Fink

Gutachter:

Prof. Dr. Jakob Rehof (Technische Universität Dortmund, Deutschland)
Prof. Dr.-Ing. Anne Meyer (Technische Universität Dortmund, Deutschland)

Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer **276879186**/**GRK2193**

Danksagung

Ich bedanke mich bei meinem Doktorvater Jakob Rehof, dafür, dass er mir ermöglichte in seiner Arbeitsgruppe zu forschen. Seine fachliche Kompetenz und seine Unterstützung waren für mich während der gesamten Zeit von unschätzbarem Wert. Ich bin ihm zutiefst dankbar für seine engagierte Betreuung und seine wertvolle Unterstützung. Ebenso möchte ich mich bei Anne Meyer für ihre hervorragende Unterstützung bedanken. Nicht nur ihre fachliche Kompetenz, sondern auch ihre Wissbegierde und ausgezeichneten Fragen haben mir sehr geholfen.

Auch möchte ich mich bei meinen Kolleg*innen am Lehrstuhl für das angenehme Arbeitsklima bedanken. Insbesondere möchte ich Christian Riest und Tristan Schäfer für unsere gewinnbringenden fachlichen Diskussionen als auch lustigen Kaffeerunden danken. Zudem bedanke ich mich bei Christian für das eifrige Korrekturlesen dieser Arbeit. Ein herzlicher Dank geht an Jan Bessai für seine Ratschläge und sein stets offenes Ohr.

Ich möchte mich auch bei meinen Mitstreitenden aus dem Graduiertenkolleg 2193 bedanken. Insbesondere bei Rui Li und Clara Scherbaum für die erholsamen Pausenspaziergänge und den interessanten und unterhaltsamen Gesprächen. Ich danke Carina Mieth und Jakob Pfrommer für die Zusammenarbeiten, die nicht nur eine große Bereicherung für meine Dissertation, sondern mir auch eine große Freude waren. Vielen Dank an meine ehemalige studentische Hilfskraft Thomas Schuster für die Unterstützung bei der Implementierung.

Auch geht mein Dank an die Deutsche Forschungsgemeinschaft, die mir ermöglichte, im Rahmen des Graduiertenkollegs 2193 die vorliegende Dissertation zu realisieren. Vielen Dank an TRUMPF für die Zusammenarbeit und das Vertrauen. Die Forschungsergebnisse dieser Kooperationen haben einen wesentlichen Beitrag zum Erfolg dieser Dissertation beigetragen.

Bedanken möchte ich mich auch bei meinen Eltern und Schwestern Farah und Jasmin für die Unterstützung und den Glauben an mich. Und natürlich geht ein unermessliches Dankeschön an meine geliebte Frau Kira, dafür, dass sie stets an mich glaubte, mich auch in den schwierigsten Phasen motivierte und immer den Rücken stärkte. Einen lieben Dank geht an meinen Sohn Lounis, der in den letzten Zügen der Dissertation das Licht der Welt erblickte. Er bescherte mir zwar kürzere Nächte, aber brachte umso mehr Freude und Glück in mein Leben. Er gab mir die Kraft und die Liebe zum Beenden der Dissertation und er ist das beste Geschenk, das ich mir je wünschen konnte.

Zusammenfassung

Die Simulation stellt in der Produktion und Logistik ein wesentliches Instrument dar, um ein virtuelles Abbild der Systeme aus diesem Bereich zu schaffen und dieses Abbild hinsichtlich relevanter Kennzahlen zu untersuchen, bevor diese Systeme gebaut oder angepasst werden. Jedoch ist die Modellierung dieser virtuellen Abbilder, auch Simulationsmodelle genannt, ein zeitaufwendiger und kostenintensiver Prozess. Insbesondere wenn unterschiedliche Varianten eines Systems untersucht werden sollen, übersteigen oftmals die benötigte Zeit und die entstehenden Kosten der Modellierung die verfügbaren Projektressourcen. Daraus folgt, dass in vielen Fällen lediglich eine Auswahl an Lösungsvarianten untersucht wird, sodass möglicherweise vielversprechende Lösungen nicht berücksichtigt werden. Diese Dissertation untersucht die Eignung der komponentenbasierten Softwaresynthese zur automatischen Generierung von ereignisorientierten Simulationsmodellen, die sich hinsichtlich ihrer Strukturen und den Kontrollstrategien unterscheiden. Bei diesem Ansatz werden die Simulationsmodelle nicht von Grund auf generiert, sondern aus Komponenten eines Repositoriums zusammengesetzt. Im Rahmen dieser Arbeit wurde eine Technologie entwickelt, die eine Migration von ereignisorientierten Simulationsmodellen in Produktlinien von Simulationsmodellen ermöglicht. Hierfür wurde eine Methodik zur (automatisierten) Aufteilung eines ereignisorientierten Simulationsmodells in getypte Komponenten entwickelt. Zu diesem Zweck wurden unterschiedliche Komponenten entworfen und implementiert, die Bestandteile eines ereignisorientierten Simulationsmodells produzieren, und anwendungsfallübergreifend dynamisch instantiiert werden können. Ferner wurde ein Vorgehen zur Anpassung der Komponenten entwickelt, das einer anwendenden Person, ohne Kenntnisse der Programmierung oder Synthese, die Markierung von Variabilitätspunkten ermöglicht. Letztere resultieren bei der Durchführung der komponentenbasierten Synthese in der Generierung von Simulationsmodellvarianten. Die Technologie wurde anhand der Migration von Simulationsmodellen unterschiedlicher praxisnaher Anwendungsfälle evaluiert. Damit konnte gezeigt werden, dass die Technologie die Planung von Systemen im Bereich der Produktion und der Logistik durch die automatisierte Generierung einer Vielzahl von Simulationsmodellen unter Verwendung von Softwaresynthese unterstützt und das Treffen von fundierten Entscheidungen erleichtert. Ferner bietet der Ansatz das Potenzial, Lösungsvarianten zu entdecken, die eine fabrikplanende Person möglicherweise nicht erwogen hätte. Durch die Ergebnisse dieser Dissertation konnte gezeigt werden, dass die komponentenbasierte Synthese zur Generierung von Varianten basierend auf der Variabilität der Struktur eingesetzt werden kann. Ferner wurde gezeigt, dass bestehende komplexe Simulationsmodelle schrittweise in eine Produktlinie migriert werden können.

Abstract

The event-based simulation is a common tool in production and logistics to create virtual representations of systems in this area. These virtual representations can be used to examine the systems regarding specific key performance indicators, even before a system is built or adopted. However, the modelling of such simulation models is a time-consuming and cost-intensive process. Especially when a large number of variants are to be examined, often the required time and costs exceeds the budget constraints in a project. By that, often not all possible variants can be considered, but a selection of variants is modeled. Thus, very promising solutions may be omitted. This dissertation examines the suitability of component-based software synthesis in terms of the automatic generation of event-based simulation models, which differ in their structure and control strategies. Hence, this dissertation presents an approach to migrate event-based simulation models into product lines of simulation models using component-based software synthesis. The approach contains a methodology for splitting an existing simulation models in typed components. Therefore, a number of typed components were designed that produces parts of a simulation model and allows a dynamical instantiation. Moreover, the approach allows a user to mark variability points, which leads to the generation of variants. The approach was implemented as a web application, which was used to evaluation regarding various industrial use cases. The approach aims to facilitate the planning of systems around production in logistics by terms of the automatic generation of simulation models using component-based software synthesis. The latter aims to support the decision-making process. Furthermore, the approach allows generating solutions which a factory planner would not have considered. This dissertation shows that the component-based software synthesis can be used to generate simulation models that especially differ in their structure. Moreover, the migration approach shows that complex simulation models can be migrated into product lines in a stepwise manner.

Abkürzungsverzeichnis

ABS Agent-based Simulation. 5, 20, 21

ASMG Automatic Simulation Model Generation. 5, 6, 34, 39, 40, 59

CLS Combinatory Logic Synthesizer. 10, 42, 43, 50–52, 54, 55, 57, 58, 69, 70, 73, 76, 82, 85, 95, 104, 117–120, 128–130, 133, 138, 179, 180, 186, 194, 196, 202

CMSD Core Manufacturing Simulation Data. 30, 31, 37, 40, 173, 205

DES Discrete-event Simulation. 4, 5, 17

GRK Graduiertenkolleg. 2, 3, 6, 7, 164

HTTP Hypertext Transfer Protocol. 85, 94

JSON JavaScript Object Notation. 73, 82–85, 91–93, 125, 128–131, 145–149

KL Kombinatorische Logik. 42–45, 47–50

RBG Regalbediengerät. 176, 177, 195

REST Representational State Transfer. 82–87, 90–93, 125, 128, 145, 148, 179

SMT Satisfiability Modulo Theories. 8–11, 38, 55, 63, 85, 109, 110, 112, 114–125, 127, 128, 130–133, 199

SysML Systems Modeling Language. 31, 207

UML Unified Modeling Language. 30, 31, 35, 36, 70, 72, 183, 187

URL Uniform Resource Locator. 85–87, 90

VDI Verein Deutscher Ingenieure. 13, 26, 28, 64, 65

XML Extensible Markup Language. 30, 34–36, 38, 70, 75, 96, 105–107, 109, 111, 132, 138, 145, 175, 201

Inhaltsverzeichnis

Kapitel 1

Motivation und Einleitung

Die Simulation ermöglicht die Nachahmung der Realität in einer virtuellen Umgebung. In dieser virtuellen Umgebung können Experimente durchgeführt werden, ohne hohe Kosten oder Zeitaufwände zu beanspruchen oder gar Menschenleben zu gefährden. Die Erkenntnisse dieser Experimente können anschließend auf die Realität übertragen werden. Die Simulation wird in den verschiedensten Bereichen eingesetzt, so zum Beispiel in der Produktion und Logistik, um unterschiedliche Varianten einer Produktionsstätte zu untersuchen, bevor diese gebaut oder angepasst wird. Hierfür wird häufig die ereignisorientierte Simulation eingesetzt, die sich durch das Vorkommen von festgelegten Ereignissen charakterisiert. Zum Beispiel stellen in einer Produktionsstätte die Ankunft von Rohmaterial oder die Fertigstellung der Verarbeitung von Aufträgen Ereignisse dar. Die virtuelle Umgebung wird in Form von Simulationsmodellen durch Simulationsfachkräfte modelliert. Anpassungen und Optimierungen hinsichtlich der Struktur oder des Verhaltens zur Untersuchung verschiedener Varianten können durch Anpassungen des Simulationsmodells evaluiert werden, bevor sie praktisch umgesetzt werden. Jedoch erhöht sich hierdurch bei einer Vielzahl von zu untersuchenden Varianten der Aufwand einer Simulationsstudie. Daher ist das Ziel dieser Dissertation, eine Möglichkeit zu entwickeln, die Generierung von ereignisorientierten Simulationsmodellen zu automatisieren, die sich insbesondere in Bezug auf die Struktur und das Verhalten unterscheiden. Ferner soll die Generierung nicht von Grund auf erfolgen, sondern bereits existente Simulationsmodelle verwenden. Hierzu soll eine Technologie entwickelt und implementiert werden, die eine Migration von Simulationsmodellen in eine Produktlinie mit Simulationsmodellvarianten unter Verwendung der komponentenbasierten Synthese ermöglicht. Die Markierung der Variabilitätspunkte soll durch eine anwendende Person erfolgen.

Die Technologie baut auf einem komponentenorientierten Ansatz auf. Hierzu sollen bereits existente Simulationsmodelle in Komponenten aufgeteilt werden. Dieser Schritt soll automatisiert durch eine Implementierung des Prozesses oder händisch durch die anwendende Person erfolgen. Die Komponenten sollen etwa die Produktion eines ausgewählten Bestandteils des Simulationsmodells verantworten. Überdies sollen die Komponenten semantische Informa-

tionen mit einer Beschreibung der Funktion sowie der Angabe der enthaltenen Bestandteile umfassen. Die Idee besteht darin, dass diese Komponenten zu beliebigen Simulationsmodellvarianten zusammengesetzt werden. Durch eine Anpassung der semantischen Informationen kann die anwendende Person beeinflussen, wie diese Varianten zusammengesetzt werden. Ferner soll die Erzeugung von Variabilität durch das Hinzufügen von Komponenten realisiert werden, die alternative Modellierungen von Bestandteilen umfassen. Die Komposition der Komponenten zu allen möglichen Varianten soll automatisiert erfolgen. Am Ende der Dissertation soll eine Konzeption und Implementierung einer Technologie entstehen, welche die beschriebene Migration eines ereignisorientierten Simulationsmodell in eine Produktlinie zur Synthese von Varianten erlaubt. Hierbei liegt der Fokus auf einem möglichst intuitiven und leicht zu bedienenden Zugang zu dieser Implementierung.

Zur Erreichung des Ziels werden Methoden der komponentenbasierten Softwaresynthese eingesetzt. Letztere ermöglicht, aus einer gegebenen Menge unterschiedlicher Softwarekomponenten individuelle Programme zu generieren. Die Synthese kann dabei als eine Suche über alle möglichen Programme betrachtet werden, die durch die Angabe einer logischen Spezifikation und weiteren Bedingungen eingeschränkt wird. Das Ergebnis der Synthese ist nicht zwangsläufig ein einzelnes Programm, sondern umfasst möglicherweise eine Familie von Programmen. Am Lehrstuhl für Software-Engineering der Technischen Universität Dortmund wird die kombinatorische Logiksynthese bereits zur Umsetzung der komponentenbasierten Softwaresynthese eingesetzt. Diese wird bereits erfolgreich für die Synthese von unter anderem Motion Plans [103, 104], Planungsvarianten im Kontext der Fabrikplanung [132] und Algorithmen zur Maschinenbelegung [77] verwendet.

1.1 Motivation und Hintergrund des Graduiertenkollegs 2193

Die vorliegende Dissertation ist im Rahmen des DFG-geförderten Graduiertenkolleg (GRK) 2193 entstanden, das sich mit der Anpassungsintelligenz von Fabriken in einem dynamischen und komplexen Umfeld beschäftigt. Die Besonderheit am Graduiertenkolleg 2193 ist die interdisziplinäre Ausrichtung durch eine Vielzahl angebundener Forschungsinstitute unterschiedlicher Fachdisziplinen. Die interdisziplinäre Ausrichtung begründet sich in den stark interdisziplinären Problemstellungen, die in dem Forschungsgebiet vorherrschen. Durch die interdisziplinäre Forschung wird ein besseres Verständnis für die Wechselwirkungen zwischen den einzelnen Forschungsgebieten geschaffen und eine Synchronisierung der fachspezifischen Lösungsansätze ermöglicht [29]. In diesem Abschnitt wird die Problemstellung des Graduiertenkolleg vorgestellt, ehe im darauffolgenden Abschnitt Forschungslücken identifiziert werden, die durch die vorliegende Dissertation adressiert werden.

Eine zunehmende Individualisierung der Produkte, immer kürzer werdende Produktlebenszyklen und zunehmende Absatzschwankungen gelten neben dem demographischen Wandel und einer fortschreitenden Digitalisierung als Auslöser für eine steigende Dynamik und Intensität

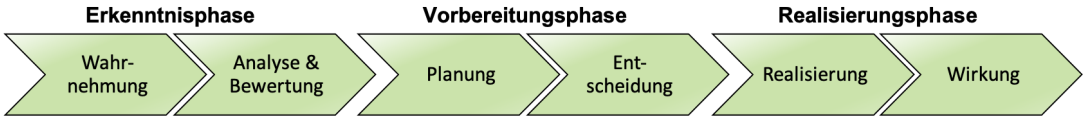


Abbildung 1.1: Phasen des Fabrikanpassungsprozesses nach Moralez [49], die in sequenzieller Reihenfolge von links nach rechts durchlaufen werden.

der Umfeldveränderungen eines Unternehmens. Durch das Bereithalten einer systemimmanenten Flexibilität und Wandlungsfähigkeit sind die Unternehmen imstande, schnell auf einen Teil dieser Umfeldveränderungen reagieren zu können. Dies ermöglicht Unternehmen bei Veränderungen, die in diesen Flexibilitätskorridoren liegen, ihre Fabriksysteme mit geringen Zeit- und Kostenaufwänden anzupassen [29]. Die Umfeldveränderungen, die außerhalb der Flexibilitätskorridore liegen, sind Entwicklungen, die zum Zeitpunkt der Planung des Fabriksystems nicht absehbar waren, wie technische Fortschritte [90]. Aufgrund der Häufigkeit und Intensität dieser nicht absehbaren Umfeldveränderungen befinden sich die Unternehmen in einem ständigen Planungs- und einem kurzzyklischen Anpassungsprozess [29].

Der Fabrikanpassungsprozess lässt sich nach Moralez [49] in die Erkenntnis-, Vorbereitungs- und Anpassungsphase aufteilen. Die Abbildung 1.1 zeigt die einzelnen Phasen sowie die einzelnen Schritte der jeweiligen Phase. In der Erkenntnisphase erfolgt die Wahrnehmung eines Anpassungsbedarfes, der durch die Veränderung eines Einflussfaktors ausgelöst wird. Nach der Wahrnehmung erfolgt eine Analyse und Bewertung der Signifikanz der Veränderung und dabei entscheidet sich, ob die Veränderung nicht durch den vorgehaltenen Flexibilitätskorridor abgedeckt ist, sondern eine Anpassung des Systems erfordert. Sofern letzteres zutrifft, folgt die Vorbereitungsphase der Fabrikanpassung. In der Planung erfolgt zunächst die Entwicklung, Konfiguration, Evaluation und Validierung von möglichen Anpassungsmaßnahmen, die den Flexibilitätskorridor des Systems derart verschieben, sodass das neue Anforderungsprofil umfasst wird. Im Anschluss an die Planung erfolgt die Entscheidung über die Implementierung von Maßnahmen, die in der Planung entwickelt wurden. In der Anpassungsphase erfolgt die Implementierung der beschlossenen Anpassungsmaßnahmen, die nach dem Ablauf einer Wirkungszeit ihre Wirkung entfalten [65]. Die gesamte Zeitspanne von der Erkenntnis über die Vorbereitung und der eigentlichen Implementierung bezeichnet die Reaktionszeit. Dabei gilt: Je kürzer die Reaktionszeit, desto größer ist der strategische Wettbewerbsvorteil für Unternehmen in Hochlohnländern [88].

Die Beschleunigung der Fabrikanpassungsprozesse stellt ein übergeordnetes Ziel des Graduiertenkollegs 2193 dar. Neben der Beschleunigung der Anpassungsprozesse, verfolgt die Forschung im Kolleg zudem das Ziel, die Qualität der Prozesse zu verbessern. [40] Im nachfolgenden Abschnitt wird erläutert, wie die Simulation und die komponentenbasierte Synthese von Simulationsmodellen einen Beitrag zur Erreichung dieser Ziele leisten.

1.2 Ausgangssituation und Problemstellung

Ein wesentliche, disziplinübergreifende Basis in der Forschung zur Anpassungsintelligenz stellt die adäquate Repräsentation eines Fabriksystems dar. Die digitale Abbildung, oft als digitaler Zwilling bezeichnet, dient als Unterstützung bei der Bewertung von Einflussfaktoren sowie zur Evaluation von möglichen Anpassungsmaßnahmen, die in der Vorbereitungsphase erarbeitet werden [29]. Die Gesellschaft für Informatik definiert den digitalen Zwilling als digitalen Repräsentanten einer Menge von physischen Objekten (z. B. Fabriksysteme) und nicht physischen Objekten (z. B. Produktionsprozesse) [54]. Häufig wird zur Realisierung und Untersuchung von digitalen Zwillingen die Computersimulation eingesetzt.

Die Computersimulation (nachfolgend Simulation genannt) ist in der Produktion und Logistik ein etabliertes Werkzeug zur Planung und Anpassung von Fabriksystemen [5, 84, 86]. Die Simulation wird beispielsweise zur Verbesserung der Ergonomie an Arbeitsplätzen, Planung von Fabrikgrundrissen, Optimierung von physikalischen Transformationsprozessen oder auch zur Untersuchung von Materialflüssen in Produktionsprozessen eingesetzt [84]. Die vorliegende Dissertation fokussiert in den untersuchten Anwendungsfällen insbesondere den Einsatz der Simulation im Kontext der Intralogistik, die sich mit den innerbetrieblichen Material- und Warenflüssen beschäftigt [3].

In der Intralogistik kann durch den Einsatz der Simulation in der Planungsphase die Leistungsfähigkeit noch nicht existierender Fabriksysteme untersucht und optimiert werden. Beispielsweise lässt sich die Dimensionierung eines Fabriksystems durch die Optimierung von Puffer- und Lagergrößen anpassen. Eine weitere Möglichkeit zur Anpassung der Dimensionierung besteht in dem Einsparen oder Vereinfachen von ursprünglich überdimensionierten Systemelementen [36]. Diese Auswahlmöglichkeiten können in einer Simulation konfiguriert und nach der Ausführung der Simulation ausgewertet werden. Zuvor müssen im Simulationsmodell die einzelnen Teilsysteme (z. B. Fördersysteme, Lagersysteme, Arbeitsstationen oder Wareneingänge und -ausgänge) inklusive der Parameter modelliert und in Beziehung gesetzt werden. Auch die Produktionsaufträge sind als Eingabe für die Simulation zu definieren. Weiterhin müssen im Simulationsmodell relevante Kennzahlen (z. B. durchschnittliche Durchlaufzeit, mittlerer Bestand an Arbeitsstationen oder Auslastung der Puffer und Lager) definiert werden, sodass diese nach der Ausführung des Simulationsmodells ausgewertet und zur Auswahl einer Lösungsvariante beitragen [32]. Während die statistische Auswertung der Kennzahlen als ein aussagekräftiges Entscheidungsinstrument gilt, wird die Visualisierung einer Simulation in Form einer Animation häufig bei der Präsentation bei den entsprechenden Stakeholdern eingesetzt und dient als Diskussionsgrundlage [126]. Die Abbildung 1.2 zeigt die 3D-Visualisierung eines Simulationsmodells in der Simulationsumgebung AnyLogic 8.

Der Begriff der Simulation dient als Oberbegriff für eine Sammlung verschiedenster Simulationsmethoden, die je nach Einsatzgebiet entsprechend ausgewählt werden. Im Bereich der Simulation von intralogistischen Systemen hat sich die ereignisorientierte Simulation (engl. Discrete-event Simulation (DES)) als gängige Simulationsmethodik etabliert [55]. Diese

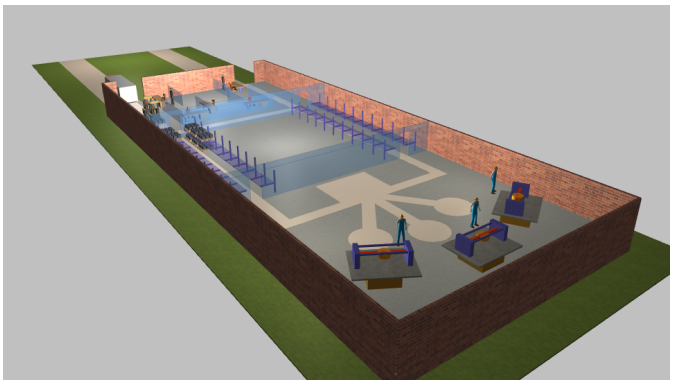


Abbildung 1.2: 3D-Visualisierung eines Simulationsmodells in der Simulationsumgebung AnyLogic 8, das ein Job-Shop-System mit Lagersystemen und Arbeitsstationen umfasst.

repräsentiert ein System durch verschiedene Zustände (z. B. Verarbeitung von Materialien an einer Arbeitsstation), die zu bestimmten Zeitpunkten betreten oder verlassen werden. Die Zeitpunkte ergeben sich durch sogenannte Ereignisse (z. B. Ankunft von Gütern am Wareneingang), die den Systemzustand ändern können [67]. Neben der DES hat die agentenbasierte Simulationen (engl. Agent-based Simulation (ABS)) in den vergangenen Jahren an Bedeutung im Bereich der Produktion und Logistik gewonnen [32, 55]. Dabei handelt es sich um eine Variante der ereignisorientierten Simulation, bei der autonome Agenten mit ihrer Umgebung und untereinander agieren. Daher schließt im Nachfolgenden die Verwendung des Begriffs der ereignisorientierten Simulation auch die agentenbasierte Simulation ein. Die Agenten verfügen über ein Verhalten, das durch Attribute und Entscheidungsregeln definiert ist. Das Verhalten wird durch die Interaktion mit der Umgebung beeinflusst und ändert sich somit möglicherweise im Laufe der Zeit [67].

In beiden Fällen gestaltet sich die Durchführung einer Simulationsstudie als derart aufwendig, sodass diese häufig ein eigenes Teilprojekt bildet. Aufgrund der Komplexität von Simulationsstudien haben sich einige Forschungsarbeiten der Ausarbeitung von Leitfäden zur Durchführung dieser Simulationsstudien gewidmet [89, 93, 128]. In den betrachteten Leitfaden bildet oftmals die Modellierung beziehungsweise die Implementierung eines Simulationsmodells einen Teilschritt, der mit einem hohen Zeit- und Kostenfaktor verbunden ist [53, 67, 89]. Der durchschnittliche, prozentuale Anteil, den die Modellierung von Simulationsmodellen in Simulationsstudien einnimmt, liegt bei etwa 36 % [53]. Der hohe Zeit- und Kostenfaktor erweist sich insbesondere als herausfordernd, wenn eine Vielzahl unterschiedlicher Planungsvarianten möglich ist. Da die Zeit- und Kostenressourcen zur Durchführung einer Simulationsstudie beschränkt sind, wird häufig eine Vorauswahl der zu modellierenden Planungsvarianten getroffen. Diese Vorauswahl erfolgt in der Regel auf Basis von Erfahrungen durch vorhergehende Planungen. Hierbei besteht das Risiko, dass Varianten aussortiert werden, die möglicherweise besonders zufriedenstellende Lösung darstellen [58]. Um dem entgegenzuwirken, stellt die Automatic Simulation Model Generation (ASMG) eine potenzielle Lösung dar, um die Kosten und die benötigte Zeit zur Modellierung der Simulationsmodelle zu reduzieren.

Bei der Automatic Simulation Model Generation (ASMG) werden Teile oder auch ganze Simulationsmodelle automatisiert generiert [81]. Dadurch können Teile einer Simulationsstudie ohne den Einsatz von Simulationsfachkräften durchgeführt werden [45, 87]. Dies wiederum resultiert in geringeren Kosten und einer kürzeren benötigten Zeitdauer einer Simulationsstudie insofern, als die Modellierung (teil)automatisiert erfolgt. Die ASMG stellt ein seit Jahrzehnten aktiv erforschtes Forschungsgebiet dar, wobei die Forschungsarbeiten häufig Problemstellungen aus der Produktion und Fertigung untersuchen [98]. Trotz der Vielzahl an Forschungsarbeiten werden fortlaufend Herausforderungen und Limitierungen als Forschungslücken genannt. So sind die Forschungsarbeiten vielfach auf einzelne Anwendungsfälle oder Problemstellungen zugeschnitten, sodass ein universaler Einsatz nicht möglich ist [98, 127]. Ferner stellt die Generierung von strukturellen Varianten eines Simulationsmodells eine Herausforderung dar [127]. In diesem Zusammenhang ist auch die Generierung von Kontrollstrategien zu nennen, die gewisse Logiken in einem Simulationsmodell abbilden. Bei der Generierung einer Strukturvariante eines Simulationsmodells müssen diese Kontrollstrategien an die neue Struktur angepasst werden. Jedoch beschränkt sich in vielen Forschungsarbeiten die Anpassung der Kontrollstrategien auf simple Kontrollstrategien, die durch einfache Entscheidungsregeln oder -tabellen definiert sind [127, 115]. In der Praxis werden Kontrollstrategien jedoch häufig durch Programme definiert, die in einer höheren Programmiersprache, wie Java oder Python, implementiert sind. Überdies hat die Generierung einer Vielzahl von Lösungsvarianten zur Folge, dass ab einer höheren Anzahl an generierten Simulationsmodellen eine automatisierte Ausführung und Evaluation unabdingbar ist. Ansonsten erweist sich die manuelle Ausführung der Simulationsmodelle als ein hinzukommender Flaschenhals [59].

1.3 Aufbau und Zielsetzungen der Arbeit

Das Ziel der vorliegenden Dissertation ist die Konzeption und Implementierung einer Technologie zur automatischen Simulationsmodellgenerierung unter Verwendung der komponentenbasierten Softwaresynthese. Unter Berücksichtigung der zuvor genannten Herausforderungen, den Ergebnissen einer Literaturrecherche hinsichtlich bisheriger Ansätze zur komponentenorientierten Generierung von Simulationsmodellen in Kapitel 2.3 sowie der Einbettung des Dissertationsprojekts in das GRK 2193 ergibt sich folgendes Anforderungsprofil:

1. Fähigkeit zur automatisierten Generierung einer Produktlinie von Simulationsmodellen, die sich hinsichtlich ihrer Struktur und ihrem Verhalten unterscheiden
2. Anwendbarkeit auf beliebige ereignisorientierte (und agentenbasierte) Simulationsmodelle
3. Anwendbarkeit auf Problemstellungen aus dem Bereich der Produktion und Logistik
4. Verwendbarkeit von bereits existierenden Simulationsmodellen, sodass eine Migration realisiert werden kann

5. Verwendbarkeit durch anwendende Personen, die über geringe bis keine Kenntnisse der Programmierung oder komponentenbasierten Softwaresynthese verfügen

Die erste Anforderung umfasst die grundlegendste funktionale Anforderung. So soll die Ausgabe der Implementierung eine Menge an Simulationsmodellen umfassen, die sich hinsichtlich ihrer Struktur und dem Verhalten voneinander unterscheiden. Die Struktur umfasst unterschiedliche Kompositionen der Bestandteile eines Simulationsmodells. Das Verhalten wird unter anderem durch die Kontrollstrategien des Simulationsmodells beeinflusst, die wiederum auf der Struktur basieren. Daher umfasst diese Anforderung ebenfalls die automatisierte Anpassung der Kontrollstrategien. Die zweite und dritte Anforderung betrifft die Annahmen bezüglich der zu unterstützenden Simulationsmodelle. So soll die Migration von ereignisorientierten und agentenbasierten Simulationsmodellen in Produktlinien möglich sein. Weiterhin soll der Einsatz in Simulationsstudien zur Untersuchung von Systemen aus der Produktion und Logistik möglich sein. Letztere Anforderung ergibt sich durch die Einbettung im Graduiertenkolleg. Diese fordert, dass der zu entwickelnde Ansatz einen Beitrag zur Fabrikplanung und -anpassung liefern soll. Die vierte Anforderung fordert, dass Simulationsmodelle, die durch Simulationsfachkräfte erstellt werden, als Ausgangsbasis für die Migration und damit auch für die Synthese der Simulationsmodellvarianten dienen sollen. Dadurch kann der Ansatz in bestehende Vorgehensmodelle zur Durchführung von Simulationsstudien integriert werden. Zum Beispiel als einen zusätzlichen Schritt zwischen der Modellierung eines Simulationsmodells und der Durchführung der Experimente. Um die Barriere der Verwendung der Technologie zu reduzieren, fordert die fünfte Anforderung, dass der Ansatz ohne Kenntnisse der Programmierung oder den theoretischen Grundlagen des verwendeten Verfahrens zur komponentenbasierten Synthese nutzbar sein soll. Abschließend sei anzumerken, dass die Dissertation übergeordnete Ziele aus der Sicht zweier Fachdisziplinen fokussiert. Aus der Sicht der Informatik wird die Erforschung der komponentenbasierten Synthese von strukturellen Varianten untersucht, während die Zielstellung aus der logistischen Sicht das Finden der besten Lösung gemessen an festgelegten Kennzahlen verfolgt.

In dieser Dissertation werden in Kapitel 2 die Grundlagen der Simulation und deren Einsatz in der Produktion und Logistik eingeführt. Weiterhin werden in dem Kapitel die Ergebnisse einer Literaturrecherche hinsichtlich bisheriger Ansätze der komponentenorientierten Generierung von Simulationsmodellen vorgestellt. Im darauffolgenden Kapitel 3 wird ein Überblick über Verfahren der Softwaresynthese gegeben, ehe die theoretischen Grundlagen der kombinatorischen Logiksynthese folgen. Ebenso wird eine Implementierung der kombinatorischen Logiksynthese vorgestellt, die im Rahmen der Implementierung der zu entwickelnden Technologie eingesetzt wird. In Kapitel 4 wird zunächst das Vorgehen der Migration von Simulationsmodellen motiviert. Anschließend folgt die Vorstellung der Konzeption und Realisierung dieses Vorgehens. Ferner wird die Implementierung der Technologie thematisiert. In den Kapiteln 5, 6 und 7 folgt der Einsatz der Technologie zur Migration von Simulationsmodellen unterschiedlicher Anwendungsfälle. Diese unterscheiden sich nicht nur in Bezug auf den Einsatzzweck, sondern ebenso in der Art der Generierung, die entweder

intern in einem Simulationsmodell eingebettet oder extern außerhalb der Simulationsumgebung erfolgt. Weiterhin fokussiert jeder Anwendungsfall einen thematischen Schwerpunkt, wie das Filtern mittels Techniken des SMT-Constraint-Solvings. Abschließend folgt in Kapitel 8 eine Zusammenfassung sowie eine kritische Auseinandersetzung mit den Ergebnissen der Dissertation, ehe ein Ausblick auf weiterführende Arbeiten gegeben wird.

1.4 Wissenschaftliche Beiträge der Arbeit

In diesem Abschnitt werden die wissenschaftlichen Beiträge aufgelistet, die im Rahmen der Forschung zur Erstellung der vorliegenden Dissertation erarbeitet wurden. Die Beiträge lassen sich auf theoretische und technische Beiträge aufteilen:

Die **theoretischen** Beiträge umfassen:

1. Literaturrecherche bezüglich bisheriger Ansätze zur komponentenorientierten, automatischen Generierung von Simulationsmodellen.
2. Die Ergebnisse dieser Literaturrecherche wurden zur Gestaltung von Simulationsbausteinen und Komponenten verwendet. Erstere repräsentieren Bestandteile des Simulationsmodells und letztere produzieren Kompositionen der Simulationsbausteine. Hierfür sind Anforderungen sowie ein Datenmodell zur Repräsentation dieser vorhanden.
3. Ein Vorgehen zur automatisierten Aufteilung eines Simulationsmodells in Simulationsbausteine und Komponenten. Das Vorgehen umfasst ein Schema zur Typisierung der Komponenten, unabhängig von konkreten Anwendungsfällen.
4. Erarbeitung einer Technologie, welche
   (a) die kombinatorische Logiksynthese nutzt, um Simulationsmodellvarianten, die sich hinsichtlich ihrer Struktur unterscheiden, zu synthetisieren
   (b) einer anwendenden Person die zielgerichtete Markierung von Variabilitätspunkte ermöglicht, die im Rahmen der Softwaresynthese berücksichtigt werden
   (c) bereits existierende Simulationsmodelle als Ausgangsbasis nutzt und eine Migration erlaubt
   (d) in gängige Vorgehensmodelle zur Durchführung von Simulationsstudien eingebunden werden kann
   (e) sowohl in Simulationsmodellen eingebettet (intern) als auch außerhalb von Simulationsmodellen eingesetzt werden kann (extern)
5. Erweiterung der Technologie um die komponentenbasierte Synthese von Kontrollstrategien. Diese ermöglicht die Migration von Kontrollstrategien in Produktlinien zur Synthese von Kontrollstrategien, die auf zuvor synthetisierte Strukturvarianten zugeschnitten sind.

6. Verwendung von Techniken des SMT-Constraint-Solving in Ergänzung zur kombinatorischen Logiksynthese in Bezug auf die Filterung von synthetisierten Simulationsmodellen.

Die **technischen** Beiträge umfassen:

7. Konzeption und Implementierung einer Softwarearchitektur zur Realisierung der externen Generierung von Simulationsmodellen der Simulationsumgebung AnyLogic 8 mittels der komponentenbasierten Synthese. Die Softwarearchitektur ist in Form einer Client-Server-Webanwendung gestaltet, die einen cloudbasierten Betrieb nach dem as-a-Service Prinzip erlaubt. Die Webanwendung erstreckt sich über sämtliche Phasen der Migration eines Simulationsmodells. Insbesondere wird eine Unterstützung bei der Anpassung von Komponenten zur Variantenbildung geboten, die eine Vereinfachung der Verwendung anstrebt.

8. Konzeption und Implementierung einer Softwarearchitektur zur Realisierung der internen Generierung von Simulationsmodellen innerhalb der Simulationsumgebung AnyLogic 8 mittels der komponentenbasierten Synthese. Hierbei ist eine Schnittstelle zur Kommunikation zwischen dem Simulationsmodell und der Implementierung dieser Technologie entstanden.

9. Die Einsatzfähigkeit der konzeptionierten und implementierten Technologie wurden anhand der Migration von Simulationsmodellen unterschiedlicher Anwendungsfälle in Produktlinien demonstriert.

1.5 Eigene Publikationen

Im Rahmen dieser Dissertation sind Forschungsbeiträge entstanden, die auf wissenschaftlichen Konferenzen veröffentlicht wurden. Die Inhalte der Veröffentlichungen überschneiden sich dementsprechend mit den Inhalten dieser Dissertation. Nachfolgend werden die Veröffentlichungen kurz vorgestellt und es folgt eine Abgrenzung des Eigenanteils.

In der ersten Veröffentlichung ist ein Framework entstanden, welches das SMT-Constraint-Solving und die komponentenbasierte Synthese kombiniert, um das Filtern von synthetisierten Lösungen zu ermöglichen. Der Eigenanteil im Rahmen dieser Veröffentlichung besteht in der Durchführung der Literaturrecherche in Bezug auf ähnliche Ansätze sowie dem Testen der Implementierung.

In der zweiten Veröffentlichung wird eine erste Version der Technologie vorgestellt, die eine komponentenbasierte Synthese von agentenbasierten Simulationsmodellen ermöglicht. Ferner wird das Framework aus der ersten Veröffentlichung verwendet, um die synthetisierten Simulationsmodelle zu filtern. Der Eigenanteil besteht in der Gestaltung und Implementierung

der Komponentensammlung, der Ergänzung von SMT-Formeln sowie in der Durchführung der komponentenbasierten Synthese und des Filterns.

In der dritten Veröffentlichung wird der Ansatz derart erweitert, dass beliebige ereignisorientierte Simulationsmodelle in Komponentensammlungen migriert werden können. Hierbei besteht der Eigenanteil in der Konzeption und Implementierung der automatisierten Aufteilung in Komponenten und deren Weiterverarbeitung. Ferner umfasst der Eigenanteil die Implementierung der Webanwendung zur Verwendung des Ansatzes sowie die Durchführung der komponentenbasierten Synthese.

In der vierten Veröffentlichung wird der Migrationsprozess als interner Ansatz zur Generierung von Simulationsmodellen realisiert, indem die Migration in ein bestehendes Simulationsmodell eingebettet wird. Der Eigenanteil im Rahmen dieser Veröffentlichung umfasst die Konzeption und Implementierung einer Softwarearchitektur zur Realisierung des eingebetteten Ansatzes, die Erstellung der Komponenten sowie die Unterstützung in der Durchführung der Simulationsstudie. Ferner umfasst der Eigenanteil die Erarbeitung des Vorgehens zur Auslagerung von Agenten in Maschinenzellen. Nachfolgend werden die Veröffentlichungen detaillierter vorgestellt.

1.5.1 CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories

- **Autorenschaft:** Fadil Kallat, Tristan Schäfer, Anna Vasileva
- **Erschienen in:** Proceedings of the Sixth Workshop on Proof eXchange for Theorem Proving (PxTP), Natal, Brasilien, 2019 [60]

In dieser Veröffentlichung wird die Kombination von SMT-Techniken mit der kombinatorischen Logiksynthese untersucht. Dabei wird die Übersetzung einer Baumgrammatik, die das Ergebnis der Synthese durch das Framework Combinatory Logic Synthesizer (CLS) darstellt, in korrespondierende SMT-Formeln vorgestellt. Diese SMT-Formeln werden anschließend um domänenspezifische Randbedingungen, die ebenfalls als SMT-Formeln formuliert sind, erweitert. Mittels eines SMT-Solvers werden anschließend Modelle berechnet, die Wörter der Baumgrammatik darstellen, die wiederum validen Inhabitanten entsprechen, welche die domänenspezifischen Randbedingungen berücksichtigen. Im Rahmen dieses Dissertationsprojekts ist diese Veröffentlichung eine Vorarbeit für die nachfolgende Veröffentlichung, in der die SMT-Techniken eingesetzt werden, um Fabrikkonfigurationen nach bestimmten Randbedingungen zu filtern und die Lösungen nach gegebenen Optimierungskriterien zu sortieren. Durch diese Veröffentlichung wurde gezeigt, dass die Verwendung von SMT-Techniken hierfür geeignet ist und es wurde ein Framework entwickelt, das die Übersetzung in SMT-Formeln ermöglicht.

1.5.2 Using Component-based Software Synthesis and Constraint Solving to generate Sets of Manufacturing Simulation Models

- **Autorenschaft:** Fadil Kallat, Carina Mieth, Jakob Rehof, Anne Meyer
- **Erschienen in:** Proceedings of the 53rd CIRP Conference on Manufacturing Systems (CMS), Chicago, USA, 2020 [58]

Im Rahmen dieser Veröffentlichung wird ein Simulationsmodell einer blechverarbeitenden Fabrik in eine Produktlinie zur Synthese von Simulationsmodellvarianten migriert. Die Variabilitätspunkte in dem betrachteten Anwendungsfall umfassen Maschinenkonfigurationen, die sich beispielsweise hinsichtlich der Verarbeitungsgeschwindigkeit oder der kompatiblen Maße des Rohmaterials unterscheiden. In Ergänzung zur komponentenbasierten Synthese werden in dieser Veröffentlichung SMT-Techniken zur Filterung von Lösungen verwendet. Hierfür wird das Framework aus der ersten Veröffentlichung verwendet. Das Filtern erfolgt basierend auf numerischen Randbedingungen wie der Durchlaufzeit oder Kosten. Hierzu werden die Komponenten um numerische Gewichtungen ergänzt. Damit wird in dieser Veröffentlichung die Kombination der komponentenbasierten Synthese und der SMT-Techniken im Kontext der Generierung von Simulationsmodellen demonstriert. Ferner wird die Idee der Migration eines bereits existierenden Simulationsmodells in eine Produktlinie vorgestellt, die in nachfolgenden Veröffentlichungen weiter verfolgt wird.

1.5.3 Automatic Building of a Repository for Component-based Synthesis of Warehouse Simulation Models

- **Autorenschaft:** Fadil Kallat, Jakob Pfrommer, Jakob Rehof, Anne Meyer
- **Erschienen in:** Proceedings of the 54th CIRP Conference on Manufacturing Systems (CMS), Athen, Griechenland, 2021 [59]

In dieser Veröffentlichung wird ein Simulationsmodell eines Bodenblocklagers in eine Produktlinie zur Synthese entsprechender Simulationsmodelle migriert. Die Variabilitätspunkte in diesem Anwendungsfall bilden die Anzahl der Wareneingänge und -ausgänge, die Wahl der Umlagerungsstrategie sowie das wahlweise Auslassen der Umlagerung von Gütern im Lager. Im Fokus dieser Veröffentlichung steht die automatisierte Erstellung einer Komponentensammlung. Hierfür wurde ein Vorgehen entwickelt, dass automatisiert Simulationsbausteine eines Simulationsmodells ausliest und diese in getypte Komponenten überführt. Ferner ist eine Benutzeroberfläche entstanden, die einer anwendenden Person die Markierung von Variationspunkten erlaubt. Dies ermöglicht die Verwendung der komponentenbasierten Synthese von Simulationsmodellen ohne Programmierkenntnisse oder tiefgreifende Kenntnisse über die komponentenbasierte Softwaresynthese. Weiterhin umfasst die Veröffentlichung eine durchgeführte Simulationsstudie, basierend auf den synthetisierten Simulationsmodellen.

1.5.4 Automatic Component-based Synthesis of User-configured Manufacturing Simulation Models

- **Autorenschaft:** Fadil Kallat, Jens Hetzler, Alexander Mages, Carina Mieth, Jakob Rehof, Christian Riest und Tristan Schäfer
- **Erscheint in:** Proceedings of the 2022 Winter Simulation Conference (WSC), Singapur, Singapur, 2022 [37]

Diese Veröffentlichung stellt eine Fortführung der Forschungsarbeiten in [58] dar. In dieser fortführenden Arbeit wird kein einzelnes Simulationsmodell einer blechverarbeitenden Fabrik, sondern ein Simulationsbaukasten in eine Produktlinie migriert. Dieser Simulationsbaukasten umfasst unterschiedliche Bausteine in Form von Agenten, welche die Modellierung von Fabriksystemen ermöglichen. So repräsentieren die Bausteine unter anderem unterschiedliche Maschinentypen oder Lagersysteme. Im Rahmen der Veröffentlichung wird eine Auswahl dieser Simulationsbausteine in Komponenten überführt. Die fachlichen Variabilitätspunkte bilden die Anzahl und Auswahl von Maschinentypen sowie die Anzahl der Lagertürme eines Lagersystems. Weiterhin wird in dieser Veröffentlichung ein Vorgehen zur internen Generierung von Simulationsmodellen unter Verwendung der komponentenbasierten Softwaresynthese entwickelt. In diesem Vorgehen interagiert die anwendende Person mit einer grafischen Benutzeroberfläche, in der die zu synthetisierenden Simulationsmodellvarianten konfiguriert werden. Die Konfiguration erfolgt nicht auf Basis der Komponenten, sondern beispielsweise basierend auf der Angabe von Intervallen, die eine Mindest- und Maximalanzahl an Maschinen umfasst. Diese Konfiguration wird automatisiert in ein Syntheseziel übersetzt und zur komponentenbasierten Synthese der Varianten verwendet. Auch in dieser Veröffentlichung werden die synthetisierten Simulationsmodellvarianten im Rahmen einer exemplarischen Simulationsstudie verwendet.

Kapitel 2

Simulation in der Produktion und Logistik

Die Simulation ist eine sehr beliebte Methode, um beliebige Abläufe vor ihrem Eintreten zu untersuchen und somit die Entscheidungsfindung zu unterstützen. Dabei ist die Verwendung der Simulation auch vor der Ära der Computer dokumentiert, so zum Beispiel in der historischen Kriegsführung, bei der mittels geografischer Karten und darauf platzierter Figuren, die Bewegungen der eigenen und feindlichen Seiten sowie unterschiedliche Szenarien simuliert werden. Das Interesse an der Computersimulation und deren Nutzung nimmt seit den 70er-Jahren stetig zu. Dabei sind die Einsatzgebiete recht unterschiedlich und umfassen etwa die Raum- und Luftfahrt, Elektrotechnik, das Gesundheits-, Transport- oder Bauwesen sowie die Produktion und Logistik. [67] Der Verein Deutscher Ingenieure (VDI) hat den Begriff der Simulation für den Bereich der Produktion und Logistik folgendermaßen definiert [1]: „Simulation ist das Nachbilden eines Systems mit seinen dynamischen Prozessen in einem experimentierbaren Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind“.

In diesem Kapitel wird die Simulation im Kontext der Produktion und Logistik vorgestellt. Hierzu werden zunächst die theoretischen Grundlagen eingeführt. Anschließend werden Produktions- und Logistiksysteme definiert sowie die Nutzung der Simulation zur Untersuchung dieser Systeme. Daraufhin folgen die Ergebnisse einer Literaturrecherche, die komponentenorientierte Ansätzen zur automatischen Generierung von Simulationsmodellen umfasst. Abschließend erfolgt eine Vorstellung der identifizierten Forschungslücken.

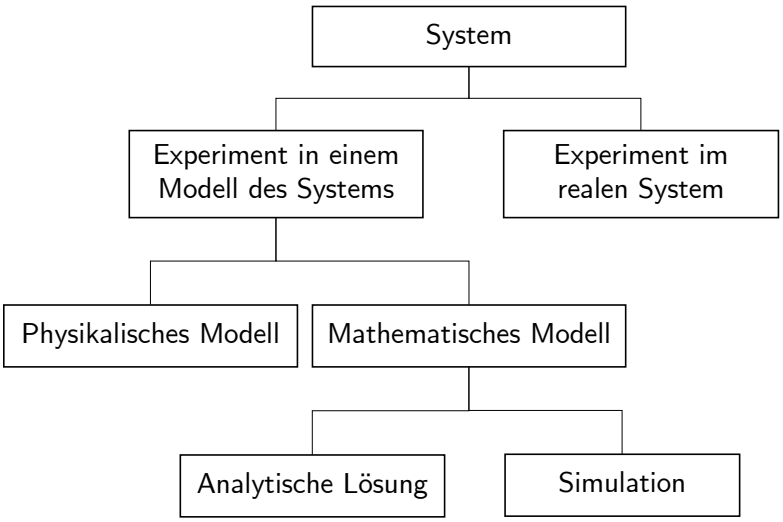


Abbildung 2.1: Möglichkeiten zur systematischen Untersuchung eines Systems (Quelle: In Anlehnung an Law [67]).

2.1 Theoretische Grundlagen zur Simulation

In diesem Unterkapitel werden die notwendigen theoretischen Grundlagen der Simulation in Bezug auf diese Dissertation eingeführt. Hierzu werden zunächst die relevanten Grundbegriffe vorgestellt, ehe eine Klassifikation von Simulationsmodellen anhand dreier Dimensionen folgt. Anschließend werden die ereignisorientierte und agentenbasierte Simulationsmethodik und eine Modellierung von Simulationsmodellen basierend auf Bausteinbibliotheken vorgestellt.

2.1.1 Grundbegriffe

Die Simulation ermöglicht die computergestützte Imitation des Betriebs von in der Realität vorhandenen *Objekten* wie Fabriken, Kraftfahrzeuge oder Prozesse. Diese Objekte werden im Kontext der Simulation als *Systeme* bezeichnet [23]. Ein System ist wiederum als eine Sammlung von Entitäten (z. B. Maschinen, Ottomotoren oder Prozessbausteine) definiert, die im Verbund zur Bewältigung einer übergeordneten Aufgabe (inter)-agieren [21].

Ein System kann auf unterschiedliche Weisen untersucht werden, die in der Abbildung 2.1 zusammengefasst sind. Hierbei ist grundlegend zu unterscheiden, ob die Experimente in einem realen System oder in einem Modell durchgeführt werden. Jedoch sind Experimente in realen Systemen nicht immer durchführbar. Zum Beispiel, wenn ein System noch nicht vorhanden ist (z. B. Neubau eines Fabriksystems) oder die Gefährdung von Menschenleben nicht ausgeschlossen werden kann (z. B. Abschuss von Raumfahrzeugen in der Raumfahrt). Ferner gestalten sich diese Experimente häufig aus Kosten-Nutzen-Sicht als weniger sinnvoll. Stattdessen können die Experimente auch in einem Modell eines Systems durchgeführt

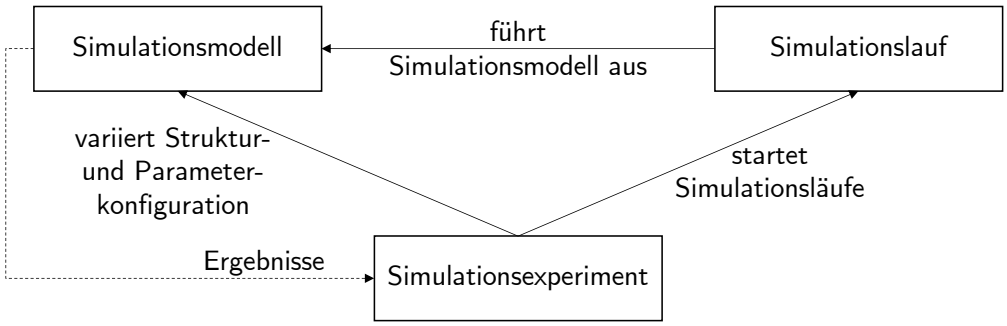


Abbildung 2.2: Schematische Darstellung der Beziehung zwischen dem Simulationsmodell, -experiment und -lauf.

werden. Hierbei kann zwischen physikalischen und mathematischen Modellen unterschieden werden. Beispiele für physikalische Modelle sind Kraftfahrzeuge in einem Windkanal, eingebettete Flugzeugcockpits in einem Flugsimulator oder schwimmende Miniatur-Schiffe in einem Wasserbehälter. Einem mathematischen Modell liegen eine Reihe von Annahmen zugrunde, welche die Funktionsweise des Systems beschreiben [67]. Bei diesen Annahmen handelt es sich um logische und quantitative Beziehungen, deren systematische Veränderung Auswirkungen auf das zu untersuchende System hat. Solch ein Modell kann durch die Festlegung von Ein- und Ausgaben modelliert werden. Die Eingaben stellen Variablen dar, die über einen betrachteten Zeitraum variiert werden, während die Ausgaben die berechneten Ergebnisse darstellen. Die Verbindung zwischen den Ein- und Ausgaben erfolgt durch das Aufstellen von Funktionen. Da die Modellierung über einen Freiheitsgrad hinsichtlich der Auswahl der Ein- und Ausgaben verfügt, existieren oftmals mehrere Modelle für ein System [23]. Diese Art der Untersuchung wird als *analytische* Lösung bezeichnet und wird bei Modellen mit einer geringen Komplexität eingesetzt. Jedoch sind die betrachteten Systeme in der Praxis häufig derart komplex, dass analytische Methoden nicht durchführbar sind. In diesen Fällen stellt die *Simulation* eine vielversprechende Alternative dar, die als eine rechnergestützte numerische Auswertungsmethode definiert ist. Im Rahmen einer Simulation werden Daten gesammelt und für eine Schätzung der Charakteristika eines Systems verwendet. [67] Damit kann diese als eine Problemlösungsmethode aufgefasst werden, die durch die Durchführung von Experimenten mit Simulationsmodellen zu Erkenntnissen über das modellierte System führt [46].

Das *Simulationsmodell* umfasst die Nachbildung eines Systems, wobei diese häufig einer vereinfachten Nachbildung der Realität entspricht. Gleichwohl stimmt diese Nachbildung mit dem bereits existenten oder geplanten System hinsichtlich der zu untersuchenden Aspekte überein. Der Begriff des *Simulationslaufs* bezeichnet die Ausführung eines ablauffähigen Modells innerhalb des Simulationsmodells. Die Ausführung eines Laufs wird durch eine festgelegte Parameterkonfiguration sowie durch die Angabe einer zu simulierenden Zeitspanne beeinflusst. Nach der Beendigung eines Simulationslaufes werden die Kennzahlen zur Bewertung des Systems ausgelesen und liefern somit einen Erkenntnisgewinn über das zu untersuchende System. Die gezielte Untersuchung des Modellverhaltens mittels wiederholter

Simulationsläufe wird als ein *Simulationsexperiment* bezeichnet [1]. Der Begriff steht mit den zuvor genannten Begriffen Simulationsmodell und Simulationslauf in Verbindung. Diese Beziehung ist in der Abbildung 2.2 visualisiert. Zunächst sei anzumerken, dass im Vorfeld der Durchführung einer Simulationsstudie Zielgrößen wie der Durchsatz pro Stunde oder die Maschinenauslastung pro Schicht als Projektziele definiert werden. Die Erfüllung dieser Zielgrößen wird im Rahmen des Simulationsexperiments forciert. [46] Hierzu stößt das Simulationsexperiment Änderungen hinsichtlich der Struktur und der Parameterkonfiguration eines Simulationsmodells an und startet den Simulationslauf. Nach Beendigung eines Simulationslaufs werden die Ergebnisse bewertet und darauf basierend wird das Vorgehen wiederholt. Die Parameter werden im Kontext von Simulationsexperimenten als Faktoren bezeichnet und zwischen qualitativ (z. B. Strategien zur Einlagerung von Gütern in einem Lagersystem) und quantitativ (z. B. Anzahl der fahrerlosen Transportsysteme in einem System) unterschieden.

In der Praxis erfolgt die Durchführung der Simulationsexperimente in einem iterativen Prozess, bei dem nach jedem Simulationslauf die Ergebnisse ausgewertet und als Indikator für den Grad der Zielerfüllung dienen. Die Herausforderung bei diesem Prinzip stellt die hohe Anzahl an Variationsmöglichkeiten der Parameterkonfiguration und der möglichen Strukturanpassungen des Systems dar. [61] Jedoch lässt sich der gesamte Raum an möglichen Konfigurationen häufig ohne den Einsatz von Verfahren zur automatischen Simulationsmodellgenerierung nicht vollumfänglich untersuchen [58]. Eine Alternative zum iterativen Vorgehen stellt die Durchführung einer statistischen Versuchsplanung dar, bei der eine Liste durchzuführender Experimente systematisch erarbeitet wird. Diese stößt allerdings bei einer hohen Anzahl an konfigurierbaren Parametern an ihre Grenzen [128].

2.1.2 Klassifikation von Simulationsmodellen

Nachfolgend folgt die Vorstellung einer Möglichkeit zur Klassifikation von Simulationsmodellen. Diese wird eingesetzt, um eine geeignete Simulationsmethodik für die zu untersuchenden Simulationsmodelle im Rahmen der Problemstellungen aus der Produktion und Logistik zu ermitteln. Die Entscheidung erfolgt basierend auf dem zu untersuchenden System sowie den geforderten Zielgrößen.

Nach Law [67] können Simulationsmodelle bezüglich drei Dimensionen klassifiziert werden, die in der Abbildung 2.3 abgebildet sind. Die erste Dimension unterscheidet zwischen *statischen* und *dynamischen* Simulationsmodellen. Ein statisches Simulationsmodell repräsentiert entweder ein System zu einem nicht variablen Zeitpunkt oder Systeme, in denen der zeitliche Verlauf irrelevant ist. Die dynamischen Simulationsmodelle stellen eine Repräsentation von Systemen dar, die sich im Laufe der Simulation verändern. Die zweite Dimension unterscheidet zwischen *deterministischen* und *stochastischen* Simulationsmodellen. Sofern in einem Simulationsmodell keine probabilistischen Einflüsse vorhanden sind, sind das Simulationsmodell als auch dessen Ausgabe deterministisch. Enthält das Simulationsmodell hingegen probabilistische Komponenten, so handelt es sich um ein stochastisches Simulationsmodell.

Die dritte Dimension differenziert zwischen *kontinuierlich* und *diskret*. In kontinuierlichen Simulationsmodellen wird das System durch eine Reihe von Zustandsvariablen repräsentiert, die sich kontinuierlich über die Zeit verändern. Im Gegensatz dazu findet die Veränderung von Zustandsvariablen in diskreten Simulationsmodellen zu fest definierten Zeitpunkten statt. Somit verändert sich das System in einer endlichen Anzahl an Zeitpunkten. Hierbei ist hervorzuheben, dass die Wahl zwischen diskreten und kontinuierlichen Simulationsmodellen, insbesondere von den spezifizierten Zielen einer durchzuführenden Simulationsstudie abhängig ist. Im Rahmen der Verkehrssimulation einer Autobahn bietet sich die diskrete Simulationsmethodik an, falls der Fokus auf der Untersuchung der Bewegung der einzelnen Fahrzeuge liegt. Sollte jedoch die Betrachtung der sich bewegenden Fahrzeuge als Strom fokussiert werden, so erweist sich ein kontinuierliches Simulationsmodell als geeignet. [67]

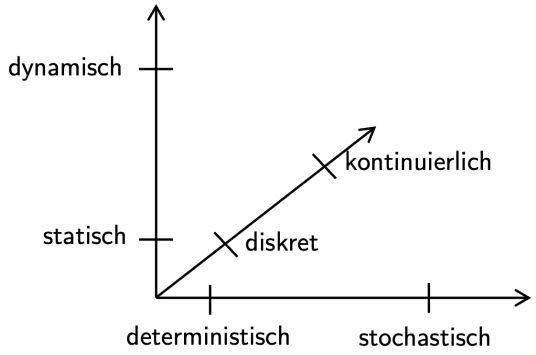


Abbildung 2.3: Dimensionen zur Klassifikation von Simulationsmodelles (Quelle: Eigene Darstellung nach Law [67]).

Im Bereich der Simulationsmodellierung in der Produktion und Logistik werden vorwiegend dynamische, stochastische und diskrete Simulationsmodelle eingesetzt. Die zu modellierenden Systeme sind in der Regel Anlagen (z. B. Fabriken oder Lager) oder Abläufe, die ein dynamisches Verhalten aufweisen [80]. Zudem werden in dieser Domäne die Kennzahlen zur Bewertung von Systemen häufig in Abhängigkeit zu einem Zeitintervall gemessen, wie dem Durchsatz pro Stunde [46]. Weiterhin sind Produktions- und Logistiksysteme häufig probabilistischen Einflüssen wie Maschinenausfällen, Personalfehlzeiten oder Lieferengpässen ausgesetzt. Ebenso werden in diesen Systemen Zustandsänderungen durch eine endliche Anzahl zeitlicher Ereignisse wie die Ankunft von Materialien oder der Beendigung eines Bearbeitungsschritts ausgelöst. Die genannten Charakteristika führen daher in der Regel zur Verwendung der diskreten ereignisorientierten Simulation zur Untersuchung von Produktions- und Logistiksystemen [80].

2.1.3 Ereignisorientierte Simulation

In der ereignisorientierten Simulation (DES) (engl. Discrete-event Simulation) wird ein System mittels eines Simulationsmodells repräsentiert, das sich fortwährend über die Zeit verändert.

Die Veränderungen erfolgen durch die verzögerungsfreie Änderung von Zustandsvariablen des Simulationsmodells. Diese Änderungen erfolgen zu bestimmten Zeitpunkten, die wiederum durch die Ereignisse vorgegeben werden. [67]

Die Zeit ist ein zentrales Element in der ereignisorientierten Simulation. Daher verfügt ein Großteil der Simulationsumgebungen zur ereignisorientierten Simulation über eine Simulationsuhr als Variable, welche die bisher simulierte Zeit beinhaltet. Der Umgang mit der Zeit erfolgt häufig nach dem *Next-event-Time-Advance*-Prinzip, bei dem zu Beginn der Simulation die Simulationsuhr mit dem Wert 0 initialisiert und eine Liste mit Zeitpunkten von zukünftig eintretenden Ereignissen erstellt wird. Die Simulationsuhr wird während der Simulation stets auf den Zeitpunkt des nächst bevorstehenden Ereignisses vorgestellt. Nach Eintreten eines Ereignisses und einer ausgelösten Veränderung im System, wird die geführte Liste der Simulationsuhr mit den bevorstehenden Ereignissen aktualisiert und die Uhr wird vorgestellt. Dies wiederholt sich so lange, bis eine Bedingung eintritt, welche die Simulation terminiert. Nach diesem Ansatz wird die Zeit zwischen zwei Ereignissen, in denen das System inaktiv ist, nicht simuliert, sondern übersprungen. [67]

Die Simulationsumgebungen, welche die ereignisorientierte Simulation implementieren, und die Zeit nach dem Next-event-Time-Advance-Prinzip verwalten, bestehen nach Law [67] häufig aus den zehn folgenden Bestandteilen:

1. Systemzustand, der wiederum aus Zustandsvariablen zur zeitpunktabhängigen Beschreibung des Systems besteht.
2. Simulationsuhr, die als eine Variable mit der bisher simulierten Zeit implementiert ist.
3. Ereignisliste, welche die nachfolgenden Ereignisse einschließlich ihrer Eintrittszeitpunkte umfasst.
4. Zähler für Statistiken, die als Variablen mit statistischen Informationen über die Systemperformanz vorhanden sind.
5. Initialisierungsroutine zur Initialisierung des Simulationsmodells zu Beginn der Simulation.
6. Zeitführungsroutine, die das nächste Ereignis bestimmt und die Simulationsuhr auf den Zeitpunkt des Ereigniseintritts vorstellt.
7. Ereignisroutine, die für zur Verarbeitung eines Ereignisses die entsprechenden Zustandsvariablen des Systems anpasst. Hierfür ist für jedes Ereignis eine separate Ereignisroutine implementiert.
8. Bibliotheksroutinen, welche die Generierung von Wahrscheinlichkeitsverteilungen der Zufallsgrößen verantworten.
9. Reportgenerator, der einen Bericht basierend auf den Werten der Zähler für Statistiken zur Beschreibung der Systemperformanz erzeugt.

10. Hauptprogramm, das die Initialisierungsroutine, Zeitführungsroutine sowie die entsprechenden Ereignisroutinen aufruft. Ferner umfasst das Hauptprogramm die Überprüfung, ob die Simulation beendet ist. Sofern dies zutrifft, ruft das Hauptprogramm den Reportgenerator auf.

Wie in der Beschreibung der Bestandteile bereits erwähnt, stehen diese in logischer Beziehung zueinander und bilden somit einen Kontrollfluss zur ereignisorientierten Simulation eines Systems. Die Abbildung 2.4 fasst den resultierenden Kontrollfluss zusammen. Der zentrale Kern einer ereignisorientierten Simulation ist nach diesem Kontrollfluss das Hauptprogramm, das zu Beginn einer Simulation ausgeführt wird. Dieses Hauptprogramm ruft die Initialisierungsroutine und Zeitführungsroutine auf. Ferner wird gemäß der Ereignisliste ein entsprechendes Ereignis ausgewählt und die entsprechende Routine aufgerufen, die wiederum Bibliotheksroutinen aufruft. Nach der Beendigung der Verarbeitung eines Ereignisses prüft das Hauptprogramm, ob die Simulation beendet ist. Je nach Ergebnis wird das nächste Ereignis im Hauptprogramm ausgewählt oder die Simulation wird mit der Generierung des Berichts abgeschlossen.

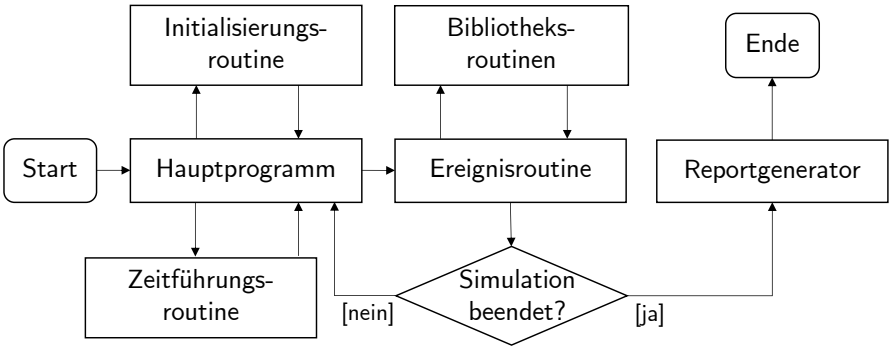


Abbildung 2.4: Kontrollfluss zur Durchführung einer ereignisorientierten Simulation (Quelle: In Anlehnung an Law [67]).

Zur Veranschaulichung wird nachfolgend ein beispielhaftes System (angelehnt an einem Beispiel aus [80]) als ein ereignisorientiertes Simulationsmodell modelliert. Dieses Beispiel umfasst eine Bearbeitungsstation und einen vorgelagerten Puffer, der die Aufträge nach dem Vorgehen First-in-First-out freigibt. Zunächst werden die Ereignisse identifiziert. In diesem Beispiel bilden die Ankunft eines Auftrags und deren Fertigstellung der Verarbeitung an einer Bearbeitungsstation die relevanten Ereignisse. Bei der Ankunft eines Auftrags ist entweder eine Bearbeitungsstation verfügbar, sodass der Auftrag unmittelbar verarbeitet wird, oder die Bearbeitungsstation ist belegt, sodass der Auftrag in den Puffer geschoben wird. Die Fertigstellung eines Auftrags führt zu einer Entnahme eines nachfolgenden Auftrags aus dem Puffer, sofern dieser nicht leer ist. In dem genannten Beispiel stellen die durchschnittliche Durchlaufzeit der Aufträge oder die Auslastung der Bearbeitungsstationen mögliche Untersuchungsziele dar. Die genannten Ziele beeinflussen die Modellierung der Zustandsvariablen, die für die exemplarischen Ziele wie folgt definiert werden:

- Belegungsstatus der Bearbeitungsstationen (z. B. frei oder belegt),
- Anzahl der wartenden Aufträge im Puffer,
- Zeitpunkte der Ankunft und Beendigung der Aufträge und
- Zeitpunkte des Wechsels der Verfügbarkeit der Bearbeitungsstationen.

Das Ereignis der Auftragsankunft und der Fertigstellung eines Auftrags resultieren in einer Anpassung der Zustandsvariablen, die den Belegungsstatus sowie die Anzahl der Aufträge im Puffer umfassen. Ferner werden bei Auftreten der Ereignisse die Zustandsvariablen angepasst, welche die Zeitpunkte der Ankunft und Beendigung eines Auftrags sowie die Zeitpunkte des Wechsels des Belegungsstatus der Bearbeitungsstationen beinhalten. Letztere können zur Berechnung der Auslastung der Bearbeitungsstationen verwendet werden. An dieser Stelle sei hervorzuheben, dass die Ereignisse nicht zwingend die Zustandsvariablen beeinflussen. Dies ist unter anderem der Fall, wenn ein Ereignis die Aktualisierung des Zeitplans zur Durchführung der Simulation herbeiführt.

2.1.4 Agentenbasierte Simulation

Die agentenbasierte Simulation (engl. Agent-based Simulation (ABS)) stellt eine Variation der ereignisorientierten Simulation dar und entwickelte sich in den letzten Jahren zu einer beliebten Simulationsmethodik [24]. Typischerweise bestehen agentenbasierte Simulationsmodelle aus den folgenden Komponenten:

1. Agenten, die über *Eigenschaften* und ein *Verhalten* verfügen,
2. Topologie, welche die Beziehung der Agenten zueinander definiert und eine
3. Umgebung, in der sich die Agenten befinden und miteinander interagieren. [75]

Die Agenten weisen eine Reihe von Charakteristika auf. Während es in der Literatur keine einheitliche Übereinstimmung zu diesen Charakteristika gibt, wird in mehreren Forschungsarbeiten die Fähigkeit des autonomen Handelns genannt. Letzteres erlaubt den Agenten auf Situationen ohne einen fremden Einfluss zu reagieren. Hierbei wird die Reaktion durch das Verhalten des Agenten bestimmt. [75] Nach Macal und North [76, 75] weisen Agenten die folgenden Charakteristika auf:

- eindeutig identifizierbar,
- über eine Menge an Charakteristika und Regeln verfügend, die das Verhalten und die Entscheidungsfindung beeinflussen,
- in sich abgeschlossen,

- autonom handlungsfähig und selbst gesteuert,
- in einer Umgebung lebend und in dieser mit anderen Agenten interagierend,
- fähig andere Agenten zu erkennen und zu unterscheiden,
- erinnerungsfähig,
- zielgerichtet,
- flexibel,
- lernfähig und
- anpassungsfähig.

Der Einsatz der ABS bietet sich nach Law [67] insbesondere in Systemen an, in denen die Entitäten eines Systems

1. in Interaktion mit ihrer Umgebung und anderen Entitäten stehen,
2. ihr Verhalten basierend auf gelerntem Wissen anpassen und
3. Bewegungen im System nicht planen, sondern basierend auf der aktuellen Wahrnehmung der Umgebung vornehmen.

Hierbei fällt auf, dass die genannten Merkmale mit den aufgezählten Charakteristika der Agenten übereinstimmen. Ein exemplarisches geeignetes System stellen Produktionssysteme dar, in denen die Maschinen, Arbeitskräfte und Transportsysteme (durch Agenten repräsentiert) miteinander (z. B. Arbeitskraft bedient Maschine) als auch mit dem System (z. B. Transportsystem beliefert Warenausgang) interagieren. Weiterhin zeichnen sich Produktionssysteme durch unvorhergesehene Ereignisse aus, wie Maschinenausfälle, die eine Agilität und Anpassungsfähigkeit der Arbeitskräfte und Transportsysteme verlangen.

Zur Veranschaulichung der agentenbasierten Simulation wird das Beispiel aus dem vorhergehenden Abschnitt um das Konzept der Agenten erweitert (in Anlehnung an das Beispiel aus [67]). Statt einer einzelnen Bearbeitungsstation werden fünf Stationen betrachtet. Die Auftragsankunft erfolgt weiterhin zufällig, jedoch werden die Aufträge um eine Abfolge, bestehend aus zehn Prozessschritten, erweitert. Hierbei sei zu erwähnen, dass die Prozessschritte an den Bearbeitungsstationen erfolgen, wobei unmittelbar aufeinanderfolgende Prozessschritte an unterschiedlichen Stationen verarbeitet werden. In der Modellierung von agentenbasierten Simulationsmodellen wird in der Regel ein Bottom-up-Ansatz verfolgt, der die Modellierung des Verhaltens der Agenten sowie der Interaktionen zwischen diesen fokussiert [75]. Daher erfolgt für das aktuelle Beispiel die Modellierung ausgehend von den identifizierten Agenten. Im Zuge dessen repräsentiert ein Agent einen Auftrag. Ferner wird angenommen, dass eine Bearbeitungsstation i für die Bearbeitung eines Prozessschrittes eine konstante Zeit von

t_i Minuten mit $i \in \{1,2,3,4,5\}$ benötigt. Zu Beginn ist den Aufträgen beziehungsweise den Agenten die Bearbeitungszeit an den einzelnen Stationen unbekannt, sodass diese eine Dauer von zwei Minuten annehmen. Erst nach dem Verlassen einer Station i ist dem Agenten die Bearbeitungsdauer t_i bekannt. Diesen Erkenntnisgewinn vermerkt der Agent, in dem der Kenntnisstand des Agenten aktualisiert wird. Bei der Suche nach der nächsten Bearbeitungsstation berücksichtigt der Agent das gewonnene Wissen und wählt die Bearbeitungsstation mit der kürzesten Bearbeitungsdauer, sofern diese nicht im vorherigen Schritt besucht wurde und die Liste der Prozessschritte nicht leer ist. Sofern gemäß dem Kenntnisstand des Agenten mehrere Bearbeitungsstationen über die identische Bearbeitungszeit verfügen, erfolgt die Entscheidung zufällig. Damit handeln die Agenten autonom, da sie für die Auswahl der Bearbeitungsstationen verantwortlich sind. Ferner sind sie lernfähig, da sie sich die Bearbeitungszeiten der Stationen merken. In einer Erweiterung können die Agenten um eine Übertragungsfunktion ergänzt werden, die es ihnen erlaubt, ihre Kenntnisse hinsichtlich der Bearbeitungszeiten mit anderen Agenten im System zu teilen.

2.1.5 Bausteinorientierte Modellierung

Die Modellierung von ereignisorientierten Simulationsmodellen kann auf unterschiedliche Weisen erfolgen. Ein Hilfsmittel stellen *Modellierungskonzepte* dar, welche die Sichtweise einer modellierenden Person festlegen und damit Einfluss auf das resultierende Simulationsmodell nehmen. Die Wahl eines Modellierungskonzeptes erfolgt häufig unter Berücksichtigung der Zielsetzung und der Domäne einer Simulationsstudie. Im Bereich der Produktion und Logistik wird häufig das Konzept der *bausteinorientierten Modellierung* gewählt [46]. Ferner wird dieses Konzept von den marktführenden Simulationsumgebungen wie AnyLogic 8 [42] und Plant Simulation[1] unterstützt. Die Simulationsumgebungen zur bausteinorientierten Modellierung umfassen domänenspezifische Bausteinbibliotheken. Die Bausteine dieser Bibliotheken ermöglichen in einer zusammengesetzten Form die Bildung eines Simulationsmodells. So beinhaltet die Simulationsumgebung AnyLogic 8 unter anderem die *Process Modeling*-Bibliothek, die Bausteine zur domänenübergreifenden Modellierung beliebiger Systeme umfasst, oder die *Material Handling*-Bibliothek, welche die bausteinorientierte Modellierung von Produktions- und Logistiksystemen ermöglicht [42]. Sofern die vorhandenen Bausteine einer Simulationsumgebung nicht die Modellierung eines gewünschten Systems ermöglichen, können die Bausteine häufig durch den Einsatz einer Programmierschnittstelle oder durch das Zusammensetzen eigener Bausteinen erweitert werden [46].

Die Bausteine setzen sich wie folgt zusammen [46]:

1. Zuständen und Zustandsübergängen,
2. einer Ablauflogik,

[1] Siemens Plant Simulation Webseite - https://plant-simulation.de/bausteine/

3. Mechanismen zum Datenaustausch zwischen Bausteinen sowie

4. einer Parametrisierung.

Die Parametrisierung erfolgt häufig mittels Eingabemasken, welche die Simulationsumgebungen zur Verfügung stellen. Gutenschwager et al. [46] stellen eine Klassifikation von Simulationsbausteinen vor, welche die Bausteine hinsichtlich der Merkmale Standort, zeitliche Beständigkeit, Beschaffenheit und Aufbau klassifiziert. Die Abbildung 2.5 fasst diese Klassifikation zusammen. Der Standort eines Bausteins kann *stationär* (z. B. Maschine an festem Standort) oder *mobil* (z. B. fahrerlose Transportsysteme zum Transport von Gütern zwischen verschiedenen Standorten) sein. Die zeitliche Beständigkeit eines Bausteins kann in *temporär* (z. B. Rohmateriallieferung eines LKWs) oder *permanent* (z. B. Maschine zur Verarbeitung von Gütern) klassifiziert werden. Ein weiteres Merkmal zur Klassifizierung der Bausteine betrifft die Beschaffenheit, die *physisch* (z. B. Maschine, Werker oder Transportsystem) oder *logisch* (z. B. Steuerung eines Transportsystems oder Stör- und Pausenkonzepte) sein kann. Letztere referenzieren häufig andere Bausteine im System und weisen eine dementsprechend hohe Komplexität auf. Daher werden logische Bausteine häufig durch die Implementierung von Programmen oder dem Einsatz von Entscheidungstabellen umgesetzt. Weiterhin kann ein Baustein in seinem Aufbau *technikorientiert* (z. B. Repräsentation von Fördertechnik, Lagereinrichtungen oder manuelle Werkertätigkeiten) oder *prozessorientiert* (z. B. Erzeugung oder Prüfen von Entitäten) strukturiert sein. Technikorientierte Bausteine verfügen in der Regel über technische Leistungsdaten (z. B. Geschwindigkeiten, Bearbeitungszeiten oder Rüstzeiten) und sind in den gängigen Simulationsumgebungen durch maßstabsgetreue Visualisierungen in Form einer 2D- oder 3D-Visualisierung dargestellt. [46]

Die bausteinorientierte Modellierung ermöglicht eine effiziente Modellierung insofern als die modellierende Person hauptsächlich die Modellbildung, Parametrisierung und Festlegung der Kennzahlen vornimmt [46]. Da die Simulationsumgebungen häufig grafische Benutzeroberflächen zur Verfügung stellen, können die Bausteine oftmals über Ziehen-und-Ablegen in einem Simulationsmodell platziert werden, sodass die Einstiegshürde für Nicht-Fachkräfte der Simulationsmodellierung signifikant reduziert wird. Ferner können Kompositionen von

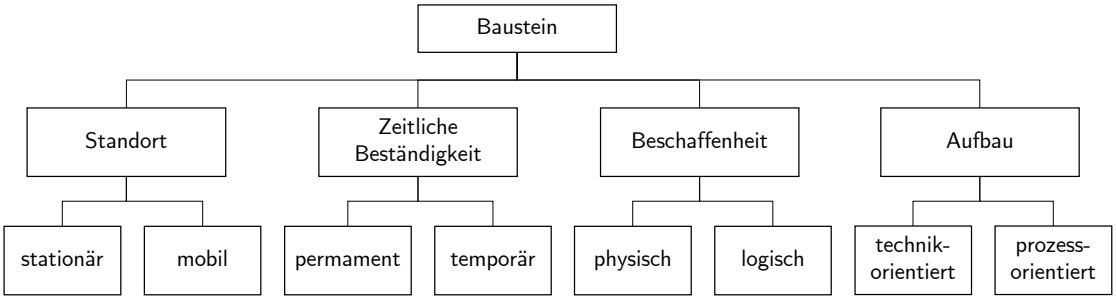


Abbildung 2.5: Dimensionen zur Klassifikation von Simulationsbausteinen (Quelle: Eigene Darstellung nach Gutenschwager [46]).

Bausteinen oder angepasste Bausteine in anderen Projekten häufig wiederverwendet werden, sodass redundante Arbeiten vorgebeugt werden [120]. Gleichwohl ist in der bausteinorientierten Modellierung auf eine korrekte Zusammensetzung der Bausteine zu achten, da nicht alle Bausteine in Kombination kompatibel sind. Weiterhin kann die Auswahl eines geeigneten Bausteins aufgrund einer hohen Anzahl an adäquaten Bausteinen herausfordernd sein. Zudem kann nicht ausgeschlossen werden, dass die Bausteine der Simulationsumgebungen fehlerfrei sind. [46] Trotz der genannten Herausforderungen überwiegen die Vorteile einer bausteinorientierten Modellierung im Bereich der Produktion und Logistik. Daher wird diese Form der Modellierung häufig eingesetzt (vgl. Kapitel 2.3).

2.2 Einsatz der ereignisorientierten Simulation in Produktion und Logistik

In diesem Kapitel wird der Einsatz der ereignisorientierten Simulation in der Produktion und Logistik thematisiert. Wie in von Forschenden durchgeführten Literaturrecherchen festgestellt, wird die Simulation zur Untersuchung einer Vielzahl unterschiedlicher Problemstellungen innerhalb dieser Anwendungsdomäne eingesetzt. Eine Auswahl der betrachteten Literaturrecherchen ist in der Tabelle 2.1 aufgelistet. Jede Zeile der Tabelle umfasst eine durchgeführte Literaturrecherche, mit Nennung der Autorenschaft, dem Jahr der Durchführung, dem betrachteten Zeitraum und der Anzahl der untersuchten Veröffentlichungen. Der Anzahl der betrachteten Forschungsarbeiten innerhalb der Literaturrecherchen kann ein seit mehreren Jahrzehnten vorhandenes Interesse an dem Einsatz der Simulation im Kontext von Produktions- und Logistiksystemen entnommen werden. Ferner stellen Jahangirian et al. [55] fest, dass die ereignisorientierte Simulation die häufigst verwendete Simulationsmethode der im Rahmen ihrer Literaturrecherche 281 untersuchten Forschungsarbeiten ist.

Autorenschaft	Jahr	Zeitraum	Anzahl untersuchter Veröffentlichungen
Smith [110]	2003	1969 - 2002	169
Jahangirian et al. [55]	2010	1997 - 2006	281
Negahban und Smith [86]	2014	2002 - 2013	221
Mourtzis [83]	2020	2010 - 2018	158

Tabelle 2.1: Übersicht über Literaturrecherchen zum Einsatz der Simulation in der Produktion und Logistik.

Im Kontext der Simulation in der Produktion und Logistik ist insbesondere die Bewertung von Varianten eines Systems relevant, um eine Entscheidungsunterstützung bei der Wahl zwischen verschiedenen Lösungen zu ermöglichen. Daher werden im nachfolgenden Unterkapitel logistische Kennzahlen zur Bewertung von Produktions- und Logistiksystemen vorgestellt. Danach folgt die Vorstellung von Vorgehensmodellen, die im Rahmen der Durchführung von

Simulationsstudien eingesetzt werden. Abschließend werden gängige Simulationswerkzeuge und -standardisierungen im Bereich der Produktion und Logistik vorgestellt. Diese Punkte beeinflussen die Konzeption und die Implementierung der in dieser Dissertation entwickelten Technologie.

2.2.1 Kennzahlen zur Bewertung von Produktions- und Logistiksystemen

Mithilfe von Kennzahlen können Planungsvarianten von Produktions- oder Logistiksystemen bewertet werden [125]. Die Kennzahl ist als präziser Zahlenwert definiert, die eine komplexe Realität auf das Wesentliche beschränkt und über quantitative, betriebswirtschaftliche Gesichtspunkte aufklärt [94]. Sie besteht aus einem Kennzahleninhalt, einer Berechnungsformel sowie einem Kennzahlenwert. Der Kennzahleninhalt umfasst den Teil eines Systems, den die Kennzahl beschreibt. Die Berechnungsformel umfasst die Berechnung eines Kennzahlenwerts als absolute oder relative Zahl basierend aus gegebenen Einzelinformationen. [25] In der Regel stellen Kennzahlen Beziehungsgrößen dar, die in Relation zu Sollgrößen stehen. Eine Abweichung einer Kennzahl zu einer vorgegebenen Sollgröße impliziert einen Handlungsbedarf. [94]

Da einzelne Kennzahlen häufig nicht aussagekräftig genug sind, um ein komplexes System zu bewerten, werden in der Praxis eine Menge an Kennzahlen betrachtet, die in Beziehung zueinander stehen und einander ergänzen. Diese Menge wird als ein *Kennzahlensystem* bezeichnet und ermöglicht die vollständige Erfassung einer oder mehrerer Gesichtspunkte eines Systems, um somit eine ganzheitliche Bewertung zu ermöglichen. [114]

Schlüsselbegriff	Beschreibung
Durchsatz	Anzahl der Einheiten, die das System durchlaufen
Bestand	Anzahl der Einheiten, die im System befindlich sind
Durchlaufzeit	Zeitspanne für erfolgreichen Abschluss einer Aufgabe im System
Auslastung	Prozentualer Anteil, in dem sich eine Ressource über einen vorgegebenen Zeitraum in einem bestimmten Zustand befindet (z. B. produzierende Maschine im Zustand „belegt“)
Termin	Anteil termingerechter Fertigstellungen von Aufgaben im System
Ausbeute	Anteil der verwertbaren Einheiten, die Ausgabe des Systems sind

Tabelle 2.2: Kennzahlen zur Bewertung von produktionslogistischen Systemen, gruppiert nach Schlüsselbegriffen (Quelle: Weigert et al. [125]).

Weigert et al. [125] gruppieren relevante Kennzahlen im Bereich der Produktion und Logistik nach Schlüsselbegriffen. Letztere sind in der Tabelle 2.2 aufgelistet. Im Kontext der Logistik sind insbesondere die Schlüsselbegriffe Durchsatz, Bestand und Durchlaufzeit hervorzuheben. Diese sind ein Bestandteil des *Gesetzes von Little*, das als eine fundamentale Gleichung zur

Bewertung logistischer Systeme gilt und wie folgt definiert ist [70]:

$$Bestand = Durchsatz * Durchlaufzeit$$

Weitere Schlüsselbegriffe beziehen sich auf die Auslastung der Ressourcen in einem System, der termingerechten Fertigstellung der Aufgaben sowie der Ausbeute. Letztere umfasst den Anteil der verwertbaren Teile im Vergleich zur Gesamtanzahl produzierter Teil. Daher beziehen die Kennzahlen, die sich diesem Schlüsselbegriff zuordnen lassen, auf die Qualität der Ausgabe des Systems. Die Autorenschaft betont, dass in der Praxis eine Zuweisung der Kennzahlen zu den Schlüsselbegriffen nicht immer eindeutig durchführbar ist. Beispielsweise kann eine Kennzahl, welche die Kapazität eines Systems beschreibt, sowohl dem Durchsatz als auch dem Bestand zugeordnet werden. Ferner weist die Autorenschaft darauf hin, dass gesetzte Systemgrenzen einen Einfluss auf die Kennzahlenwerte nehmen. So unterscheidet sich die Durchlaufzeit, je nachdem, ob die Systemgrenze einen Teilbereich der Produktion oder die gesamte Produktionsanlage umfasst.

Neben Weigert et al. haben auch der Verein Deutscher Ingenieure (VDI) [71, 72, 73] sowie Plümmer und Steinfatt [94] Kennzahlensysteme für die Produktionslogistik entworfen. Der VDI hat in der Richtlinie 4400 Kennzahlensysteme für die unterschiedlichen Aufgabenbereiche der Logistik erarbeitet, namentlich die Beschaffungs-, Produktions und Distributionslogistik. Die Kennzahlen sind in den Kennzahlensystemen in jeweils drei Gruppen unterteilt, welche die Bewertung eines Systems aus leistungs-, kosten- und strukturtechnischer Sicht ermöglichen. Plümmer und Steinfatt gruppieren ebenfalls Kennzahlen gemäß der Aufgabenbereiche der Logistik. Überdies stellt die Autorenschaft Kennzahlen in Bezug auf Transport- und Lagersysteme vor. Letztere beziehen sich beispielsweise auf die Anzahl der bevorrateten Güter und Mitarbeitenden sowie der Lagerbewegungen.

Da bei der Planung von Produktions- und Logistiksystemen unterschiedliche Unternehmensbereiche involviert sind, sollten die Kennzahlen nach Weigert et al. [125] um Metainformationen erweitert werden. Diese Metainformationen sollten etwa den Zeitpunkt oder das Zeitintervall des auslösenden Ereignisses (z. B. Start- oder Endzeitpunkt der Bearbeitung) umfassen. Ferner sollten die Metainformationen den Ort oder die referenzierten Ressource (z. B. Puffereingang) einschließen. Weiterhin kann das Material (z. B. Baugruppe oder Produkt) ebenso einen Teil der Metainformationen darstellen. Diese Ergänzungen ermöglichen ein einheitliches Verständnis zwischen den einzelnen Unternehmensbereichen bezüglich der Kennzahlen. In diesem Zusammenhang sei zu betonen, dass die angestrebten Ziele bei der Planung von Produktions- und Logistiksystemen sich je nach Unternehmensbereich unterscheiden. Während insbesondere die Verkaufsabteilung eine möglichst hohe Termintreue anstrebt, strebt die Produktionsleitung auf einen kontinuierlichen Produktionsfluss mit möglichst wenigen Umrüstungen ab. [80]

2.2.2 Vorgehensmodelle zur Durchführung von Simulationsstudien

Die Durchführung einer Simulationsstudie umfasst mehrere Schritte, die sich von der Datenbeschaffung und -aufbereitung über die Erstellung des Simulationsmodells bis hin zur Durchführung der Simulationsexperimente erstrecken. Aufgrund der Komplexität einer Simulationsstudie werden diese in Unternehmen häufig als eigenständige Projekte durchgeführt. Zur Unterstützung der Durchführung der Simulationsstudien existieren Vorgehensmodelle, welche die einzelnen Schritte einer Simulationsstudie in einer sinnvollen Reihenfolge umfassen. Die Befolgung und Abarbeitung dieser Schritte stellt sicher, dass sämtliche Aspekte einer Simulationsstudie in der korrekten Reihenfolge berücksichtigt werden. Ebenso erwirkt die Berücksichtigung eines Vorgehensmodells häufig einer Verbesserung der Qualität einer Simulationsstudie, da diese eine

1. sorgfältige Projektvorbereitung,
2. konsequente Dokumentation,
3. durchgängige Verifikation und Validierung,
4. kontinuierliche Integration des Auftraggebers sowie
5. systematische Projektdurchführung

bedingen [128]. In diesem Abschnitt werden beispielhafte Vorgehensmodelle zur systematischen Durchführung von Simulationsstudien vorgestellt. Dies erfolgt vor dem Hintergrund, dass die im Rahmen dieser Dissertation entwickelte Technologie in bestehende Vorgehensmodelle integrierbar sein soll. Zunächst werden jedoch grundlegende Phasen einer Simulationsstudie vorgestellt, die laut Banks und Carson [6] übergreifend in den unterschiedlichen Vorgehensmodellen vorhanden sind. Hierbei werden die folgenden aufeinanderfolgenden Phasen genannt:

Aufgabenanalyse Erarbeitung der Aufgaben und Ziele einer Simulationsstudie. Ferner erfolgt die Versuchsplanung und Auswahl einer Simulationsumgebung.

Modellformulierung Beschaffung und Aufbereitung notwendiger Daten sowie Erstellung eines formalen Modells des zu untersuchenden Systems.

Modellimplementierung Überführung des formalen Modells in ein ausführbares Simulationsmodell durch eine Simulationsfachkraft.

Modellüberprüfung Validierung und Verifizierung des ausführbaren Simulationsmodells.

Modellanwendung Ausführung der Simulationsmodelle und Auswertung der Simulationsläufe zur Beantwortung der aufgestellten Fragestellungen.

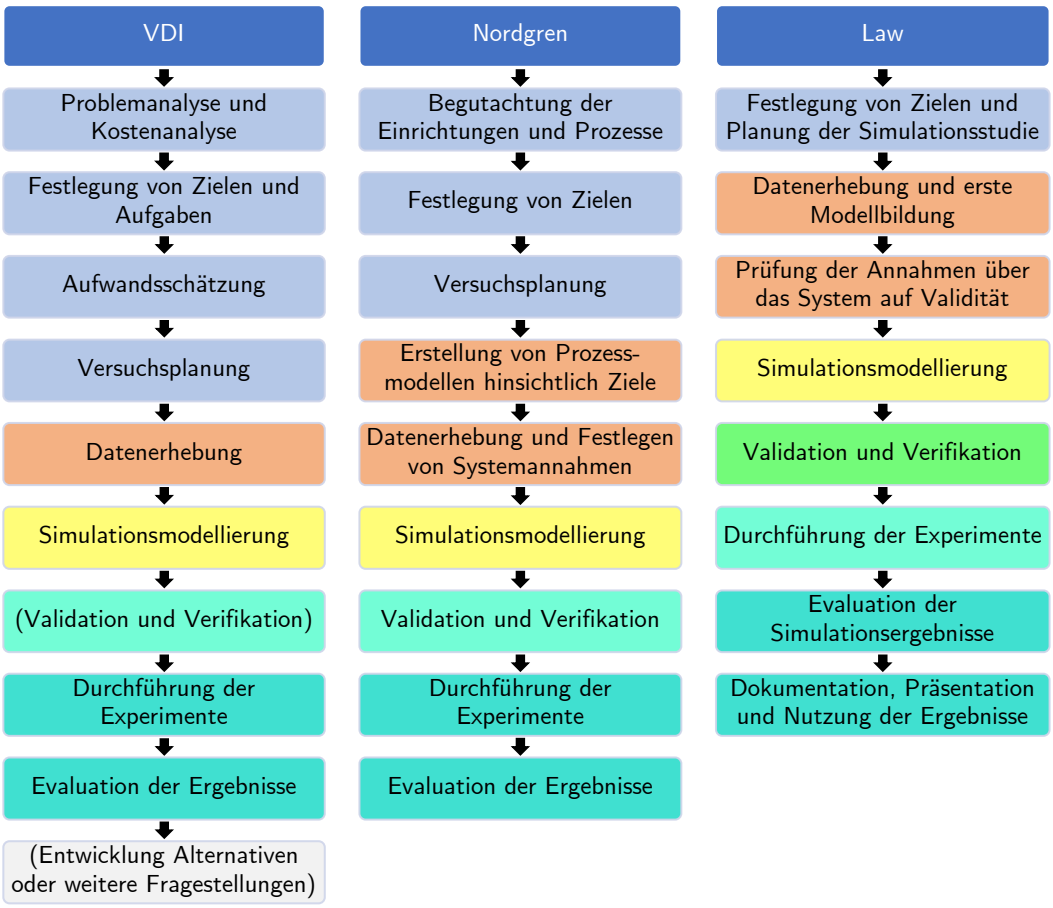


Abbildung 2.6: Vorgehensmodelle zur Durchführung von Simulationsstudien gemäß dem VDI [1], Nordgren [89] und Law [67], wobei die Teilschritte gemäß den von Banks et al. [6] identifizierten Phasen zugeordnet und eingefärbt sind: Aufgabenanalyse in Blau, Modellformulierung in Orange, Modellimplementierung in Gelb, Modellüberprüfung in Grün und Modellanwendung in Türkis.

Nachfolgend werden drei exemplarische Vorgehensmodelle vorgestellt, die in der Abbildung 2.6 dargestellt sind. Die Teilschritte der Vorgehensmodelle sind in der Abbildung den grundlegenden Phasen einer Simulationsstudie mittels farblicher Markierungen zugewiesen. Hierbei sind die Schritte der Aufgabenanalyse in Blau, Modellformulierung in Orange, Modellimplementierung in Gelb, Modellüberprüfung in Grün und Modellanwendung in Türkis gefärbt. Die Abläufe der Vorgehensmodelle sind ähnlich und unterscheiden sich lediglich hinsichtlich der Benennung oder dem Vorhandensein zusätzlicher Schritte. Angesichts dessen werden im Folgen ausschließlich die Unterschiede der einzelnen Vorgehensmodelle thematisiert. In der Abbildung links ist das Vorgehensmodell gemäß dem Verein Deutscher Ingenieure [1] dargestellt. Dieses Vorgehensmodell sieht im Gegensatz zu den anderen Vorgehensmodellen eine Kostenanalyse und Aufwandsschätzung der Simulationsstudie zu Beginn einer Simulationsstudie vor. Dies soll verhindern, dass die Kosten einer Simulationsstudie den

tatsächlichen Nutzen übersteigen. Letzteres ist jedoch häufig schwer zu quantifizieren, da das Aufdecken von Planungsfehler oder der Entdeckung vielversprechenderer Planungsalternativen oftmals nicht zu beziffern ist. Ferner sieht das Vorgehensmodell einen abschließenden, optionalen Schritt zur Untersuchung weiterer Planungsalternativen oder Fragestellungen vor. Das in der Abbildung 2.6 mittige Vorgehensmodell zeigt das Vorgehen nach Nordgren [89]. Die Besonderheit an diesem Vorgehensmodell besteht in der Begutachtung der zu untersuchenden Einrichtungen und Prozesse zu Beginn einer Simulationsstudie. Überdies betont das Vorgehensmodell, dass innerhalb der Phase der Modellformulierung die Prozessmodelle hinsichtlich der Ziele der Simulationsstudie erstellt werden sollen. In der Abbildung 2.6 befindet sich auf der rechten Seite das Vorgehensmodell nach Law [67]. Dieses Vorgehensmodell sieht bereits in der Phase der Modellformulierung eine erste Prüfung auf Validität vor. Ferner sieht das Vorgehensmodell die Erstellung einer Dokumentation und die Präsentation der Ergebnisse als einen Teilschritt vor.

2.2.3 Simulationsumgebungen und Standardisierungen

Die Modellierung von Simulationsmodellen erfolgt häufig innerhalb von Simulationsumgebungen, welche die Modellierung in einer grafischen Oberfläche ermöglichen. Die Auswahl einer Simulationsumgebung erfolgt basierend auf der Problemstellung und den Zielen einer Simulationsstudie. Während ausgewählte Simulationsumgebungen domänenübergreifend für unterschiedliche Problemstellungen einsetzbar sind, ist eine Auswahl auf branchenspezifische Problemstellungen spezialisiert. Zugleich existieren Simulationsumgebungen, die eine domänenübergreifende Modellierung ermöglichen, jedoch branchenspezifische Bausteinbibliotheken zur Verfügung stellen. [46]

Simulations-umgebung	Simulationsmethodik		Sprache	Format	Baustein-orientiert	3D
	Agenten-basiert	Ereignis-orientiert				
AnyLogic	✓	✓	Java, Python	XML	✓	✓
Plant Simulation		✓	SimTalk	XML	✓	✓
Simio	✓	✓	.NET	XML, CSV	✓	✓
FlexSim	✓	✓	FlexScript, C++	XML	✓	✓
ManPy [27]		✓	Python	Python	✓	

Tabelle 2.3: Simulationsumgebungen zur Modellierung von Simulationsmodellen aus dem Bereich der Produktion und Logistik.

Die Tabelle 2.3 zeigt eine Auflistung ausgewählter Simulatiomsumgebungen [31, 27], die mindestens die ereignisorientierte Simulation von Systemen der Produktion und Logistik ermöglichen. Die Simulationsumgebung ManPy ist als Open-Source unter einer LGPL-Lizenz verfügbar, während die Verbleibenden kommerziell vertrieben werden. Wie der Tabelle zu

entnehmen ist, erlauben die aufgeführten Simulationsumgebungen eine bausteinorientierte Modellierung. Weiterhin zeigt die Auflistung, dass das XML-Format ein beliebtes proprietäres Format zur Persistierung der Simulationsmodelle darstellt, wenngleich die Formate nicht anwendungsübergreifend einsetzbar sind. Auch die bausteinorientierte Modellierung sowie das Vorhandensein einer Möglichkeit zur dreidimensionalen Animation der simulierten Systeme wird von einem Großteil der aufgeführten Simulationsumgebungen unterstützt. Jedoch unterscheiden sich die Simulationsumgebungen hinsichtlich der Programmiersprache, in der die Modellierung erfolgt oder ergänzt wird. Während AnyLogic 8 die Mehrzweckprogrammiersprachen Java und Python unterstützt, erfolgt die Programmierung in Plant Simulation und FlexSim in proprietären Sprachen. Ferner unterstützen lediglich die Simulationsumgebungen AnyLogic, Simio und FlexSim die agentenbasierte Simulationsmethodik in ihrer ausgelieferten Form.

Neben der Wahl einer Simulationsumgebung ist auch die Wahl einer Standardisierung zu treffen. Im Kontext der Simulation in der Produktion und Logistik existieren Standardisierungen, die einen möglichst unkomplizierten Datenaustausch zwischen unterschiedlichen Simulationsumgebungen anstreben. Dies ist beispielsweise bei Multi-Level-Simulationen relevant, da hier verschiedene Simulationsmethodiken kombiniert werden, die zur Synchronisation Daten austauschen. Die Multi-Level-Simulation einer Produktionsanlage umfasst etwa eine Maschinenprozesssimulation mittels der Finite-Elemente-Simulation, während der übergeordnete Materialfluss durch die ereignisorientierte Simulation repräsentiert wird. Ein weiterer Anwendungsfall für den Datenaustausch ist die Kommunikation zwischen Simulationen und externen (Software)systemen.

Nachfolgend werden zwei Standardisierungen vorgestellt, die auch in Bezug auf die Generierung von Simulationsmodellen eingesetzt werden. Eine weitverbreitete Standardisierung im Kontext der Produktion und Logistik ist Core Manufacturing Simulation Data (CMSD), den die Arbeitsgruppe *Simulation Interoperability Standards Organization* als ein neutrales, abstrahierendes, und erweiterbares Framework zur Beschreibung von Komponenten eines Produktionssystems (z. B. Werkzeuge, Materialien oder Produktionsprozesse) definiert. Neben dem vereinfachten Datenaustausch verfolgt die Arbeitsgruppe die Vereinfachung der Simulationsmodellerstellung und eine daraus resultierende häufigere Nutzung in der Produktion und Logistik. CMSD besteht aus einer Vielzahl von parametrisierbaren Klassen, welche die entsprechenden Komponenten eines Produktionssystems repräsentieren. Der Standard sieht die Verwendung sowohl der Modellierungssprache Unified Modeling Language (UML) als auch der Auszeichnungssprache Extensible Markup Language (XML) vor, um Produktionssysteme zu beschreiben. Das Listing 2.1 zeigt die Definition eines Förderfließbandes im XML-Format. [99] Fournier [38] nutzt den Standard CMSD zur Übersetzung eines Systems in proprietäre Formate ausgewählter Simulationsumgebungen. Jedoch sehen Bergmann und Straßburger [14] den Standard nicht als ein generisches Austauschformat für Simulationsmodelle, da sich die Simulationsumgebungen in ihrer Funktionsweise und dem Umfang ihrer Funktionen unterscheiden.

```
<Resource identifier="Conveyor:1">
 <Description>ALONG</Description>
 <ResourceType>conveyor</ResourceType>
 <Speed>
   <Value unit="meterPerSecond">0.33</Value>
 </Speed>
</Resource>
```

Listing 2.1: Definition eines Förderbandes mit der Kennung *Conveyor:1* und einer Geschwindigkeit von 0.33 $\frac{\text{m}}{\text{s}}$ nach der Standardisierung CMSD (Beispiel aus Johansson et al. [57]).

Eine weitere Standardisierung zur Beschreibung von Produktionssystemen stellt die grafische Modellierungssprache Systems Modeling Language (SysML) dar. Diese basiert auf der Modellierungssprache UML und nutzt eine Teilmenge dieser [91]. Auch bei dieser Standardisierung gibt es Bestrebungen, eine automatisierte Übersetzung in proprietäre Formate ausgewählter Simulationsumgebungen zu realisieren. Die Autorenschaft Huang et al. untersucht in [51] dieses Vorhaben, mit dem Ziel, die Modellierung eines Systems in unterschiedlichen Softwareanwendungen (unter anderem Simulationsumgebungen) durch eine einzelne Beschreibung mittels SysML und einer automatisierten Übersetzung obsolet zu machen. Daraus resultiert eine inhärente Verifizierung des Simulationsmodells durch die automatisierte Übersetzung. Ferner reduziert die Geläufigkeit der Modellierungssprache UML die Einstiegshürde zur Verwendung der Simulation. Die Abbildung 2.7 zeigt ein *internes Blockdiagramm*, das eine Arbeitsstation mit einem Puffer und zwei Maschinen repräsentiert. Die Elemente dieser Arbeitsstation sind untereinander und mit der Arbeitsstation über Ports (Quadrate mit einem Pfeil) verbunden. Der Puffer und die Maschinen sind wiederum durch separate Blockdiagrammen repräsentiert, die an dieser Stelle nicht dargestellt sind.

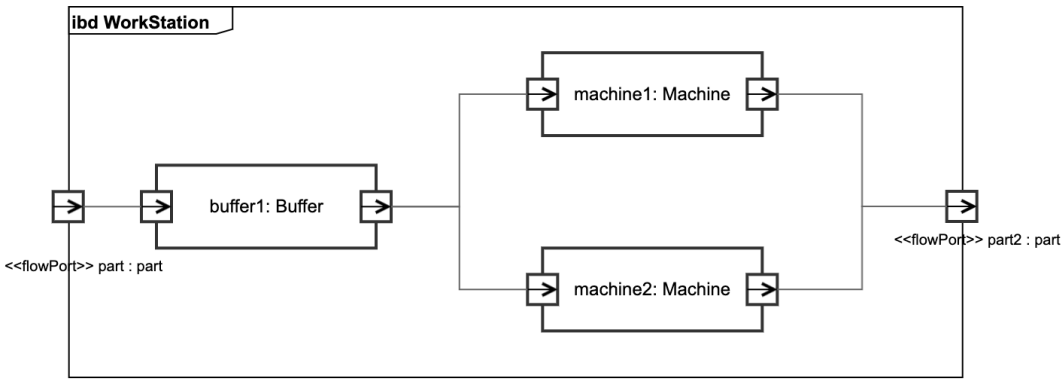


Abbildung 2.7: Beispiel für die Darstellung einer Arbeitsstation mit einem Puffer und zwei Maschinen in einem internen Blockdiagramm in der Modellierungssprache SysML (Quelle: In Anlehnung an Huang et al. [51]).

Abschließend lässt sich festhalten, dass Standardisierungen in Bezug auf die Simulation von

Produktions- und Logistiksystemen zwei Rollen spielen. Zum einen als Austauschformat zwischen Simulationen oder produktionsnahen Softwaresystemen und zum anderen als ein Format zur Beschreibung von Systemen, das in validierter und verifizierter Form zur automatisierten Übersetzung in proprietäre Formate von Simulationsumgebungen genutzt werden kann.

2.3 Automatische Simulationsmodellgenerierung

Die automatische Generierung von Simulationsmodellen ist ein Forschungsgebiet, das Forschungsarbeiten umfasst, welche die zeitaufwendige und fehleranfällige Erstellung oder Anpassung von Simulationsmodellen adressiert und als Lösung (teil)automatisiert [98]. Eine der ersten Definitionen eines Simulationsmodellgenerators stammt aus dem Jahr 1984 von Mathewson [81], der diesen als eine interaktive Anwendung, mit einer in einer generischen Sprache vorliegenden Modelllogik, die für die Übersetzung in den Quellcode einer Simulationssprache verantwortlich ist, definiert. Neyrinck et al. [87] ergänzen, dass die Ansätze zur Simulationsmodellgenerierung auch ohne Kenntnisse der Simulationsmodellierung und Programmierung nutzbar sein sollten.

Das Forschungsgebiet verfügt über eine weitreichende Historie und eine Vielzahl an Veröffentlichungen. Eine der frühen Arbeiten stammt von Lorenz und Schulze aus dem Jahr 1995 [74], bei der Layoutinformationen die Basis für die Generierung der Simulationsmodelle bilden. Einen Überblick über weitere Forschungsansätze geben verschiedene Literaturrecherchen, die jeweils unterschiedliche Schwerpunkte wie die Datengewinnung [98, 7, 122], Schlüsselherausforderungen [39, 13, 115], Strukturvarianz [127] oder Einsatzgebiete in der Praxis [86, 83] setzen.

In den nachfolgenden Unterkapiteln werden zunächst Möglichkeiten zur Klassifikation der Ansätze zur Simulationsmodellgenerierung vorgestellt, ehe Forschungsarbeiten vorgestellt werden, die eine komponentenorientierte Generierung von Simulationsmodellen erlauben.

2.3.1 Klassifikation der Ansätze zur Simulationsmodellgenerieurng

Eine der ersten Klassifikationen von Ansätzen zur Generierung von ereignisorientierten Simulationsmodellen stammt von Eckardt in [35] aus dem Jahr 2002, welche die Ansätze wie folgt unterteilt:

1. parametrische Ansätze,
2. strukturbasierte Ansätze und
3. hybrid-wissensbasierte Ansätze.

Die *parametrischen* Ansätze bauen auf Simulationsbausteinen auf, die im Rahmen der Generierung hinsichtlich einer Parametrisierung selektiert und konfiguriert werden. *Strukturbasierte* Ansätze nutzen Daten (z. B. Layoutinformationen aus CAD-Systemen), welche die Struktur eines abzubildenden Systems beschreiben. Die *hybrid-wissensbasierten* Ansätze setzen auf eine Kombination der parametrischen und strukturbasierten Ansätzen sowie dem Einsatz von Verfahren der künstlichen Intelligenz wie Expertensysteme oder neuronale Netze.

Kriterium	Ausprägung			
Einsatzfall der generierten Simulationsmodelle	planungs-begleitend	betriebsbegleitend		hybrid
Fertigungstyp	Job-Shop	Flow-Shop		Open-Shop
Anwendende	Fachabteilung	Simulations-fachkräfte		Sämtliche
Automatisierungsgrad der Modellerstellung	keine	partiell (Struktur, Verhalten oder beides		voll
Ansatz der Simulationsmodell-erstellung	Interaktion mit Anwendenden	Referenz-modelle	Direkter generischer Modellaufbau	
Unterstützte Phasen der Simulations-studie	Simulations-modell-erstellung	Verifikation & Validierung	Experimente/ Initialisierung	mehrere
Technische Umsetzung der Schnittstelle	textbasiert	tabellenbasiert	XML	Datenbank
Art der fachlichen Umsetzung	layoutbasiert	produktbasiert	prozessbasiert	sonstige/ hybride
Wiederverwendung des Simulation-modells	keine		mehrmalig	

Tabelle 2.4: Klassifikationsmöglichkeiten von Ansätzen zur automatischen Generierung von Simulationsmodellen (Quelle: Straßburger et al. [115]).

Straßburger et al. erweitern in [115] die Klassifikation von Eckardt unter Berücksichtigung der zunehmenden Bedeutung der digitalen Fabrik. Weiterhin merkt die Autorenschaft an, dass die Vielzahl der Veröffentlichungen, die sich durch unterschiedliche Anwendungsfälle und Implementierungen auszeichnen, eine feinere Klassifikation bedingen. Die Tabelle 2.4 zeigt einen Ausschnitt der von Straßburger et al. erweiterten Klassifikation. Zunächst können die Ansätze anhand des Einsatzfalls klassifiziert werden. Hierbei wird unterschieden, ob die Simulationsmodelle für die Planung und/oder den Betrieb eines Systems generiert werden.

Ferner erfolgt die Klassifizierung anhand des Fertigungstyps des zu simulierenden Systems. Die Zielgruppe der Anwendenden wird gemäß der Klassifikation auf drei Gruppen abstrahiert: die Fachabteilung, Simulationsfachkräfte oder keine spezifische Angabe der Anwendenden. Das nächste Merkmal unterscheidet den Automatisierungsgrad hinsichtlich der Erstellung des Simulationsmodells. Ferner wird bei einem partiellen Automatisierungsgrad unterschieden, ob die Struktur und/oder das Verhalten innerhalb der Modellerstellung automatisiert generiert wird. Überdies können die Ansätze in Bezug auf die Art der Simulationsmodellerstellung unterschieden werden. Letztere kann durch eine Interaktion mit den Anwendenden, Referenzmodellen oder durch einen direkten generischen Aufbau erfolgen. Das nächste Merkmal berücksichtigt die Phase der Simulationsstudie, die unterstützt wird. Weiterhin kann anhand der technischen Umsetzung der Schnittstelle klassifiziert werden. Beispielsweise kann diese text- oder tabellenbasiert, über eine Datenbank oder mittels des XML-Formats implementiert sein. Auch die Art der fachlichen Umsetzung stellt ein Merkmal dar. Zum Beispiel kann die fachliche Schnittstelle auf Informationen bezüglich der Prozesse basieren. Ein weiteres Merkmal unterscheidet, ob die generierten Simulationsmodelle ein- oder mehrmalig wiederverwendet werden.

Bergmann et al. nennen in [14] eine weitere Dimension zur Klassifikation der Ansätze, welche die Form der Implementierung betrifft. Die Implementierung kann zum einen *intern* erfolgen, sodass die Simulationsmodellgenerierung innerhalb eines Simulationsmodells in der Simulationsumgebung erfolgt. Die Voraussetzung für solch einen Ansatz ist die Fähigkeit des dynamischen Hinzufügens, Bearbeiten und Entfernen von Simulationsbausteinen in der Simulationsumgebung. Zum anderen kann die Implementierung *extern* erfolgen, sodass die Simulationsmodellgenerierung außerhalb und unabhängig von der Simulationsumgebung ausgeführt wird. Bei diesen Ansätzen wird oftmals Quellcode erzeugt, der das generierte Simulationsmodell in einem proprietären Format beinhaltet, das von der Simulationsumgebung eingelesen wird.

2.3.2 Forschungsstand der komponentenorientierten Ansätze

In diesem Kapitel wird der aktuelle Forschungsstand zur automatischen Generierung von Simulationsmodellen (engl. Automatic Simulation Model Generation (ASMG)) vorgestellt. Aufgrund der Vielzahl vorhandener Forschungsarbeiten, werden nachfolgend ausschließlich die Ansätze vorgestellt, die eine komponentenorientierte Beschreibung des zu generierenden Simulationsmodells erlauben oder die Generierung mittels des Einsatzes von Komponenten durchführen.

Eine der Pionierarbeiten zur komponentenbasierten ASMG stammt von Lee und Zobel [69] zu einer Zeit, als die üblichen Simulationsmodelle ausschließlich mathematische Formeln beinhalteten, die wiederum dynamische Prozesse oder physikalische Systeme repräsentierten. Damit waren keine Metainformationen hinsichtlich der Funktion und dem Einsatzgebiet dieser Formeln bei der Modellierung verfügbar. Dieser Umstand erschwerte die Wiederverwen-

dung von Teilen aus den Simulationsmodellen als auch die Generierung. Angesichts dessen erarbeitete die Autorenschaft eine Repräsentation von Komponenten aus kontinuierlichen und ereignisorientierten Simulationsmodellen zum Aufbau einer Komponentenbibliothek. Diese Repräsentation sieht vor, dass jede Komponente über unterschiedliche Attribute wie eine eindeutige Kennung, Beschreibung und Simulationsmethode (kontinuierlich oder ereignisorientiert) verfügt. Der Simulationsmodellquellcode der Komponenten ist als binär codierter Text vorhanden. Weiterhin erlaubt der Ansatz die Vererbung und Komposition von Komponenten. Die Komponentenbibliothek wird in der Arbeit von Lee und Zobel mittels einer objektorientierten Datenbank realisiert. Letztere beinhaltet die Komponenten sowie existierende und neue Simulationsmodelle. Durch entsprechende Anfragen an die Datenbank in der Sprache OQL, die an SQL angelehnt ist, erfolgt die Generierung der Simulationsmodelle.

In der Arbeit von Arief und Speirs [2] wird ein Unified Modeling Language (UML)-Werkzeug vorgestellt, das auf die Beschreibung von Prozessen in Simulationsmodellen ausgelegt ist. In diesem UML-Werkzeug können sowohl Klassendiagramme, die statische Eigenschaften abbilden, als auch Sequenzdiagramme zur Modellierung der Prozesse erstellt werden. Die UML-Diagramme werden im Werkzeug durch eine komponentenorientierte Datenstruktur repräsentiert, die zur automatisierten Generierung von Simulationsmodellen für die Simulationsumgebung *JavaSim* verwendet wird. Ferner umfasst das Werkzeug eine im Hintergrund durchgeführte Generierung und Ausführung des Simulationsmodells, während das System durch die anwendende Person grafisch modelliert wird. Damit erfolgt eine automatisierte Überprüfung der Anforderungen, die das System erfüllen soll, bereits während der Modellierung.

In der Arbeit von Son et al. [112] wird ein formales, neutrales Format zur Beschreibung von ereignisorientierten Job-Shop Simulationsmodellen erarbeitet. Dieses Format baut auf vorhandenen Komponenten auf, welche die Simulation von verschiedenartigen Szenarien wie dem Materialfluss oder einer Supply Chain ermöglichen. Im Rahmen der Forschungsarbeit werden die Komponenten identifiziert, die für die Beschreibung von Job-Shops relevant sind, und anschließend in Komponenten überführt. Die Komponenten sind in einer Datenbank gespeichert und werden zur Komposition von Simulationsmodellen für die Simulationsumgebungen *Arena* und *ProModel* verwendet, wobei die Komposition auf Basis des neutralen, formalen Formats erfolgt. Letzteres wird in der Modellierungssprache *EXPRESS-G* formuliert und umfasst Informationen zu den Experimenten, Prozessen, Stationen, Kennzahlen oder der Reihenfolgeplanung. Hierbei beeinflusst etwa die Auswahl der Stationen die Komposition des Simulationsmodells insofern als die Anzahl und der Typ (z. B. Bearbeitungsstation oder Puffer) der Stationen variieren.

Die Arbeit von Röhl und Morgenstern [101] widmet sich der Problematik inhomogener Schnittstellen von Komponenten im Kontext der Simulationsmodellierung. Die Autoren identifizieren einen Ansatz aus dem Bereich der Entwicklung von Webservices als einen potenziellen Lösungsansatz. Bei diesem Ansatz werden XML-Schemata eingesetzt, um die Diversität der Schnittstellen unterschiedlicher Implementierungen zu bewältigen. Jedoch setzen die Autoren

nicht auf eine Beschreibung im XML-Format, sondern nutzen Kompositionsstrukturdiagramme der Modellierungssprache UML, die wiederum im XML-Format gespeichert werden. In Kompositionsstrukturdiagrammen werden Komponenten nicht durch direkte Referenzen miteinander verbunden, sondern die Komponenten verfügen über Ports, die eine Schnittstelle zur Kommunikation mit weiteren Komponenten ermöglichen. Die Ports werden durch syntaktische und semantische Beschreibungen ergänzt, die sicherstellen, dass keine willkürlichen Verbindungen unterschiedlicher Ports ermöglicht wird. In einer Fortführung der Forschungsarbeit von Röhl [102] wird der Ansatz um einen Simulationsmodellgenerator für die Simulationsumgebung *James* erweitert. In dieser Dissertation werden die zulässigen Ports von Komponenten durch die Verwendung von Intersektionstypen beschrieben.

Lattner et al. stellen in [66] einen wissensbasierten Ansatz vor, der automatisiert strukturell differente Simulationsmodelle generiert. Die Ausgangsbasis bildet ein bereits existierendes Simulationsmodell. Die Autorenschaft betont, dass die strukturelle Anpassung von Simulationsmodellen im automatisierten Generierungsprozess ein wichtiger Bestandteil ist, da hierdurch Lösungen generiert werden, die möglicherweise von einer modellierenden Person nicht berücksichtigt worden wären. Der entwickelte Ansatz von Lattner et al. besteht aus den folgenden drei Bestandteilen: Wissensbasis, Modelladaption und Simulationsmodellsteuerung. Die *Wissensbasis* enthält die Komponenten sowie die Informationen zu den Beziehungen dieser zueinander, wie hinsichtlich der Vererbung. Die *Modelladaption* ist ein Programm in der Programmiersprache Prolog, das die Generierung der Simulationsmodelle verantwortet. Innerhalb dieses Bestandteils übersetzt ein Konverter eine ermittelte Lösungsvariante in Anweisungen, die wiederum das korrespondierende Simulationsmodell in der Simulationsumgebung Plant Simulation erstellen. Letzteres wird durch feste Funktionen im Prolog-Programm realisiert, die unter anderem eine neue Station einfügen, die Pufferkapazität erhöhen oder eine Station austauschen. Jede Funktion übernimmt hierbei genau eine Aufgabe. Bei dem Ansatz handelt es sich somit einen internen Ansatz, der keine lokalen Simulationsmodelldateien erzeugt. Die *Simulationsmodellsteuerung* kommuniziert über eine Schnittstelle mit der Modelladaption und ist für das Anstoßen der Generierung, Ausführung sowie der Auswertung der Ergebnisse für eine Lösungsvariante verantwortlich. Die Autorenschaft nennt die frühzeitige Erkennung von wenig zufriedenstellenden Lösungsvarianten als eine potenzielle Erweiterung, um die zeitintensive Simulation dieser Varianten zu verhindern.

Die Arbeit von Lattner et al. weist einige Parallelen zu dem Ansatz dieser Dissertation auf. In beiden Arbeiten wird die Strukturvarianz fokussiert und existierende Simulationsmodelle als Ausgangsbasis verwendet. Die Arbeiten unterscheiden sich insofern als dass in der Arbeit von Lattner et al. eine Wissensbasis initial von der anwendenden Person angelegt wird, während in dieser Dissertation die Komponentenbibliothek zur Synthese von Kontrollflussgraphen durch einen Algorithmus automatisiert erzeugt werden kann (vgl. Kapitel 6.1). Weiterhin ist die Anpassung des Simulationsmodells im Ansatz von Lattner et al. durch festgelegte Funktionen in der Modelladaption eingeschränkt. Die Funktionen sind zwar erweiterbar, jedoch muss für jede weitere Operation eine weitere Funktion ergänzt werden. In dieser Dissertation erfolgt das Hinzufügen, Austauschen und Entfernen von Komponenten über die Anpassung der Typspezi-

fikationen der Komponenten (vgl. Kapitel 4.4.2). Weiterhin weist die Autorenschaft darauf hin, dass der Ansatz in der vorgestellten Version eine vollständige Suche über den Lösungsraum ausschließlich für gering komplexe Simulationsmodelle erlaubt. Angesichts dessen setzt die Autorenschaft auf Metaheuristiken, die den Suchraum einschränken. In dieser Dissertation wird ein Syntheseframework verwendet, das eine hohe Anzahl an Lösungsvarianten synthetisieren kann (vgl. Kapitel 5). Dennoch sollte der Einsatz von Metaheuristiken in Anknüpfung an die Ergebnisse dieser Dissertation untersucht werden, da die willkürliche Ausführung einer hohen Anzahl an Simulationsmodellen aufgrund eingeschränkter Ressourcen hinsichtlich der Kosten und der Zeit vermieden werden sollte.

Huang et al. stellen in [52] einen Ansatz zur automatisierten Selektion, Konfiguration und Komposition von Simulationsbausteinen vor. Letztere sind Teil einer Bibliothek, die durch die manuelle Dekomposition von bestehenden Simulationsmodellen aufgebaut wird. Weiterhin enthalten die Simulationsbausteine Informationen über die Funktion und die möglichen Zusammensetzungen mit weiteren Bausteinen. Als Datenstruktur zur Sicherung der Simulationsbausteine wird eine relationale Graphdatenbank verwendet. Zur Selektion und Konfiguration der zu generierenden Simulationsmodelle werden bestehenden Datenquellen der Designphase (z. B. CAD-Systeme), Planungsphase (z. B. Reihenfolgeplanung), Operativphase (z. B. Sensordaten) oder auch frühere Simulationsergebnisse verwendet, die zum Teil durch den Einsatz von Data-Mining-Verfahren zunächst aufbereitet werden. Nach der Generierung werden die generierten Simulationsmodelle automatisiert ausgeführt und die Simulationsmodelle basierend auf den Ergebnissen des Simulationslaufs angepasst. In der Veröffentlichung wird der Ansatz anhand eines Anwendungsfalls aus der Schienenverkehrslogistik evaluiert.

Bergmann und Straßburger fassen in [14] gesammelte Erfahrungen im Umgang mit der Standardisierung Core Manufacturing Simulation Data im Kontext der Simulationsmodellgenerierung zusammen. Die Autorenschaft bestätigt die Eignung der Standardisierung in Bezug auf die Simulationsmodellgenerierung, -initialisierung sowie der Präsentation der Ergebnisse. So nutzen Bergmann und Straßburger die Standardisierung CMSD um ein System mittels der unterschiedlichen Klassen der Standardisierung zu beschreiben und anschließend ein Simulationsmodell zu generieren. Durch den Aufbau der Standardisierung CMSD in Klassen ist der Ansatz inhärent komponentenorientiert. Die Autorenschaft beschreibt den Interpretationsspielraum in der Nutzung bestimmter Klassen als eine Herausforderung, da Systemkomponenten durch unterschiedliche Klassen repräsentiert werden können. Daher erarbeiten Bergmann und Straßburger eine Erweiterung der Standardisierung in Form der Einführung fehlender spezifischer Klassen (z. B. Puffer) und durch die Ergänzung weiterer Eigenschaften (z. B. Kapazität einer Ressource) innerhalb vorhandener Klassen.

Neyrinck et al. entwickeln in [87] einen Ansatz zur Erkundung von Fabriksystemvarianten mittels der Generierung entsprechender Simulationsmodelle, die wiederum aus Komponenten zusammengesetzt werden. Die Komponenten repräsentieren mechanische Komponenten (z. B. lineare Axen, Antriebe oder Getriebe) und enthalten unter anderem Informationen zu den Materialkosten, Schaltplänen oder Maschinenprogrammierungen. Weiterhin enthalten

die Komponenten Simulationsmodelle, die etwa die Kinematik einer Komponente simulieren. Die Beschreibung der Komponenten erfolgt durch Fähigkeiten, die durch die Angabe von Nebenbedingungen ergänzt werden. Zur Generierung eines Simulationsmodells wird eine Problembeschreibung erwartet, welche die geforderten Fähigkeiten sowie Einschränkungen in Form von Nebenbedingungen enthält. Dies hat den Hintergrund, dass die Autorenschaft eine Verwendung des Ansatzes für anwendende Personen ohne Simulationsexpertise anstrebt. Daher erfolgt auch die Auswertung der Simulation automatisiert, sodass die anwendende Person nach der Generierung und Ausführung der Simulationsmodelle ausschließlich eine Zusammenfassung erhält.

Die Ergebnisse von Neyrinck et al. sind in einigen Punkten vergleichbar zu den Ergebnissen der vorliegenden Dissertation. In beiden Forschungsarbeiten erfolgt die Beschreibung der Komponenten durch Fähigkeiten. In dieser Dissertation werden die Fähigkeiten der Komponenten mittels des Einsatzes von Intersektionstypen beschrieben (vgl. Kapitel 4.4.2). Auch eine Berücksichtigung von domänenspezifischen Wissen ist durch den Einsatz der Intersektionstypen oder Techniken des SMT-Constraint-Solvings (vgl. Kapitel 5) möglich. Jedoch unterscheiden sich die Ansätze in Bezug auf die Erstellung der Simulationsmodelle. Die Generierung im Ansatz von Neyrinck et al. umfasst die Vervollständigung lückenhafter XML-Fragmente, welche die Simulationsmodelle repräsentieren. In dieser Dissertation werden hingegen XML-Fragmente von Grund auf synthetisiert. Neyrinck et al. begründen ihr Vorgehen mit einer hohen Rechenzeit, die eine Generierung einer XML-Datei von Grund auf verursacht. In dieser Dissertation wird die Herausforderung durch den Einsatz der komponentenbasierten Softwaresynthese umgangen, welche die Generierung einer komplexen XML-Datei auf verschiedene Komponenten herunterbricht, die XML-Fragmente produzieren.

In der Literaturrecherche von Wenzel et al. [127] wird die Bedeutung der Strukturvarianz im Kontext der Simulationsmodellgenerierung hervorgehoben. Im Rahmen der Veröffentlichung wird die Generierung von Strukturvarianten eines Simulationsmodells mittels des Einsatzes der komponentenbasierten Logiksynthese prototypisch demonstriert. Hierfür wird eine Komponentenbibliothek händisch aufgebaut, welche die Synthese von Strukturvarianten eines Simulationsmodelles einer fiktiven Produktionslinie ermöglicht. Die Autorenschaft nennt den Aufbau einer Komponentenbibliothek (inklusive der Beziehungen der Komponenten zueinander) zur Synthese beliebiger Simulationsmodelle sowie die automatisierte Anpassung von Kontrollstrategien als Forschungslücken. Die vorliegende Dissertation knüpft an die genannten Forschungslücken an, setzt jedoch den Fokus auf die Migration bestehender Simulationsmodelle und nicht auf den Aufbau einer universalen Komponentenbibliothek.

In der Arbeit von Wincheringer et al. [130] wird ein Ansatz vorgestellt, der die Generierung ereignisorientierter Simulationsmodelle einer Matrixproduktion ermöglicht. Auch in dieser Forschungsarbeit strebt die Autorenschaft, ähnlich wie Neyrinck et al. [87], eine Nutzung des Ansatzes auch für Nicht-Simulationsfachkräfte an. Die Generierung erfolgt in den Teilschritten: Generierung, Initialisierung und Adaption eines Simulationsmodells. Die Generierung erfolgt auf Basis von Komponenten, die in der verwendeten Simulationsumgebung *WITNESS*

als Module bereits vorhanden sind und unter anderem Betriebsmittel, Materialpuffer, Transportvorrichtungen oder Prioritätsregeln repräsentieren. Die Initialisierung und Adaption der Komponenten erfolgt über eine Schnittstelle, die wiederum eine dynamische Parametrisierung der Komponenten erlaubt. Die Adaption ermöglicht die Durchführung von Experimenten ohne eine erneute Generierung des Simulationsmodells. Eine mögliche Adaption stellt das Hinzufügen weiterer Betriebsmittel dar. Auch diese Forschungsarbeit stellt die Ergebnisse der Ausführung der Simulation der anwendenden Person in einer zusammengefasster Form zur Verfügung. In der Veröffentlichung betrachtet die Autorenschaft die Matrixproduktion, jedoch sollte der Ansatz aufgrund der Verwendung vordefinierter Module der Simulationsumgebung auf weitere Anwendungsfälle übertragbar sein.

2.3.3 Offene Forschungsfragen

In diesem Abschnitt werden die identifizierten Forschungslücken und Herausforderungen vorgestellt, die aus der durchgeführten Literaturrecherche resultieren. Eine Herausforderung betrifft das Sammeln der erforderlichen Daten aus einer häufig heterogenen Systemlandschaft. Daraus resultiert die Anforderung, dass die Ansätze zur Simulationsmodellgenerierung derart konzeptioniert sein sollten, dass sie mit unterschiedlichen Schnittstellen kompatibel sein sollten [98]. Ferner sind in diesem Zusammenhang unvollständige Daten als eine offene Herausforderung zu nennen [13].

Eine weitere Forschungslücke betrifft die Strukturvarianz, die in den Forschungsarbeiten von Lattner et al. [66] und Wenzel et al. [127] genannt wird. Die Berücksichtigung der Strukturvarianz ist insbesondere im Kontext der Industrie 4.0 von Bedeutung, da die Produktionssysteme über eine hohe Anpassungsfähigkeit verfügen [122]. Daher sollten bei der Planung von Produktionssystemen möglichst viele strukturelle Varianten untersucht werden, um die vielversprechendsten Lösungen in Bezug auf festgelegte Kennzahlen zu ermitteln [127]. Die händische Erstellung aller möglichen Varianten ist häufig aufgrund eingeschränkter Ressourcen hinsichtlich der Kosten und Zeit nicht möglich. Weiterhin ermöglicht die automatisierte Erkundung und Generierung von strukturell unterschiedlichen Planungsvarianten das Finden von Lösungen, die eine modellierende Person unter Umständen nicht berücksichtigt hätte [58, 66].

Eine Veränderung der Struktur eines Produktionssystems bedingt häufig eine Anpassung der Verhaltensmuster (z. B. Kontrollstrategien). Die automatisierte Anpassung der Verhaltensmuster stellt eine Herausforderung im Kontext der Simulationsmodellgenerierung dar [127]. Die Verhaltensmuster sind oftmals derart komplex, dass sie nur durch Algorithmen erfasst werden können [13]. Hierbei stellt nicht nur die Generierung der Verhaltensmuster (z. B. in Form von Algorithmen) eine Herausforderung dar, sondern auch die Verknüpfung einer Strukturvariante und dem dazugehörigen generierten Verhaltensmuster [127].

Eine weitere Forschungslücke ergibt sich durch die Einschränkung vieler ASMG-Ansätze auf

konkrete Problemstellungen oder ausgewählte Lebenszyklen eines Produktionssystems [13, 127]. Weiterhin fokussiert der Großteil der Forschungsansätze die Intralogistik und vernachlässigt insbesondere die externe Logistik (z. B. Supply Chains) [122], die jedoch ein gängiger Untersuchungsgegenstand von Simulationsstudien ist [55].

In der Forschung existieren Ansätze, die eine Beschreibung in Form einer Standardisierung (z. B. Core Manufacturing Simulation Data) zur Generierung in ein proprietäres Format einer Simulationsumgebung nutzen [11, 12]. Die Herausforderung besteht darin, dass die gängigen Simulationsumgebungen oftmals keine Unterstützung für Standardisierungen wie CMSD bieten. Um Letztere zu realisieren, müssten sich die Entwicklungsteams der Simulationsumgebungen zunächst auf eine übergreifende Grundfunktionalität sowie einer gemeinsamen Interpretation der Standardisierungen einigen. Zudem kann das Vorhandensein einer unterstützten Standardisierung in einen ähnlicheren Aufbau der Implementierungen der ASMG-Ansätze resultieren, sodass die Ansätze wiederverwendbar, erweiterbar und vergleichbarer sind. Ferner könnte die Unterstützung von Standardisierungen in den Simulationsumgebungen die Übersetzung in proprietäre Formate vermeiden, sodass die ASMG-Ansätze, die Lösungsvarianten in dem Format einer Standardisierung generieren. Aufgrund der Vielzahl unterschiedlicher Simulationsumgebungen können Standardisierungen zwar kein universales Austauschformat darstellen, jedoch sind sie für den Einsatz in festgelegten Anwendungsdomänen geeignet [14].

2.4 Zusammenfassung

In diesem Kapitel wurden die theoretischen Grundlagen zur Simulation eingeführt. Hierfür wurden die Grundbegriffe der Simulation, unterschiedliche Simulationsmethodiken und das bausteinorientierte Konzept der Modellierung vorgestellt. Anschließend wurde der Einsatz der Simulation im Bereich der Produktion und Logistik thematisiert. Ferner wurden Kennzahlensysteme zur Bewertung von produktionslogistischen Systeme sowie gängige Vorgehensmodelle zur Durchführung von Simulationsstudien gezeigt. Danach wurden gängige Simulationsumgebungen sowie Standardisierungen zur Beschreibung von Produktionssystemen identifiziert. Abschließend wurden die Ergebnisse einer Literaturrecherche zu komponentenorientierten Ansätzen der automatischen Simulationsmodellgenerierung vorgestellt und die identifizierten Forschungslücken zusammengefasst.

Kapitel 3

Synthese

In diesem Kapitel werden die theoretischen Grundlagen zur kombinatorischen Logiksynthese eingeführt, die in dieser Dissertation die Basis für die komponentenbasierte Synthese von ereignisorientierten Simulationsmodellen bildet. In diesem Zusammenhang wird eine verwendete Implementierung der kombinatorischen Logiksynthese vorgestellt, die im Rahmen dieser Dissertation verwendet wird. Abschließend wird in diesem Kapitel die Eignung des Syntheseverfahrens hinsichtlich der in der Einleitung genannten Anforderungen diskutiert.

3.1 Softwaresynthese

Die Softwaresynthese bezeichnet die automatisierte Suche nach einem Programm, das in einer beliebigen Programmiersprache implementiert ist und einem vorgegebenen Wunsch einer anwendenden Person entspricht. Dieser Wunsch kann beispielsweise in Form von Ein- und Ausgabebeispielen, natürlicher Sprache, lückenhaften Programmen, Assertionen oder auch logischen Spezifikationen ausgedrückt werden. Der Suchraum der Softwaresynthese erstreckt sich in der Regel über alle möglichen Programme und unterscheidet sich damit von Programmkompilierung, bei der eine syntaxgeführte Übersetzung eines High-Level-Quellcodes in einen Low-Level-Maschinenquellcode erfolgt. [44]

Die Softwaresynthese blickt auf eine lange Historie zurück, deren Anfänge auf frühe Arbeiten der konstruktiven Mathematik in den 1930er-Jahren zurückführen sind. Die ersten Ansätze sind der *deduktiven* Synthese zuzuordnen und entstanden mit der Entwicklung der ersten automatischen Theorembeweiser, welche die Konstruktion eines Beweises einer gegebenen Spezifikation erlauben [41, 79, 123]. Der Beweis wird anschließend zur Extraktion des korrespondierenden logischen Programms genutzt. Wenige Jahre später folgen *transformationsbasierte* Synthese-Ansätze, bei denen vollständige, abstrakte Spezifikationen, solange transformiert werden, bis sie einem gewünschten Programm entsprechen. Danach folgten *induktive* Syntheseansätze, die durch den Fakt motiviert waren, dass im Kontext der deduktiven

Synthese das Bereitstellen einer vollständigen, formalen Spezifikation häufig vergleichsweise komplex ist. [44] Daher erfolgt die Spezifikation in induktiven Syntheseansätzen beispielsweise durch Ein- und Ausgabebeispiele [108, 18, 116] oder Demonstrationen [109]. Ein weiterer Ansatz der Softwaresynthese nutzt die genetische Programmierung, um die Programme, solange automatisiert zu *evolutionieren*, bis sie einer vorgegebenen Spezifikation entsprechen [64]. Ein alternativer Syntheseansatz erwartet neben der Spezifikation des gewünschten Programms auch die Angabe einer Grammatik, die den Raum der möglichen Lösungen beschreibt. Solar-Lezama stellt in [111] das *SKETCH*-System vor, das lückenhafte Programme, welche die Grammatik darstellen, durch den Einsatz der Softwaresynthese automatisiert gemäß einer gegebenen Spezifikation vervollständigt.

Während die zuvor vorgestellten Ansätze die Programme von Grund auf synthetisieren, beschäftigt sich ein Forschungszweig mit der *komponentenorientierten* Softwaresynthese, die Komponentensammlungen zur Synthese der Programme verwendet. Diese komponentenorientierte Sichtweise findet sich auch in der modernen Softwareentwicklung, die durch einen vermehrten Einsatz von Komponenten oder Modulen charakterisiert ist [78]. Diese Art der Softwareentwicklung resultiert häufig in einer wohlbedachten Modularität und Abstraktion der Softwareanwendung. Die Abstraktion ergibt sich insbesondere durch das Vorhandensein von Schnittstellen der Komponenten. Diese geleistete Vorarbeit erweist sich im Kontext der komponentenbasierten Synthese nutzbringend, da dadurch der Aufwand der Gestaltung und Erstellung einer Komponentensammlung reduziert wird. [16]

Die kombinatorische Logiksynthese ist solch ein komponentenorientierter Syntheseansatz, die der deduktiven, typbasierten Synthese zuzuordnen ist [97]. Diese nutzt die Kombinatorische Logik, um eine Komposition der Komponenten zu realisieren und mittels des Einsatzes von Typen die zulässigen Kompositionen der Komponenten zu beschreiben. Bodik und Jobstmann klassifizieren die kombinatorische Logiksynthese in [20] als einen *funktionalen* Syntheseansatz, der sich durch einen semantischen Kandidatenraum charakterisiert.

Im Rahmen dieser Dissertation wird das Syntheseframework Combinatory Logic Synthesizer (CLS) verwendet, das eine Implementierung der kombinatorischen Logiksynthese mit Intersektionstypen darstellt. Das Syntheseframework wird zur Migration von ereignisorientierten Simulationsmodellen verwendet. Demzufolge folgt in diesem Kapitel eine Einführung in die kombinatorische Logiksynthese und in das verwendete Syntheseframework CLS.

3.2 Kombinatorische Logik

Die Kombinatorische Logik (KL) ist in den 1920er-Jahren im Rahmen der Forschungsarbeit von Moses Schönfinkel [106] entstanden. Ein gemeinsamer Wegbegleiter war dabei stets der Lambda-Kalkül. Die Kombinatorische Logik und der Lambda-Kalkül stellen eine Möglichkeit dar, die Idee von berechenbaren Funktionen oder Algorithmen zu erfassen [19]. Die Besonderheit der KL besteht darin, dass sie die Verwendung von Quantifizierungen obsolet

macht. In diesem Kapitel liegt der Fokus auf der KL, da diese die theoretische Grundlage des verwendeten Syntheseframeworks CLS bildet.

Die KL beschäftigt sich mit Funktionen, die durch Objekte repräsentiert werden, die als *Terme* bezeichnet und zwischen teilbar und unteilbar unterschieden werden. Bei den unteilbaren Termen handelt es sich um *Variablen* oder *Konstanten*. Die Menge der Variablen ist abzählbar unendlich und wird mit Kleinbuchstaben $x, y, z, \ldots$ bezeichnet. Die Konstanten werden auch *Kombinatoren* genannt und mit serifenlosen Großbuchstaben gekennzeichnet. Beispielhafte Kombinatoren der Kombinatorische Logik sind die Kombinatoren I, K und S, die im nachfolgenden Unterkapitel vorgestellt werden. Teilbare Terme entstehen durch die (Funktions)anwendung. In der Definition 1 wird die Menge der KL-Terme bestimmt. [19]

Definition 1 (KL-Terme) Die Menge der Terme in der Kombinatorische Logik wird durch 1. bis 3. induktiv definiert:

1. Wenn Z eine Konstante ist, dann ist Z ein Term.
2. Wenn x eine Variable ist, dann ist x ein Term.
3. Wenn M und N Terme sind, dann ist auch (MN) ein Term. □

In der KL ist ein Term M ein Subterm von N, wenn M in N vorkommt. In der Definition 2 werden die Subterme durch die reflexive, transitive Relation *ist-Subterm-von* auf der Menge der KL-Terme definiert. Es sei hervorzuheben, dass alle Subterme eines Terms auch Terme sind und dass alle teilbaren Terme mehr als einen Subterm haben. Weiterhin ist jeder Term ein Subterm von unendlich vielen Termen. [19]

Definition 2 (Relation *ist-Subterm-von*) Die Relation *ist-Subterm-von* auf der Menge der KL-Terme wird durch 1. bis 5. induktiv definiert:

1. x ist ein Subterm von x
2. Z ist ein Subterm von Z
3. M ist ein Subterm der Terme (NM) und (MN)
4. (NP) ist ein Subterm von (NP)
5. Wenn M ein Subterm von N ist und N ein Subterm von P, dann ist M ein Subterm von P □

3.2.1 Kombinatoren

Ein Kombinator charakterisiert sich durch die Arität (Anzahl der Argumente) und durch das Ergebnis der Anwendung. Ferner gilt, dass alle Argumente eines Kombinators gegeben sein müssen, ehe die Anwendung des Kombinators erfolgt. Das Ergebnis der Anwendung ist durch das *Axiom* des Kombinators festgelegt. Das Axiom besteht aus zwei Termen, die durch ein Symbol (oft $\triangleright$) voneinander getrennt sind. Zur Veranschaulichung wird das Axiom $\mathsf{K}xy \triangleright x$ des Konstantenkombinators K erläutert. Auf der linken Seite befindet sich der Term vor der Anwendung des Kombinators. Die Variablen bilden die Argumente des Kombinators und die Anzahl dieser ergibt die Arität des Kombinators. Hierbei wird angenommen, dass die Variablen unterschiedlich sind. Der Term auf der rechten Seite des Axioms zeigt den resultierenden Term nach der Anwendung des Kombinators. Die Anzahl und Reihenfolge der Variablen verändert sich möglicherweise. Kombinatoren, die im Rahmen ihrer Anwendung keine neuen Variablen einführen, werden als *proper* bezeichnet. Dies ist beispielsweise bei den Basiskombinatoren der Fall, die in der Tabelle 3.1 mit den zugehörigen Axiomen aufgeführt sind. [19]

Name	Axiom	Beschreibung
Identitätskombinator I	$\mathsf{I}x \triangleright x$	Der Term auf der rechten Seite entspricht dem Term auf der linken Seite des Axioms. Somit werden keine Änderungen vorgenommen.
Verschmelzungskombinator S	$\mathsf{S}xyz \triangleright xz(yz)$	Das erste Argument wird auf das zweite Argument angewendet, jedoch wird zuvor das dritte Argument in die beiden ersten Argumente eingesetzt.
Konstantenkombinator K	$\mathsf{K}xy \triangleright x$	Der Kombinator gibt einen festen Wert zurück.

Tabelle 3.1: Beschreibung und Axiome der Basiskombinatoren S, K und I.

Wie den Axiomen der Basiskombinatoren entnommen werden kann, sind die Variablen in der Kombinatorischen Logik durch keinen Quantor gebunden sind. Dieser Fakt gilt als eine der wesentlichen Eigenschaften der Kombinatorische Logik, da sich hierdurch die Substitution von Variablen im Vergleich zum Lambda-Kalkül oder anderen Logiken, in denen Variablen quantifiziert werden, weniger komplex gestaltet [19]. Die Substitution in der Kombinatorische Logik wird im nachfolgenden Kapitel vorgestellt und definiert.

3.2.2 Substitution

Die *Substitution* ermöglicht die Ersetzung von Variablen durch beliebige Terme. Das Beispiel $\mathsf{I}(yz) \triangleright yz$ stellt eine Instanz des Axioms des Identitätskombinators I dar, bei der der Term

(yz) die Variable x ersetzt. Gemäß der Definition 1 wird ein Term durch einen beliebigen Kombinator oder der Anwendung von zwei Termen gebildet. Dementsprechend kann ein Term durch diese beiden Formen substituiert werden. [19] Hierbei kann ein Term auch eine Konstante oder Variable sein. Beispielsweise können im Axiom des Verschmelzungskombinators S die Variablen x und y durch den Kombinator I ersetzt werden, sodass folgende Instanz des Axioms gebildet wird: $\mathsf{SII}z \rhd \mathsf{I}z(\mathsf{I}z)$. In der Definition 3 wird eine Notation und Definition für die Substitution eingeführt [50].

Definition 3 (Substitution von Variablen in KL-Termen) Die Substitution einer Variable x durch einen Term M in einem Term N wird durch $[M/x]N$ formalisiert und durch 1. bis 3. induktiv definiert.

1. $[M/x]x \equiv M$
2. $[M/x]y \equiv y$ für$x \neq y$
3. $[M/x](P_1P_2) \equiv [M/x]P_1[M/x]P_2$ □

Bei der Substitution handelt es sich um eine totale Operation, sodass keine Einschränkungen bezüglich der Form von N und der Anzahl der zu ersetzenden Variablen existieren. Sofern mehrere Substitutionen simultan in einem Schritt durchgeführt werden, wird dies durch eine kommaseparierte Aneinanderreihung dargestellt werden. Hierbei erfolgen die Substitutionen nicht von links nach rechts oder vice versa, sondern in einem Schritt, sodass beispielsweise $[x/y, y/x](xy)$ zu (yx) und nicht etwa (yy) oder (xx) führt. [19]

3.2.3 Reduktion

Im vorherigen Unterkapitel wird die Anwendung von Kombinatoren durch Paare beschrieben, die jeweils einen Term vor und nach der Anwendung zeigen. Nachfolgend wird die *Ein-Schritt-Reduktion* definiert, um eine formale Beschreibung der Auswirkungen der Axiome zu realisieren. Hierfür erfolgt zunächst die Definition des Begriff *Redex* in der Definition 4, der für die nachfolgende Definition der Ein-Schritt-Reduktion relevant ist. [19]

Definition 4 (Redex) Sofern Z ein n-stelliger Kombinator ist, dann wird ein Term der Form $\mathsf{Z}M_1 \ldots M_n$ als *Redex* bezeichnet. Z bildet den Kopf und $M_1 \ldots M_n$ die Argumente des Redex.□

Ein Redex enthält einen oder mehrere Kombinatoren, wobei $M_1 \ldots M_n$ beliebige aber feste Terme repräsentieren. Der Kombinator Z kann mehrfach auftreten, jedoch mindestens einmal als Kopf des Redex. Damit kann nachfolgend die Ein-Schritt-Reduktion in der Definition 5 eingeführt werden. [19]

Definition 5 (Ein-Schritt-Reduktion) Sei Z ein Kombinator mit $\mathsf{Z} \in \{\mathsf{S},\mathsf{K},\mathsf{I}\}$ mit dem Axiom $\mathsf{Z}x_1 \ldots x_n \rhd P$. Sofern N ein Term mit einem Subterm der Form $\mathsf{Z}M_1 \ldots M_n$ und N' ein Term,

der $[x_1/M_1 \ldots, x_n/M_n]P$ als Subterm enthält, so wurde N *ein-Schritt-reduziert* zu N' (gekennzeichnet durch $N \rhd_1 N'$). □

Nachfolgend wird anhand des Axioms des Verschmelzungskombinators S die Ein-Schritt-Reduktion demonstriert. Gemäß der Definition 5 nimmt der Term xyz (vgl. linke Seite des Axioms) die Rolle von M ein, während der Term $xz(yz)$ (vgl. rechte Seite des Axioms) die Rolle von P einnimmt. Hierdurch ergibt sich N als $Sxyz$ und N' als $[x/x, y/y, z/z]xz(yz)$. Gemäß der Definition der Substitution gilt $[x/x, y/y, z/z]xz(yz) \equiv xz(yz)$, wobei $xz(yz)$ dem Term auf der rechten Seite des Axioms entspricht. Damit gilt, dass $\mathsf{S}xyz \rhd_1 xy(yz)$.

Definition 6 (Schwache Reduktion) Der reflexive, transitive Abschluss der Relation der *Ein-Schritt-Reduktion* wird als *schwache Reduktion* bezeichnet und mit $\rhd_w$ notiert. □

Durch den reflexiven Abschluss der Ein-Schritt-Reduktion-Relation gilt bei der schwachen Reduktion, dass jeder Term sich selbst schwach-reduziert, also $M \rhd_w M$ gilt. Durch den transitiven Abschluss gilt zudem, dass wenn $M \rhd_1 N$ und $N \rhd_1 P$ gilt, dann auch M schwach-reduziert zu P ($M \rhd_w P$) werden kann. Hierbei kann die Anzahl der Zwischenschritte zwischen null hin zu einer beliebigen endlichen Zahl variieren. Somit stellt jede Ein-Schritt-Reduktion $M \rhd_1 N$ auch eine schwache Reduktion $M \rhd_w N$ dar. Die Reduktion wird als schwach bezeichnet, da diese ausschließlich Reduktionen umfasst, bei der die Kombinatoren die exakte Anzahl der erwarteten Argumente erhalten. [19] Beispielsweise kann daher der Term SMN nicht schwach-reduziert werden, da das Axiom des Kombinators S drei Argumente vorsieht.

3.2.4 SK-Kalkül

Eine nicht-leere Menge mit einer endlichen Anzahl an Kombinatoren wird als *Basis* bezeichnet. Eine bedeutende Basis umfasst ausschließlich die Kombinatoren S und K und ist als SK-Kalkül bekannt. Diese Basis erfüllt die Eigenschaft der *kombinatorischen Abgeschlossenheit,* die in der Definition 7 definiert ist. Letztere bezieht sich auf die Fragestellung, ob aus den vorhandenen Kombinatoren einer Basis alle möglichen Kombinatoren zusammengesetzt werden können. Dies gilt für die Basis mit den Kombinatoren S und K. [19]

Definition 7 (Kombinatorische Abgeschlossenheit einer Basis) Eine Basis wird als *kombinatorisch abgeschlossen* bezeichnet, sofern für jede beliebige Funktion f gilt, dass $f x_1 \ldots x_n \rhd_w M$, wobei M ein Term ist, der aus den Variablen $x_1 \ldots x_n$ gebaut wurde, und ein Kombinator Z ein Teil der Basis ist, sodass $\mathsf{Z}x_1 \ldots x_n \rhd_w M$ gilt. □

Die Basis mit den Kombinatoren S und K ist kombinatorisch abgeschlossen und wird als SK-Kalkül bezeichnet [19]. Weiterhin ist das SK-Kalkül äquivalent zum Lambda-Kalkül (jeweils ungetypt als auch mit simplen Typen), sodass hiermit gezeigt werden kann, dass das Inhabitationsproblem mit dieser speziellen Basis PSPACE-vollständig ist [97].

3.3 Getypte Kombinatorische Logik mit Intersektionstypen

Die Definition einer Funktion erfolgt häufig durch die Angabe der Typen der Ein- und Ausgabe, so erwartet etwa die quadratische Funktion sowohl eine Ein- und als auch Ausgabe vom Typ Integer. Eine Funktion, die auf den festgelegten Zahlenwert prüft, erhält eine Eingabe vom Typ Integer, während die Ausgabe vom Typ Boolean ist. [50] Gleichermaßen sind in modernen Programmiersprachen die Typen ein wesentlicher Bestandteil. Auch Programmiersprachen wie JavaScript, in denen die Typen nicht explizit im Quellcode angeben werden, verfügen häufig über unterliegende Typsystem. Beispielsweise erfolgt die Auswertung der Typen in der Programmiersprache JavaScript zur Laufzeit. [47]

In diesem Unterkapitel wird die vorgestellte Kombinatorische Logik um Typen erweitert. Zunächst werden in der Definition 8 *primitive Typen* (engl. *Simple Types*) eingeführt. Beispiele für primitive Typen sind der Typ N, der die Menge der natürlichen Zahlen umfasst, oder der Typ H, der die Menge mit den booleschen Werten *wahr* und *falsch* umfasst. [19]

Definition 8 (Primitive Typen in der KL) Sei $\mathbb{P}$ eine nicht-leere Menge an Basistypen und der Typkonstruktur $\rightarrow$ eine binäre Funktion auf Typen. Dann wird die Menge der primitiven Typen durch 1. und 2. induktiv definiert:

1. Wenn $\alpha \in \mathbb{P}$, dann ist α ein primitiver Typ
2. Wenn σ und τ primitive Typen sind, dann ist $(\sigma \rightarrow \tau)$ ein primitiver Typ □

Der Typkonstruktor $(\sigma \rightarrow \tau)$ formalisiert eine Menge von Funktionen, die ein Argument des Typs σ erwarten und auf einen Wert des Typs τ abbilden. Der Typausdruck $(\mathsf{N} \rightarrow \mathsf{H})$ umfasst etwa Funktionen, die eine natürliche Zahl als Argument erwarten und auf einen booleschen Wahrheitswert abbilden. Der Ausdruck $((\mathsf{N} \rightarrow \mathsf{N}) \rightarrow \mathsf{H})$ formalisiert Funktionen, die als Argument eine Funktion, die eine natürliche Zahl auf eine natürliche Zahl abbildet, erwarten und auf einen booleschen Wahrheitswert abbilden. [50]

Während in der Definition 1 Terme für die ungetypte Kombinatorische Logik definiert werden, folgt nun in Definition 9 die Definition der Terme für die getypte Kombinatorische Logik [19].

Definition 9 (Getypte KL-Terme) Die Menge der getypten Terme in der Kombinatorische Logik wird induktiv durch 1. und 2. definiert:

1. Wenn M ein atomarer Term ist und τ ein Typ, dann ist $M : \tau$ ein getypter Term
2. Wenn $M : \tau \rightarrow \sigma$ und $N : \tau$ getypte Terme sind, dann ist $(M : \tau \rightarrow \sigma N : \tau) : \sigma$ ein getypter Term □

Im Nachfolgenden werden Typvariablen mit griechischen Kleinbuchstaben $\tau, \sigma, \ldots$ und Mengen an getypten KL-Termen mit griechischen Großbuchstaben $\Gamma, \Delta, \ldots$ bezeichnet. Weiterhin

werden Ausdrücke der Form $(M : \tau \to \sigma N : \tau) : \sigma$ mit $MN : \sigma$ abgekürzt, sofern die Typen der Terme M und N bekannt sind.

3.3.1 Intersektionstypen

In der bisherigen Definition der KL-Terme mit simplen Typen ist den Termen genau ein Typ zugewiesen. Jedoch kann das gewünschte Verhalten nicht immer durch einen primitiven Typen beschrieben werden. Im Folgenden werden Intersektionstypen eingeführt, die es erlauben, dass Terme über eine Intersektion von Typen verfügen. Damit kann das Verhalten eines Terms in einer komplexeren, aber auch ausdrucksstärkeren Art beschrieben werden. Weiterhin ermöglicht die Verwendung von Intersektionstypen eine eindeutige Typspezifikation der Terme in ihrer Typumgebung. [97] Eine Definition der Intersektionstypen findet sich in Definition 10 [19].

Definition 10 (Intersektionstypen) Sei $\mathbb{P}$ eine nicht leere Menge an Typvariablen, ω eine spezielle Typkonstante, und $\to$ und $\cap$ Typkonstruktoren. Die Menge der Intersektionstypen wird durch 1. bis 3. induktiv definiert.

1. Wenn $\alpha \in \mathbb{P}$, dann ist α ein Intersektionstyp
2. ω ist ein Intersektionstyp
3. Wenn τ und σ Intersektionstypen sind, dann sind $(\tau \to \sigma)$ und $(\tau \cap \sigma)$ ebenfalls Intersektionstypen □

Der Typausdruck $x : \sigma \cap \tau$ sagt aus, dass x sowohl der Typ σ als auch der Typ τ zugewiesen ist. Die Intersektion von zwei Intersektionstypen ist kommutativ ($\tau \cap \sigma = \sigma \cap \tau$), assoziativ ($(\tau \cap \sigma) \cap \rho = \tau \cap (\sigma \cap \rho)$) und idempotent ($\tau \cap \tau = \tau$). Darüber hinaus gilt, dass der Typkonstruktor rechtsassoziativ ist und $\cap$ stärker als $\to$ bindet. [97]

3.3.2 Subtyping

Das Konzept des Subtypings stellt eine Erweiterung des Typsystems und der Typinferenz in Programmiersprachen dar, das die Ausdrucksstärke eines Typsystems insofern erhöht, als ein Ausdruck über mehrere kontextabhängige Typen verfügt [96]. Beim Subtyping handelt es sich um eine Form des Polymorphismus. Das Subtyping wird häufig durch das Vorhandensein einer Ordnungsrelation auf den Typen und Regeln hinsichtlich der Subsumtion realisiert. Hierdurch wird ermöglicht, dass ein Ausdruck vom Typ τ auch einen Typ τ' haben kann, sofern τ' in der Ordnungsrelation *größer* als τ ist [96]. Ferner ermöglicht das Subtyping, dass der Typ eines Terms ohne eine separate Ableitung der Typzuweisung geändert werden kann [19]. Rehof definiert in [97] eine transitive, reflexive Relation des Subtypings für Intersektionstypen (gekennzeichnet mit $\leq$) durch folgende Typinferenz Axiome und Regeln [97]:

- $\sigma \leq \omega, \omega \leq \omega \rightarrow \omega$
- $\sigma \cap \tau \leq \sigma, \sigma \cap \tau \leq \tau$
- $\sigma \leq \sigma \cap \sigma$
- $(\sigma \rightarrow \tau) \cap (\sigma \rightarrow \rho) \leq \sigma \rightarrow \tau \cap \rho$
- $\sigma \leq \sigma' \wedge \tau \leq \tau' \Rightarrow \sigma \cap \tau \leq \sigma' \cap \tau'$
- $\sigma \leq \sigma' \wedge \tau \leq \tau' \Rightarrow \sigma' \rightarrow \tau \leq \sigma \rightarrow \tau'$

Sofern $\sigma \leq \rho$ und $\rho \leq \sigma$ gilt, sind σ und ρ gleich. Aus den Axiomen des Subtypings können zudem folgende distributive Eigenschaften gefolgert werden: [97]

- $(\sigma \rightarrow \tau) \cap (\sigma \rightarrow \rho) = \sigma \rightarrow (\tau \cap \rho)$
- $(\sigma \rightarrow \tau) \cap (\sigma' \rightarrow \tau') \leq (\sigma \cap \sigma') \rightarrow (\tau \cap \tau')$

Moderne Programmiersprachen nutzen das Konzept des Subtypings häufig zur impliziten Umwandlung von Datentypen, sodass etwa die Konvertierung einer ganzen Zahl des Typs `Integer` in eine Gleitkommazahl des Typs `Real` dadurch möglich ist, dass der Typ `Integer` als ein Subtyp von `Real` definiert ist. Auch in nicht klassischen Anwendungsbereichen wie in der Programmanalyse, bei der automatisiert Eigenschaften von getypten Programmen aus dem Quellcode unter Nutzung eines Typinferenz-Algorithmus extrahiert werden, findet das Subtyping Verwendung. [96]

3.3.3 Inhabitation

Die Inhabitation in der kombinatorischen Logik bildet die Grundlage für die automatisierte Komposition von Softwarekomponenten in der kombinatorischen Logiksynthese. Eine Typzuweisung der Form $\Gamma \vdash e : \tau$ drückt aus, dass ein Term e durch die Komposition von Kombinatoren, die Teil einer Basis (nachfolgend auch als Repositorium bezeichnet) Γ sind, konstruiert werden kann. Die Fragestellung

> „Gegeben sei ein Typ A, gibt es einen KL-Term e mit $\Gamma \vdash_s e : A$?“

bezeichnet das Entscheidungsproblem der Inhabitation für eine feste Basis Γ. Sofern ein Term e mit $\Gamma \vdash_s e : A$ existiert, wird dieser als *Inhabitant* in A bezeichnet. Das S steht für die Typsubstitution. Das Inhabitationsprädikat sowie das dazugehörige Entscheidungsproblem der Inhabitation werden mit $\Gamma \vdash ? : A$ abgekürzt. In der *relativierten* Inhabitation wird das

Entscheidungsproblem, um ein veränderliches Repositorium Γ ergänzt, sodass das Entscheidungsproblem wie folgt lautet: [97]

„Gegeben sei ein Repositorium Γ und ein Typ A, gibt es einen KL-Term e mit $\Gamma \vdash_s e : A$?“.

Im Kontext der Inhabitation spielt die Basis, welche die Kombinatoren S und K beinhaltet, eine bedeutende Rolle, da diese mit den Typregeln der Kombinatorische Logik ein System bildet, das unter der Propositionen-als-Typen Korrespondenz zu einer propositionalen intuitionistischen im Hilbert-Stil formulierten Logik korrespondiert. Diese Logik zeichnet sich dadurch aus, dass diese ausschließlich auf zwei Prinzipien der Deduktion basiert, nämlich der Substitution von Formeln in Axiomsschemata (var) und Modus Ponens ($\rightarrow$E). Unter dem Curry-Howard-Isomorphismus korrespondiert das Inhabitationsproblem zum Problem der Beweisbarkeit in der propositionalen Logik und ein Inhabitant e nimmt die Rolle eines Beweises eines Satzes A ein. Nach dem Theorem von Statman ist die Beweisbarkeit in einer propositionalen, intuitionistischen Logik PSPACE-vollständig. Durch den Curry-Howard-Isomorphismus gilt dies auch für das Inhabitationsproblem im Lambda-Kalkül mit primitiven Typen. Und durch die Äquivalenz des SK-Kalküls und des Lambda-Kalküls mit primitiven Typen gilt dies auch für das Inhabitationsproblem des SK-Kalküls. [97]

3.4 Combinatory Logic Synthesizer – CLS

Das Syntheseframework Combinatory Logic Synthesizer (CLS) stellt eine Implementierung, der zuvor vorgestellten kombinatorischen Logiksynthese dar, die in verschiedenen Versionen verfügbar ist. Die Versionen unterscheiden sich hinsichtlich der Programmiersprache und als auch der Einsatzzwecke. So existiert etwa die Version CLS-F# [34], die insbesondere für die Untersuchung unterschiedlicher Suchstrategien im Kontext der Inhabitation und hinsichtlich der Optimierung der Performanz eingesetzt wird. Ferner existiert eine Version CLS-Scala [15], die in der Programmiersprache Scala vorliegt und einen Einsatz in realen Problemstellungen ermöglicht. Letzteres ermöglicht das Framework durch folgende Eigenschaften [15]:

- Einbettung einer domänenspezifischen Sprache zur Entwicklung von Komponenten,
- Automatisierung der Übersetzung und Ausführung von applikativen Termen,
- Programmiersprachen-unabhängige Meta-Programmierung,
- Integrierbarkeit und Synthese von Softwareanwendungen, die auf der Java-Virtual-Machine laufen,
- Debugging durch eine Visualisierung des Syntheseprozesses.

Im Rahmen dieser Dissertation wird die CLS-Scala Version verwendet, daher bezeichnet nachfolgend CLS die Implementierung in der Programmiersprache Scala. Ferner wird im Folgenden der Begriff der Komponente statt des Kombinators verwendet, sofern dieser in einem anwendungsbezogenen Kontext verwendet wird. Weiterhin werden die Begriffe Komponentensammlung und Repositorium austauschbar verwendet. Die Synthese mit dem Syntheseframework Combinatory Logic Synthesizer lässt sich gemäß den Dimensionen der Programmsynthese (vgl. [43]) wie folgt klassifizieren [132]: Die Angabe des Wunsches des Anwendenden erfolgt durch die Angabe eines Syntheseziels, das in Form eines Ziel-Intersektionstypen vorliegt. Das domänenspezifische Wissen wird durch die semantischen Typspezifikationen und den entsprechenden Implementierungsdetails der Komponenten ausgedrückt. Der Suchraum wird durch die möglichen Kompositionen der Komponenten definiert und die Suchstrategie durch den Inhabitationsalgorithmus des CLS-Frameworks implementiert.

Nachfolgend wird anhand eines Beispiels der Aufbau einer Komponentensammlung demonstriert. Anschließend wird gezeigt, wie die Komponentensammlung unter Angabe eines Ziel-Intersektionstypen zur komponentenbasierten Synthese von Lösungen verwendet wird, ehe eine Diskussion der Eignung der kombinatorischen Logiksynthese und des Syntheseframeworks CLS in Bezug auf die Generierung von strukturell unterschiedlichen Simulationsmodellen und den Kontrollstrategien folgt.

3.4.1 Spezifikation der Komponenten

Eine Komponente wird in dem Syntheseframework CLS anhand mehrerer Merkmale charakterisiert. So verfügen die Komponenten über einen eindeutigen Namen sowie einer `apply`-Methode. Letztere beinhaltet die Implementierungsdetails, welche die Ausgabe der Komponente produzieren. Die Argumente und der Rückgabetyp dieser `apply`-Methode bestimmen den *nativen* Typen der Komponente. Die nativen Typen werden somit aus der Implementierungssprache abgeleitet. Diese dient neben dem *semantischen* Typen zur Beschreibung einer Komponente. Der semantische Typ ermöglicht insbesondere die Berücksichtigung von domänenspezifischen Wissen. Die Typspezifikation einer Komponente in CLS setzt sich aus der Intersektion des nativen und semantischen Typs zusammen und erlaubt damit eine Beschreibung des Verhaltens.

Das Listing 3.1 zeigt eine beispielhafte Komponente, die durch die Annotation `@combinator` als solch eine gekennzeichnet wird. Der Name des Objekts `cuttingMachineMidLevel` (vgl. Zeile 1) bildet gleichzeitig den Namen der Komponente. Der Rückgabetyp `CuttingMachine` der `apply`-Methode (vgl. Zeile 2) bildet alleinig den nativen Typen der Komponente, da die `apply`-Methode keine Argumente erwartet. Innerhalb dieser Methode wird ein neues Objekt der Klasse `Cuttingmachine` instantiiert und zurückgegeben. An dieser Stelle sei zu erwähnen, dass die Klassen `CuttingMachineMidLevel` und `CuttingMachineHighLevel` von einer Elternklasse `CuttingMachine` abgeleitet sind. Der semantische Typ der Komponente wird in der Variable `semanticType` definiert (vgl. Zeile 3). Eine Intersektion im semantischen Typen

```
1 @combinator object cuttingMachineMidLevel {
2   def apply: CuttingMachine = new CuttingMachineMidLevel
3   val semanticType: Type = Machine :&: IsCutting :&: MidLevel
4 }
```

Listing 3.1: Implementierung einer Komponente zur Produktion einer Schneidemaschine mittlerer Leistungsstufe. In der Zeile 2 finden sich die Implementierungsdetails und in der Zeile 3 der semantische Typ. Die Annotation `@combinator` signalisiert, dass es sich bei diesem Objekt um eine Komponente handelt. Der Operator `:&:` im semantischen Typen ermöglicht die Intersektion von Typen.

erfolgt in der Implementierung einer Komponente durch die Verwendung des Operators `:&:`, während der Funktionsoperator durch den Operator `=>:` repräsentiert wird. In dem Beispiel besteht diese aus einer Intersektion der Typen *Machine*, *IsCutting* und *MidLevel*. Dadurch wird das domänenspezifische Wissen ausgedrückt, dass die Komponente eine Maschine produziert, die schneidet und einer mittleren Leistungsstufe zuzuordnen ist. Es sei hervorzuheben, dass der Inhabitationsalgorithmus des CLS-Frameworks die Komponente selektiert, sofern etwa eine Komponente des Typs $Machine \cap IsCutting \cap MidLevel$ oder $Machine \cap IsCutting$ gefordert wird. Dies wird durch die Subtyping-Relation für die Intersektionstypen ermöglicht. In der Praxis ermöglicht dies eine Unterspezifizierung der Typen, die wiederum die Bildung von Varianten erlaubt. Beispielsweise können mittels des Typs $Machine \cap IsCutting$ die Komponenten zur Produktion von Maschinen gefordert werden, ohne die Leistungsstufe zu spezifizieren.

Zur Veranschaulichung der Bildung von Varianten mittels der kombinatorischen Logiksynthese wird in Listing 3.2 das Beispiel um drei weitere Komponenten sowie der Klasse `FactoryConfiguration` ergänzt. Letztere Datenstruktur umfasst eine variable Anzahl an Schneidemaschinen. Die Komponente `cuttingMachineMidLevel` in der Zeile 2 bis 5 entspricht der Implementierung aus Listing 3.1. Ferner wird in der Zeile 7 bis 10 Zeile die Komponente `cuttingMachineHighLevel` ergänzt, welche die Produktion einer Maschine einer hohen Leistungsstufe repräsentiert. Die Komponenten unterscheiden sich hinsichtlich ihrer Implementierungsdetails sowie des semantischen Typs. Jedoch weisen beide Komponenten denselben nativen Typen `CuttingMachine` auf. Überdies wird in der Zeile 12 bis 17 die Komponente `factoryConfiguration` ergänzt, welche die Top-Level-Komponente in diesem Beispiel darstellt. Eine Top-Level-Komponente verantwortet die Produktion der geforderten Lösungen, in diesem Fall also eine Fabrikkonfiguration. Die Typspezifikation dieser Top-Level-Komponente wird daher im Rahmen der Angabe eines Syntheseziels angegeben. Anhand der angegebenen Typspezifikation sucht das CLS-Framework nach Komponenten, die dieser Typspezifikation entsprechen. Die Top-Level-Komponente erwartet zwei Argumente, die zum einen als Parameter in der Signatur der `apply`-Methode (vgl. Zeile 13) als auch in der Argumentliste des semantischen Typen (vgl. Zeile 16) spezifiziert sind. Die Argumente im semantischen Typen sind mittels des Typkonstruktors `=>:` separiert. Somit erwartet die Komponente `FactoryConfiguration` als erstes Argument eine Maschine, die schneiden kann, sowie als

```
trait MachineRepository {
  @combinator object cuttingMachineMidLevel {
    def apply: CuttingMachine = new CuttingMachineMidLevel
    val semanticType: Type = Machine :&: IsCutting :&: MidLevel
  }

  @combinator object cuttingMachineHighLevel {
    def apply: CuttingMachine = new CuttingMachineHighLevel
    val semanticType: Type = Machine :&: IsCutting :&: HighLevel
  }

  @combinator object factoryConfiguration {
    def apply(cuttingMachine1: CuttingMachine, cuttingMachine2:
     ↪ CuttingMachine): FactoryConfiguration =
      new FactoryConfiguration(cuttingMachine1, cuttingMachine2)

    val semanticType: Type = Machine :&: IsCutting =>: Machine :&:
     ↪ IsCutting :&: HighLevel =>: Factory :&: HasCuttingMachines
  }
}
```

Listing 3.2: Implementierung einer Komponentensammlung zur Produktion einer Fabrikkonfiguration.

zweites Argument eine Maschine, die schneiden kann und zudem einer höheren Leistungsstufe entspricht. Hierbei wird die Bildung von Varianten durch eine Unterspezifizierung des ersten Arguments im semantischen Typen der Top-Level-Komponente erreicht. Dadurch kann der Inhabitationsalgorithmus des Syntheseframeworks zwischen den Komponenten `cuttingMachineMidLevel` und `cuttingMachineHighLevel` wählen, da die Typspezifikationen beider Komponenten dem geforderten Typ des Arguments entsprechen. Durch die Ergänzung von Komponenten oder einer weiteren Unterspezifizierung der Typspezifikationen kann die Bildung weiterer Varianten erreicht werden.

3.4.2 Inhabitation

Das Listing 3.3 zeigt den Quellcode, der die Durchführung der komponentenbasierten Synthese mittels des Syntheseframeworks CLS für das aktuelle Beispiel anstößt. Zunächst wird in der Zeile 1 eine Instanz der Komponentensammlung (vgl. Listing 3.2) erzeugt und dem Konstruktor der CLS-Datenstruktur `ReflectedRepository` übergeben. Letzteres nutzt die Reflexion im Sinne der Programmierung, um eine Komponentensammlung für die komponentenbasierte Synthese zu konstruieren. In der Zeile 2 folgt der Aufruf der `inhabit`-Funktion der zuvor konstruierten Komponentensammlung. Die Funktion erhält als Argument das Syntheseziel. In den runden Klammern ist der semantische Zieltyp angegeben, die der Typspezifikation der Top-Level-Komponente entspricht. In den eckigen Klammern ist der native Zieltyp angegeben, der dem Rückgabetypen der Implementierungsdetails der Top-Level-Komponente entspricht.

```
1 val Gamma = ReflectedRepository(new MachineRepository {})
2 val inhabitationResult = Gamma.inhabit[FactoryConfiguration](Factory :&:
  ↪ HasCuttingMachines)
```

Listing 3.3: Quellcode zur Durchführung einer Inhabitationsanfrage mittels des Syntheseframeworks CLS. In der Zeile 1 wird eine Instanz der Komponentensammlung erzeugt und in der Zeile 2 folgt die Inhabitationsanfrage gegeben dem Syntheseziel, wobei der semantische Zieltyp in runden Klammern und der native Zieltyp in eckigen Klammern angegeben ist.

Die Rückgabe der `inhabit`-Funktion ist eine Instanz der Klasse `InhabitationResult`. Letztere verfügt über eine Funktion, welche die Interpretation ausgewählter synthetisierter Lösungsvarianten erlaubt, sofern die Lösungsmenge nicht leer ist. Somit werden die Lösungen nicht unmittelbar nach der Fertigstellung der Synthese interpretiert, sondern werden nacheinander und nach Bedarf angefordert werden. Hierzu wird eine Baumgrammatik, die aus der Synthese gegeben dem Zieltypen resultiert, verwendet. Die Baumgrammatik stellt eine Art Bauanleitung dar, aus der einzelne Lösungen abgeleitet werden können. Diese besteht aus einer Menge von Zuordnungen von Typen zu Komponenten, die diesen Typen entsprechen. Auch die Argumentlisten der Typspezifikationen der Komponenten sind in der Baumgrammatik berücksichtigt. Das Listing 3.4 zeigt die Baumgrammatik für das Beispiel. So lässt sich etwa der Zeile 1 entnehmen, dass der Typ *FactoryConfiguration* ∩ *Factory* ∩ *HasCuttingMachines* der Typspezifikation

der Komponente `factoryConfiguration` entspricht (vgl. Zeile 2). Hierbei setzt sich der Typ aus der Intersektion des nativen Typen `FactoryConfiguration` und dem semantischen Typen *Factory* ∩ *HasCuttingMachines* zusammen. Ferner können in der Zeile 3 und 4 die notwendigen Argumenttypen dieser Komponente abgelesen werden. In den darauffolgenden Zeilen werden den Argumenttypen ebenfalls entsprechende Komponenten zugewiesen. So wird das erste Argument der Top-Level-Komponente (vgl. Zeile 5) durch zwei Komponenten (vgl. Zeile 6 und 7), während das zweite Argument durch eine einzelne Komponente produziert wird.

```
1 FactoryConfiguration & Factory & HasCuttingMachines ->
2   Set((factoryConfiguration,
3     List(CuttingMachine & Machine & IsCutting,
4          CuttingMachine & Machine & IsCutting & HighLevel)))
5 CuttingMachine & Machine & IsCutting ->
6   Set((cuttingMachineMidLevel,List()),
7       (cuttingMachineHighLevel,List()))
8 CuttingMachine & Machine & IsCutting & HighLevel ->
9   Set((cuttingMachineHighLevel,List()))
```

Listing 3.4: Von CLS erzeugte Baumgrammatik für die beispielhafte Komponentensammlung aus Listing 3.2

Die Baumgrammatik ist im Kontext dieser Dissertation bei der Filterung von Lösungsvarianten mittels von Techniken des SMT-Constraint-Solvings von Relevanz (vgl. Kapitel 5). In diesem Zusammenhang wird die Baumgrammatik in prädikatenlogische Formeln übersetzt und es werden weitere domänenspezifische numerische Randbedingungen ergänzt. Mittels eines SMT-Constraint-Solvers werden anschließend Wörter oder Terme der Baumgrammatik berechnet, welche die Randbedingungen berücksichtigen und mittels des Syntheseframeworks interpretiert werden.

3.4.3 Eignung von CLS zur Simulationsmodellgenerierung

In diesem Unterkapitel wird die Eignung der kombinatorischen Logiksynthese in Bezug auf die Generierung von Simulationsmodellen und den dazugehörigen Kontrollstrategien diskutiert. Dabei wird insbesondere auf die in Kapitel 1.1 und 2.3.3 identifizierten Forschungslücken und Herausforderungen Bezug genommen, die in dieser Dissertation mittels des Einsatzes der kombinatorischen Logiksynthese adressiert werden. Zunächst sei anzumerken, dass im Kontext dieser Forschungsarbeit die Simulationsmodelle aus einer softwaretechnischen Sichtweise betrachtet werden. Diese betrachtet die Simulationsmodelle als Programme, die ein ablauffähiges Modell beinhalten, das durch eine Simulationsumgebung ausgeführt wird. Dies begründet den Einsatz eines Ansatzes zur komponentenbasierten Synthese von Software.

Agentenbasierte Simulationsmodelle stellen eine häufig verwendete Simulationsmethodik in Simulationsstudien der Produktion und Logistik dar (vgl. Kapitel 2.1.4). Die Modellierung

der Agenten und der Simulationsmodelle erfolgt in der Regel durch den Einsatz von Simulationsbausteinen (vgl. Kapitel 2.1.5). Dadurch ist häufig eine inhärente Komponentisierung der Simulationsmodelle vorhanden, bei der die Agenten Komponenten darstellen, die eine Logik kapseln. Die Logik wird mehrheitlich durch den Einsatz der ereignisorientierten Simulation repräsentiert, die ebenfalls durch die Komposition von Prozessbausteinen bausteinorientiert modelliert sind (vgl. Kapitel 2.1.3). Gemäß diesen Eigenschaften ist eine Überführung der Agenten und deren Logiken in Komponenten für die kombinatorische Logiksynthese mit einem überschaubaren Aufwand möglich. Dieses Vorgehen basiert auf einer Forschungsarbeit von Düdder et al. [33], bei der ein objektorientiertes Softwareframework in eine Komponentensammlung zur komponentenbasierten Synthese *migriert* wird.

Eine wesentliche Herausforderung bei der Überführung eines Simulationsmodells in eine Komponentensammlung stellt der Erhalt des domänenspezifischen Wissens dar. Dabei stellen die Funktion sowie die Schnittstellen einer Komponente die relevanten und notwendigen Informationen dar. In der kombinatorischen Logiksynthese mit Intersektionstypen werden diese über die semantischen Typspezifikationen der Komponenten definiert. Die Argumentlisten innerhalb der Typspezifikation einer Komponente erlauben eine Beschreibung der notwendigen und zulässigen Komponente, die eine Komponente für deren Produktion benötigt.

Wie bereits erläutert, stellt die Variation von Systemstrukturen einen wesentlichen Bestandteil in der Bildung von Varianten eines Simulationsmodells im Kontext von Problemstellungen aus der Produktion und Logistik dar. Tabeling [117] bezeichnet die bewusste, strukturelle Veränderung als *Strukturvarianz*, die wie folgt ausgeprägt sein kann:

- Entstehen und Verschwinden von Komponenten,
- Änderung der Verbindungsstruktur,
- Änderungen von Komponententypen – interner Umbau.

Die Ausprägungen sind durch den Einsatz der kombinatorischen Logiksynthese realisierbar. Das Entstehen und Verschwinden von Komponenten bezieht sich im Kontext der betrachteten Simulationsmodelle auf das Vorkommen oder Nichtvorkommen von Bestandteilen des Simulationsmodells, die durch entsprechende Komponenten repräsentiert werden. In der kombinatorischen Logiksynthese wird dies durch das Hinzufügen und Entfernen einer Komponente in der Komponentensammlung oder über die Anpassung der Typspezifikationen der Komponenten realisiert.

Die Veränderung der Verbindungsstruktur bezieht sich in dieser Dissertation auf die Synthese der Logik eines Agenten. Die Logik eines Agenten wird durch einen Prozessfluss repräsentiert, der aus Prozessbausteinen, die mittels Konnektoren miteinander verbunden sind, besteht. In diesem Zusammenhang bezieht sich die Verbindungsstruktur auf das Hinzufügen und Entfernen der Konnektoren. Die Erstellung der Konnektoren kann beispielsweise mittels

Komponenten erfolgen, die basierend auf einer synthetisierten Komposition der Bestandteile des Simulationsmodells die Konnektoren automatisiert erzeugen.

Die Änderung eines Komponententyps bezieht sich auf ein verändertes Verhalten der Komponente, das häufig das Resultat einer internen Anpassung ist. So könnte etwa ein Prozessbaustein, der einen Puffer repräsentiert, hinsichtlich der Reihenfolgeplanung derart angepasst werden, dass das Vorgehen Last-in-First-out umgesetzt wird, bei dem die Güter, die zuletzt eingelagert wurden, als Erstes ausgelagert werden. In der kombinatorischen Logiksynthese kann diese Ausprägung der Strukturvarianz durch das Hinzufügen von Komponenten mit demselben semantischen Typen, aber unterschiedlichen Implementierungen, realisiert werden. In dem genannten Beispiel, könnten zwei Komponenten verwendet werden, die jeweils einen Prozessbaustein zur Repräsentation eines Puffers produzieren, jedoch mit unterschiedlichen Konfigurationen hinsichtlich der Reihenfolgeplanung. Weiterhin kann der interne Aufbau einer Komponente durch die Komposition dieser Komponente beeinflusst werden. So könnte die Komponente, die den Puffer-Prozessbaustein produziert, die Konfiguration der Reihenfolgeplanung als Argument erwarten. Die konkreten Ausprägungen der Reihenfolgeplanung (z. B. First-in-First-out und Last-in-First-out) wären dann weitere Komponenten, die ebenfalls Teil der Komponentensammlung sind. Ferner wird neben den aufgezählten Ausprägungen auch die Beschreibung der Dynamik einer Aufbaustruktur als einen Teil der Strukturvarianz genannt. Hiermit ist gemeint, wie der dynamische Aufbau einer Aufbaustruktur beschrieben werden kann. In dieser Dissertation erfolgt diese Beschreibung durch die Gestaltung der Komponentensammlung, einschließlich der Typspezifikationen der Komponenten sowie der Angabe eines Syntheseziels.

Die Kontrollstrategien in Simulationsmodellen werden häufig durch Funktionen in höheren Programmiersprachen implementiert, die je nach vorliegender Strukturvariante eines Simulationsmodells angepasst werden müssen. Es ist etwa für eine Kontrollstrategie, die eine Reihenfolgeplanung implementiert, von Bedeutung, ob die Aufträge auf zwei oder drei Maschinen verteilt werden. Je nach Strukturvariante muss die Kontrollstrategie dann in angepasster Form generiert werden. Hierfür kann die komponentenbasierte Synthese ebenfalls verwendet werden. Insbesondere stellt die Synthese von Quellcode eine klassische Anwendung der Softwaresynthese dar. Das Framework CLS ist programmiersprachen-agnostisch und wird in unterschiedlichen Anwendungsfällen zur Synthese von Software verwendet. So wird in [77] eine Produktlinie von Maschinenbelegungsplänen generiert, in dem unterschiedliche Maschinenbelegungsalgorithmen synthetisiert, bewertet und ausgeführt werden. In [48] wird das CLS-Framework verwendet, um eine Softwareproduktlinie des Kartenspiels Solitär zu synthetisieren. Daher wird das verwendete Syntheseframework CLS zur Synthese der Kontrollstrategien eingesetzt.

Die Datengewinnung und Extraktion kann ebenfalls durch den Einsatz der komponentenbasierten Logiksynthese realisiert werden. In [105] wird das CLS-Framework verwendet, um Systeme der Tourenplanung zu synthetisieren. Hierbei werden verschiedene zusammenhängende Systeme (z. B. ERP-Systeme, Softwarekomponenten zur Kundenbewertung oder zum

Routing) bei der Generierung berücksichtigt, sodass das synthetisierte System unmittelbar nutzbar ist.

In der Arbeit von Schäfer et al. [104] werden Motion Planning Programme mit einem hohen Freiheitsgrad hinsichtlich der Variabilität synthetisiert. Um eine systematische Exploration des Lösungsraums zu realisieren, erweitert die Autorenschaft die komponentenbasierte Synthese um lernende und statistische Komponenten. Hierzu bilden die Synthese von Motion Planning Programmen und die Evaluation der synthetisierten Programme, mit einem numerischen Vektor als Ausgabe, eine Blackbox-Funktion, die von einem Mehrzieloptimierungstool verwendet, um eine automatisierte, lernende Suchraumexploration zu realisieren. Dieses Vorgehen wurde experimentell anhand der Synthese von Motion Planning Programmen demonstriert und validiert. Auch in Bezug auf die Synthese von Simulationsmodellen stellt die Fähigkeit von CLS zur Integration von lernenden und statistischen Komponenten eine wesentliche Eigenschaft dar. Damit kann der Ansatz auch auf die Synthese von Simulationsmodellen übertragen und somit eine systematische und lernende Untersuchung des Lösungsraums komplexer Simulationsmodelle realisiert werden. Es sei hervorzuheben, dass diese Erweiterung jedoch einen Forschungsschwerpunkt zukünftiger Arbeiten darstellt.

Abschließend kann zusammengefasst werden, dass die kombinatorische Logiksynthese als Unterstützung in der Formulierung, Implementierung und Überprüfung eines Simulationsmodells geeignet ist. Die Simulationsmodellformulierung kann durch eine automatisierte Datengewinnung und -konsolidierung mittels der komponentenbasierten Synthese beschleunigt und verbessert werden. Die Implementierung wird durch eine automatisierte Komposition von Strukturvarianten eines Simulationsmodells sowie der Synthese der entsprechenden Kontrollstrategien unterstützt. Die Überprüfung eines Simulationsmodells wird durch den Umstand vereinfacht, dass eine Komposition von validierten und verifizierten Komponenten ebenfalls validiert und verifiziert ist und somit eine erneute Überprüfung entfällt.

3.5 Zusammenfassung

In diesem Kapitel wurden die theoretischen Grundlagen zur kombinatorischen Logik mit Intersektionstypen eingeführt, die die Basis für das in dieser Dissertation verwendete komponentenbasierte Syntheseframework CLS bildet. Letzteres wurde im Rahmen dieses Kapitels vorgestellt sowie dessen Verwendung anhand eines Beispiels demonstriert. Abschließend wurde die Eignung der kombinatorischen Logiksynthese in Bezug auf die Generierung von strukturell verschiedenen Simulationsmodellen und Kontrollstrategien diskutiert.

Kapitel 4

Migration von Simulationsmodellen in Produktlinien

In den vorangegangenen Kapiteln wurde die Simulation im Kontext der Produktion und Logistik sowie die kombinatorische Logiksynthese vorgestellt. Diese beiden Themenkomplexe werden in diesem Kapitel miteinander verbunden, indem die Idee der Migration von ereignisorientierten Simulationsmodellen zur Synthese einer Familie von Simulationsmodellvarianten vorgestellt wird.

Bei diesem Ansatz werden die Simulationsmodelle nicht von Grund auf generiert, sondern basieren auf bereits existierenden Simulationsmodellen. Ein bereits existentes Simulationsmodell wird hierfür in eine Komponentensammlung für die komponentenbasierte Synthese überführt. Dieses Vorgehen kann graduell erfolgen, sodass die Anzahl der zu migrierenden Elemente sowie die Granularität der Aufteilung nach Bedarf angepasst werden können. In der Praxis bedeutet dies, dass nicht zwingend alle Bestandteile eines Simulationsmodells variiert werden müssen und dass die Komponenten zunächst in relativ große Komponenten aufgeteilt werden, die in weiteren Iterationen in kleinere Komponenten zergliedert werden. Bevor jedoch der Ablauf der Migration vorgestellt wird, folgt zunächst eine Diskussion über die Gründe für die Wahl dieses Ansatzes.

4.1 Gründe für die Migration

In der Literatur zum Thema Automatic Simulation Model Generation (ASMG) wird häufig die fehlende Universalität entsprechender Ansätze als Forschungslücke betrachtet, die auf höchst individuelle Lösungen für spezifische Problemstellungen zurückzuführen ist. Der vorgestellte Migrationsansatz ist hingegen auf keine feste Domäne eingeschränkt, sondern ermöglicht die Migration beliebiger ereignisorientierter Simulationsmodelle. Dies wird durch

die komponentenbasierte Sichtweise und durch eine Abstraktion der Bestandteile eines Simulationsmodells ermöglicht. Letztere reduziert die Bestandteile auf parametrisierte Bausteine, die in Verbindung zueinander stehen.

Die Verwendung eines bereits existierenden Simulationsmodells bietet mehrere Vorteile. Zum einen müssen die Komponenten nicht von Grund auf erstellt werden, sondern können aus bestehenden Simulationsmodellen extrahiert werden, sodass der Aufwand für die Erstellung der Komponentensammlung reduziert wird. Zum anderen kann der Migrationsprozess in bestehende Vorgehensmodelle zur Durchführung von Simulationsstudien integriert werden. Hierdurch können bewährte Abläufe beibehalten und die Simulationsfachkräfte können weiterhin mit den ihnen bekannten Simulationswerkzeugen arbeiten. Ferner erfolgt die Konzeption und Umsetzung des Migrationsansatzes unter dem Gesichtspunkt einer möglichst benutzerfreundlichen und intuitiven Verwendung, um die Einstiegshürde der Verwendung des Ansatzes möglichst gering zu halten. Daher wird im Kontext dieser Dissertation eine Webanwendung entwickelt, welche die komponentenbasierte Synthese der Simulationsmodelle im Rahmen des ersten und zweiten Anwendungsfalls (vgl. Kapitel 5 und 6) ermöglicht. Die komponentenbasierte Synthese der Simulationsmodelle im dritten Anwendungsfall (vgl. Kapitel 7) erfolgt in einer im Simulationsmodell eingebetteten grafischen Benutzeroberfläche.

Ebenso können Teile der Migration automatisiert werden, wie die Aufteilung und Erstellung von Komponenten. Auch kann die Anzahl der zu untersuchenden Variabilitätspunkte im Rahmen der Migration stetig erweitert werden. Dies stellt einen wesentlichen Vorteil des Migrationsansatzes dar, denn bereits in frühen Phasen der Migration können die ersten Simulationsmodelle synthetisiert werden. Damit unterscheidet sich das Vorgehen zu modellgetriebenen Ansätzen, bei denen zunächst die Entwicklung eines vollständigen formalen Modells aussteht, das dann in den entsprechenden Quellcode übersetzt wird.

Als Herausforderungen der Migration sind die initialen Kosten der Migration zu nennen, die sich bei einer gewissen Variabilität und Komplexität des betrachteten Anwendungsfalls amortisieren. Weiterhin setzt die Migration zur komponentenbasierten Synthese eine bausteinorientierte Modellierung des Simulationsmodells voraus, um diese mit geringem Aufwand in Komponenten zu überführen. Jedoch erfordert insbesondere die agentenbasierte Modellierung unabhängig davon eine objektorientierte Sicht auf die Problemstellung. Ferner ist anzunehmen, dass die Anzahl der Variabilitätspunkte in vielen Anwendungsfällen recht schnell skaliert.

4.2 Vorgehen bei der Migration

Nachfolgend wird das grundlegende Vorgehen zur Migration von Simulationsmodellen zur Synthese von Simulationsmodellvarianten vorgestellt. Dieses Vorgehen wird im Kern in den untersuchten Anwendungsfällen dieser Dissertation (vgl. Kapitel 5, 6 und 7) eingesetzt. Die Abbildung 4.1 skizziert den Ablauf der Migration.

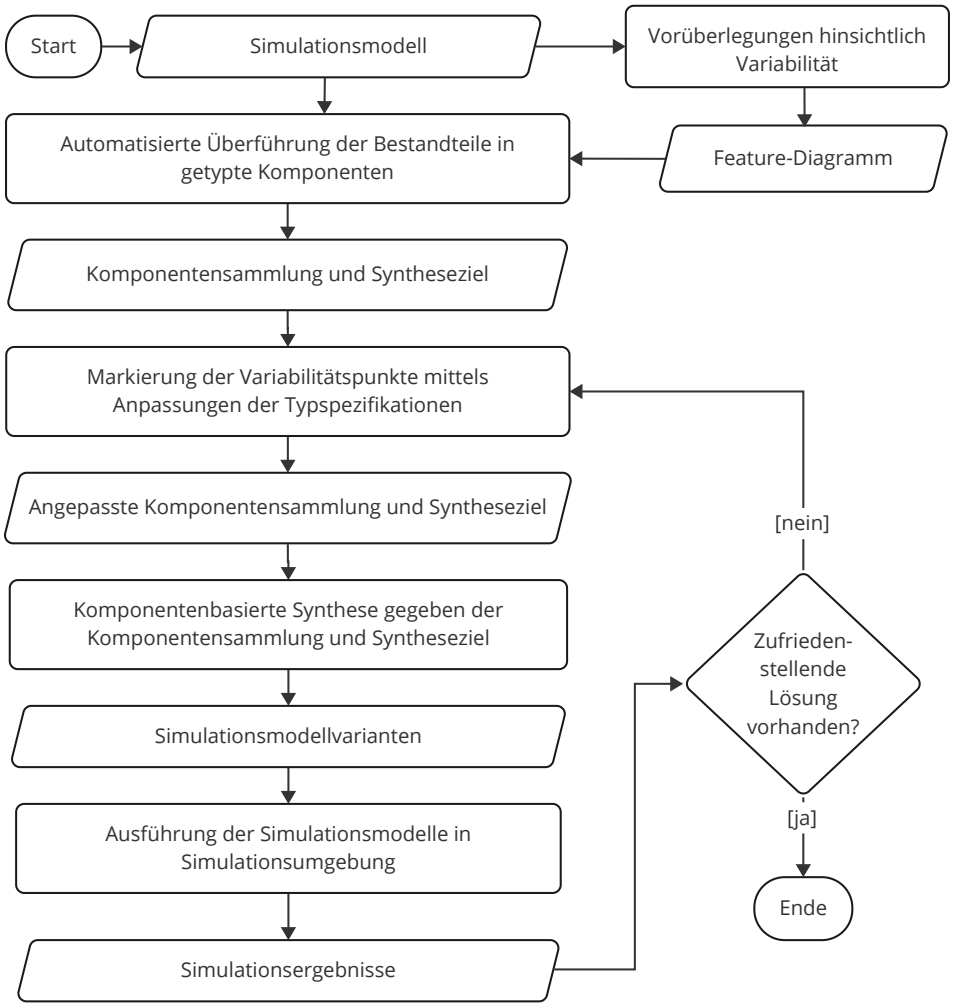


Abbildung 4.1: Vorgehen zur Migration eines bereits existierendes Simulationsmodell in eine Produktlinie mit den einzelnen Schritten beginnend von Vorüberlegungen hinsichtlich geforderter Variabilitätspunkte bis hin zur Simulation der synthetisierten Simulationsmodellvarianten.

In diesem Vorgehen bildet der Ausgangspunkt das zu migrierende Simulationsmodell, das den jeweiligen Anwendungsfall modelliert. Bevor die Migration erfolgen kann, müssen zunächst die zu untersuchenden Variabilitätspunkte des Systems systematisch identifiziert werden. Diese Punkte werden in der Regel in Bezug auf die Zielsetzungen einer Simulationsstudie festgelegt. Die systematische Variation dieser Variabilitätspunkte mittels der komponentenbasierten Synthese ermöglicht die Generierung der Strukturvarianten des Simulationsmodells. Die Variabilitätspunkte eines Systems können durch Feature-Modelle beschrieben werden, deren Visualisierung zum Beispiel durch Feature-Diagramme erfolgt. Anhand eines Feature-Modells können sämtliche Varianten eines Systems abgeleitet werden. Insbesondere können zwingend notwendige Elemente und Alternativen ausgewählter Elemente beschrieben werden. Daher werden Feature-Modelle oftmals zur Beschreibung von Softwareproduktlinien

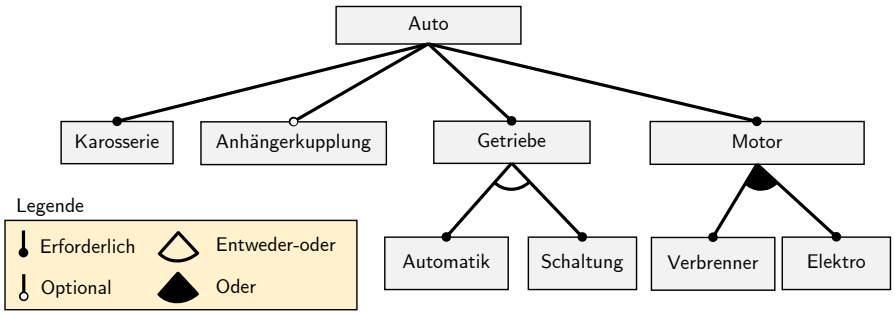


Abbildung 4.2: Feature-Diagramm, das die Variabilitätspunkte eines Kraftfahrzeugs umfasst.

verwendet [26]. Die Abbildung 4.2 zeigt das Feature-Diagramm eines Feature-Modells, das die Variabilitätspunkte eines Kraftfahrzeugs umfasst. Gemäß dem Feature-Modell benötigt ein Kraftfahrzeug zwingend eine Karosserie, während es entweder über ein Automatik- oder ein Schaltgetriebe verfügt. Der Motor setzt sich aus einem Verbrennungsmotor, Elektromotor oder einer Kombination zusammen. Anhand der Variabilitätspunkte können für dieses Beispiel $1 \times 2 \times 2 \times 3 = 12$ Varianten abgeleitet werden. Im Rahmen des Migrationsansatzes dient das Feature-Diagramm als Hilfestellung zur Aufteilung und Anpassung der Komponenten. An dieser Stelle sei anzumerken, dass das Feature-Modell im Laufe der Migration um weitere Variabilitätspunkte ergänzt werden kann.

In einem zweiten Schritt wird nach der Identifizierung und Auswahl der Variabilitätspunkte das Simulationsmodell in getypte Komponenten aufgeteilt. Hierbei werden je nach Anwendungsfall die Agenten, Prozessbausteine und/oder Funktionen in Komponenten ausgelagert. Bei der Überführung in eine Komponentensammlung wird berücksichtigt, dass domänenspezifischen Informationen, wie die Parametrisierung, oder Abhängigkeiten von anderen Elementen, nicht verloren gehen. Bezüglich der Typisierung der Komponenten muss eine eindeutige Zuordnung einer Komponente gewährleistet werden, um die ungewollte Bildung von Varianten zu vermeiden. Sofern beispielsweise zwei unterschiedliche Komponenten denselben Typen haben, werden diese im Rahmen der Ausführung der Synthese als Alternativen behandelt. Ebenso muss die Typisierung für die anwendende Person verständlich und intuitiv sein, da diese Person die Typspezifikationen zur Bildung von Varianten anpassen können soll. Neben den Simulationsbausteinen und Komponenten wird im ersten Schritt auch ein Syntheseziel extrahiert. Letzteres bezeichnet die Typspezifikation der Top-Level-Komponente, also jene, die die Produktion des tatsächlichen Agenten beziehungsweise des Simulationsmodells verantwortet. Das Syntheseziel kann als eine übergeordnete Bauanleitung betrachtet werden. Im Rahmen der Migration entspricht das abgeleitete Syntheseziel dem ursprünglichen Simulationsmodell, sodass die Ausführung der Synthese ohne eine Anpassung der Komponenten und des Syntheseziels in der Produktion einer Kopie des ursprünglichen Simulationsmodells resultiert. Die Anforderungen an die Simulationsbausteine sowie der Aufbau und die Typisierung der Komponenten werden in Kapitel 4.2 thematisiert.

Im dritten Schritt markiert die anwendende Person die Variabilitätspunkte durch Anpassun-

Nr.	Anwendungsfall	Produktionsergebnis	Integration	Automatisierung der Komponentisierung
1	Shop-Floor	Maschinenkonfiguration	extern	teil-automatisiert
2	Bodenblocklager	Kontrollflussgraph und Kontrollstrategien	extern	automatisiert
3	Shop-Floor mit Lager	Komposition von Maschinenzellen	intern	manuell

Tabelle 4.1: Übersicht über die Anwendungsfälle, die in der vorliegenden Dissertation in Produktlinien migriert werden. Hierbei wird zwischen dem fachlichen Schwerpunkt der Anwendungsfälle, dem Ergebnis der Produktion, der Umsetzung des Ansatzes sowie dem Grad der Automatisierung der Komponentenerstellung unterschieden.

gen der Komponentensammlung. Eine Anpassung erfolgt beispielsweise durch das Duplizieren, Hinzufügen oder Entfernen von Komponenten. Ferner kann die Parametrisierung eines Prozessbausteins, der von einer Komponente produziert wird, angepasst werden. Weiterhin erfolgen Anpassungen in Form von Modifikation der Typspezifikationen. Dadurch können Argumente einer Komponente hinzugefügt, entfernt oder ersetzt werden. Auch die Unterspezifizierung von Komponenten stellt ein Mittel zur Realisierung der Bildung von Strukturvarianten dar. Im Rahmen dieses Schrittes kann auch das Syntheseziel angepasst werden.

Im vierten Schritt wird die komponentenbasierte Synthese durchgeführt, wobei die angepasste Komponentensammlung und das modifizierte Syntheseziel die Eingaben bilden. Sofern die Inhabitationsanfrage erfolgreich beantwortet wird und damit die Menge der synthetisierten Lösungen nicht leer ist, werden im fünften und letzten Schritt die synthetisierten Simulationsmodellvarianten in der entsprechenden Simulationsumgebung ausgeführt. Anschließend folgt eine Auswertung der Simulationsläufe. Sofern keine der synthetisierten Varianten zufriedenstellende Ergebnisse liefert, folgt eine weitere Iteration des Vorgehens, in der die Komponentensammlung und das Syntheseziel erneut angepasst werden, um weitere Simulationsmodellvarianten zu synthetisieren.

Wie eingangs erwähnt, unterscheidet sich das grundlegende Vorgehen der Migration eines Simulationsmodells in den unterschiedlichen Anwendungsfällen lediglich in Details. Beispielsweise differiert die Art der Generierung als auch der Grad der Automatisierung im Kontext der Aufteilung in Komponenten. Die Tabelle 4.1 fasst die Unterschiede der Anwendungsfälle bezüglich der zuvor genannten Merkmale zusammen.

So werden im ersten Anwendungsfall Simulationsmodelle synthetisiert, die unterschiedliche Maschinenkonfigurationen eines Shop-Floors repräsentieren. Dabei werden die Simulationsmodelle extern generiert und die Bildung der Komponentensammlung erfolgt teilautomatisiert. Ferner wird der Einsatz von SMT-Techniken in Kombination mit der komponentenbasierten Synthese von Simulationsmodellen untersucht, um die synthetisierten

Maschinenkonfigurationen nach den Gesamtkosten und der Durchlaufzeit zu filtern. Hierfür wird die Implementierung ausgewählter Komponenten um Metainformationen erweitert, welche die Kosten und die benötigte Dauer als numerische Werte ergänzen. Der zweite Anwendungsfall fokussiert die komponentenbasierte Synthese von Kontrollflussgraphen und Kontrollstrategien agentenbasierter Simulationsmodelle. Auch in diesem Anwendungsfall erfolgt die Generierung extern. Jedoch wird im zweiten Anwendungsfall ein Vorgehen zur voll automatisierten Aufteilung eines Kontrollflussgraphen in eine Komponentensammlung vorgestellt. Anhand von Simulationsmodellen eines fiktiven Produktionssystems und eines Bodenblocklagers wird dieses Vorgehen demonstriert. Im dritten Anwendungsfall stehen hingegen die Synthese von Kompositionen von Maschinenzellen und die interne Generierung der Simulationsmodelle im Fokus.

4.3 Integration der Migration in bestehende Vorgehensmodelle

Nach der Vorstellung des grundlegenden Vorgehens zur Migration von Simulationsmodellen wird nachfolgend die Integration in ein gängiges Vorgehensmodell zur Durchführung von Simulationsstudien beschrieben. Hierbei wird exemplarisch die Integration in das Vorgehensmodell des VDI vorgenommen. Die Integration baut auf einer Forschungsarbeit von Baier et al. [4] auf, bei der die automatisierte Simulationsmodellerstellung in das genannte Vorgehensmodell integriert wird. Die Autorenschaft integriert dabei einen Ansatz, der die automatisierte Erstellung der Simulationsmodelle auf Basis einer Datenbank mit vorgefertigten Simulationsmodellen vornimmt. Der Ansatz sieht vor, dass die vordefinierten Simulationsmodelle so lange nacheinander kombiniert, ausgeführt und ausgewertet werden, bis eine der erzeugten Simulationsmodelle zufriedenstellende Ergebnisse liefert. Hierzu ergänzt die Autorenschaft mehrere Schritte in dem Vorgehensmodell des VDI. Ferner sieht das angepasste Vorgehensmodell von Baier et al. vor, dass die Datenbank mit den vorgefertigten Simulationsmodellen von einer Simulationsfachkraft während der Durchführung der Simulationsstudie bei Bedarf um weitere Simulationsmodelle ergänzt wird, um weitere Varianten abzudecken.

In diesem Abschnitt wird ein angepasstes Vorgehensmodell vorgestellt, das auf dem Vorgehensmodell von Baier et al. in [4] aufbaut. Die Abbildung 4.3 zeigt das angepasste Vorgehensmodell, in dem die neu hinzugekommenen Schritte schraffiert hinterlegt sind. Dieses angepasste Vorgehensmodell weicht in den Schritten, die nach der Erstellung des Simulationsmodells verortet sind, vom ursprünglichen Vorgehensmodell des VDI ab. Diese Schritte werden nachfolgend erläutert. Nach der Erstellung des Simulationsmodells folgt die Migration zur Synthese einer Produktlinie von Simulationsmodellvarianten. Die Migration umfasst die Aufteilung des Simulationsmodells in Komponenten, die Markierung von Variabilitätspunkten durch Anpassungen der Komponenten sowie die komponentenbasierte Synthese der Simulationsmodelle. Nach der Migration folgt die Überprüfung der synthetisierten Simulationsmodellvarianten in Bezug auf die Akkuratheit der Nachstellung des abzubildenden Systems. Hierzu können die Ergebnisse des Schritts der Problemanalyse verwendet werden,

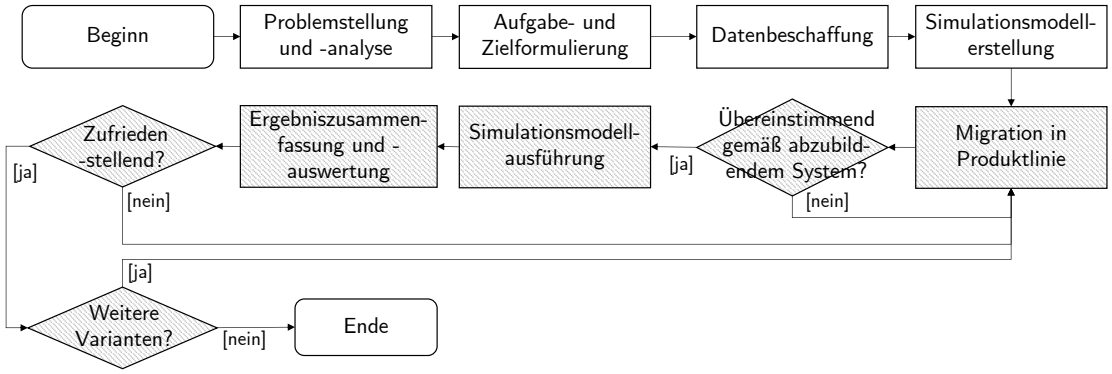


Abbildung 4.3: Einbettung des Migrationsansatzes in ein Vorgehensmodell zur Durchführung von Simulationsstudien basierend auf dem Vorgehensmodell des VDI [1] und Baier et al. [4]. Die schraffierten Rechtecke stellen die angepassten oder hinzugefügten Schritte im Vergleich zum ursprünglichen Vorgehensmodell des VDI dar.

die eine eindeutige Beschreibung des abzubildenden Systems liefern. Diese beschreibt unter anderem die Abfolge von Aktionen beim Auftritt ausgewählter Bedingungen durch den Einsatz von Entscheidungstabellen. Sofern eine Simulationsmodellvariante nicht dem abzubildenden System entspricht, wird diese Variante verworfen. In diesem Fall sollte die Migration erneut durchgeführt werden. Andernfalls folgt im nächsten Schritt die Ausführung und Auswertung der Simulationsmodelle. Da die Anzahl der Simulationsmodellvarianten unter Umständen hoch ist, wird hierbei eine automatisierte Auswertung der Simulationsläufe vorgeschlagen. Die Ausführung und Auswertung der Simulationsmodelle kann mittels Skripten oder Verwendung von Cloud-Simulationsumgebungen automatisiert werden. Im nächsten Schritt werden die Auswertungen zur Erstellung entsprechender Zusammenfassungen verwendet, die als Unterstützung zur Überprüfung der Ergebnisse genutzt werden. Die Überprüfung erfolgt häufig durch den Vergleich von Ist-Kennzahlen mit den Soll-Kennzahlen, die in der Planungsphase festgelegt wurden. Sofern die Ergebnisse nicht den erwarteten Werten entsprechen, kann auch hier die Migration erneut durchgeführt werden, um weitere Varianten zu erhalten. Sofern jedoch die Ergebnisse beziehungsweise die Lösungsvarianten die Anforderungen erfüllen, so kann die Simulationsstudie vorerst abgeschlossen werden.

Mit der exemplarischen Integration in ein bereits bestehendes, gängiges Vorgehensmodell zur Durchführung von Simulationsmodellen kann gezeigt werden, dass der Migrationsansatz in bestehende Prozesse integrierbar ist, ohne diese grundlegend neu zu gestalten. Daher ist es auch denkbar, die Integration in weitere Vorgehensmodelle vorzunehmen.

4.4 Simulationsbausteine und Komponenten

In diesem Unterkapitel werden die Simulationsbausteine und die Komponenten vorgestellt. Erstere stellen eine Datenstruktur dar, welche die Bestandteile eines Simulationsmodells reprä-

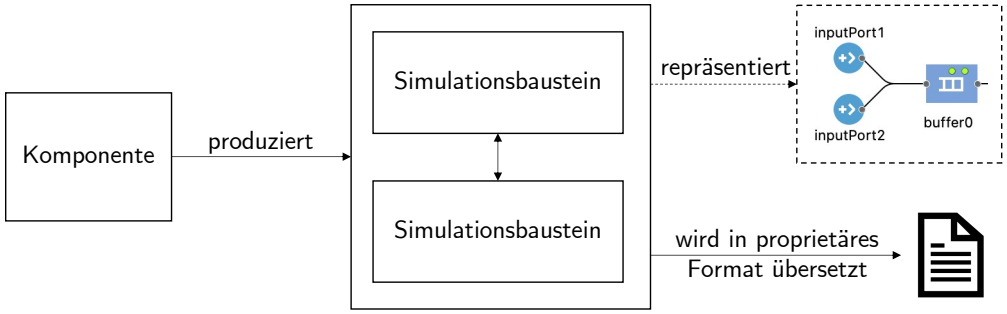


Abbildung 4.4: Schematische Darstellung der Beziehung zwischen einer Komponente und Simulationsbausteinen. Die Komponente produziert Komposition dieser Simulationsbausteine, die einen Teil eines Simulationsmodells repräsentiert und zur Übersetzung in ein proprietäres Format verwendet wird.

sentieren und im Kontext der Generierung der ausführbaren Simulationsmodelle verwendet werden. Die Komponenten hingegen verantworten die Produktion von einzelnen Simulationsbausteinen oder einer Komposition dieser. Die Abbildung 4.4 skizziert den Zusammenhang zwischen den Simulationsbausteinen und den Komponenten. So produziert eine Komponente eine Komposition von Simulationsbausteinen, möglicherweise aus einem einzelnen Simulationsbaustein bestehend. Die Komposition von Simulationsbausteinen repräsentiert einen Bestandteil des zu synthetisierenden Simulationsmodells und kann mittels eines Übersetzers in ein proprietäres Format einer Simulationsumgebung übersetzt werden.

Nachfolgend werden zunächst die Anforderungen an eine Implementierung der Simulationsbausteine und Komponenten erarbeitet. Danach wird eine Implementierung dieser vorgestellt.

4.4.1 Anforderungsanalyse

Im Folgenden bezeichnen die Anforderungen die Eigenschaften, welche die Simulationsbausteine und Komponenten aufweisen sollten. Wie bereits erwähnt, repräsentieren die Simulationsbausteine ausgewählte Bestandteile der Simulationsmodelle, während die Komponenten im Kontext der komponentenbasierten Softwaresynthese relevant sind. Zur Ermittlung der Eigenschaften der Simulationsbausteine werden Forschungsarbeiten betrachtet, welche die Eigenschaften der Simulationsbausteine im Kontext der automatischen Simulationsmodellgenerierung erarbeiten. Diese Eigenschaften werden anschließend auf die Relevanz in Bezug auf die Generierung mittels der komponentenbasierten Synthese untersucht.

Lee und Zobel zählen in [69] die Bestandteile eines Simulationsbausteins im Kontext einer komponentenorientierten Simulationsmodellgenerierung auf, die wie folgt lauten:

- Identifikationsnummer zur eindeutigen Zuordnung eines Simulationsbausteins,
- Beschreibung (z. B. textuelle Informationen über den Simulationsbaustein),

- Spezifikation (z. B. Simulationsmethodik),
- mathematisches Modell,
- physikalische Geometrie zur Visualisierung und Animation der Komponente und
- Quellcode des Simulationsbausteins.

Der Großteil der aufgelisteten Eigenschaften (ausgenommen das mathematische Modell sowie die physikalische Geometrie) werden im Rahmen dieser Dissertation ebenfalls als Eigenschaften eines Simulationsbausteins berücksichtigt. So ermöglicht die Identifikationsnummer eine eindeutige Zuordnung von Simulationsbausteinen. Im Kontext der komponentenbasierten Synthese wird diese zur gezielten Definition von Komponenten verwendet, die festgelegte Simulationsbausteine produzieren. Hierfür kann die Identifikationsnummer beispielsweise in der Typspezifikation der entsprechenden Komponente berücksichtigt werden. Ebenso relevant sind die Beschreibung und die Angabe der Simulationsmethodik, welche ebenso einen Teil der Typspezifikation bilden können, um das Verhalten einer Komponente zu beschreiben. Das mathematische Modell sowie die physikalische Geometrie werden nicht berücksichtigt. Dies hat den Hintergrund, dass diese in den geläufigen bausteinorientierten Simulationsumgebungen bereits implementiert sind. Daher werden diese bei der Definition einer Komponente nicht berücksichtigt. Jedoch ist der Quellcode eines Simulationsbausteins von Relevanz. Dieser Bestandteil wird in dieser Dissertation derart interpretiert, als dass sich der Quellcode aus Fragmenten zusammensetzt, die etwa die Logik beim Betreten und Verlassen eines Simulationsbausteins definieren.

Die Auflistung von Lee und Zobel wird in dieser Dissertation um zwei weitere Bestandteile ergänzt:

Schnittstelle Die Schnittstelle umfasst in diesem Zusammenhang die zulässigen Simulationsbausteine, die mit einem Baustein verbunden werden können. Simulationsmodelle setzen sich nämlich häufig aus Kompositionen von Simulationsbausteinen zusammen, die durch Verbindungen miteinander in Beziehung stehen. Hierbei kann über die Typspezifikation einer Komponente beeinflusst werden, aus welchen Simulationsbausteinen eine Komposition bestehen kann.

Parametrisierung Die Parametrisierung stellt in bausteinorientierten Simulationsmodellen ein wesentliches Werkzeug dar, um die vordefinierten Bausteine für unterschiedliche Problemstellungen einsetzen zu können. Im Kontext der komponentenbasierten Synthese kann hierdurch die Bildung von Varianten forciert werden, in dem unterschiedliche Komponenten denselben Simulationsbaustein mit einer differierenden Parametrisierung produzieren.

Während Lee und Zobel die Bestandteile eines Simulationsbausteins nennen, stellen Verbraeck und Valentin [120] übergeordnete designbezogene Eigenschaften vor. So sollte ein Simulationsbaustein

- in sich abgeschlossen,
- interoperabel,
- wiederverwendbar,
- austauschbar,
- über fest definierte Schnittstellen verfügen und
- die interne Struktur verbergen.

Die ermittelten Eigenschaften und Bestandteile der Simulationsbausteine werden nun verwendet, um die Bestandteile eines Simulationsmodells zu erarbeiten, die durch die Simulationsbausteine repräsentiert werden können. Zunächst sei an dieser Stelle zu wiederholen, dass im Rahmen dieser Dissertation agentenbasierte und ereignisorientierte Simulationsmodelle betrachtet werden. In den agentenbasierten Simulationsmodellen können die Agenten durch Simulationsbausteine repräsentiert werden. So charakterisieren sich die Agenten über einen eindeutigen Namen, einer Beschreibung, einer festgelegten Simulationsmethodik und Quellcode-Fragmenten zur Definition von Logiken. Ebenso können Schnittstellen der Agenten und eine Parametrisierung festgelegt werden. Überdies erfüllen die Agenten auch die designbezogenen Eigenschaften eines Simulationsbausteins, insofern, als die Modellierung eines Agenten in sich abgeschlossen und für andere Agenten nicht sichtbar ist. Weiterhin ist die Verwendung von Agenten in mehreren Simulationsmodellen möglich. Ferner sind sie austauschbar und können Eingaben erwarten und damit Schnittstellen erfüllen.

Die Logik der Agenten ist in den betrachteten Simulationsmodellen durch ein ereignisorientiertes Simulationsmodell definiert. Letztere umfassen Prozessbausteine, die ebenfalls durch Simulationsbausteine repräsentiert werden. Dementsprechend verfügen die Prozessbausteine über die notwendigen Bestandteile eines Simulationsbausteins. So besitzen die Prozessbausteine einen eindeutigen Namen, eine Beschreibung und Parametrisierung. Ferner ist den Prozessbausteinen die Simulationsmethodik bekannt, da diese häufig für eine vordefinierte Simulationsmethodik implementiert sind. Ebenfalls verfügen die Prozessbausteine über Quellcodefragmente zur Abbildung von Logiken, und über Schnittstellen, die von ausgewählten Prozessbausteinen bedient werden können. Auch die Prozessbausteine erfüllen die designbezogenen Eigenschaften nach Verbraeck und Valentin.

Nachfolgend wird auf die Verwendung des Begriffs des Simulationsbausteins verzichtet, um eine Mehrdeutigkeit zu vermeiden. Daher werden im weiteren Verlauf die Begriffe Agenten und Prozessbausteine verwendet, um auf die entsprechenden Simulationsbausteine inklusive ihrer Bestandteile zu referenzieren.

Nachstehend wird ein beispielhaftes Simulationsmodell eines Agenten mit dem Namen *Main* (Standardbezeichnung des Hauptagenten in der Simulationsumgebung AnyLogic 8) betrachtet, um exemplarisch die Agenten und Prozessbausteine zu ermitteln. Das Simulationsmodell

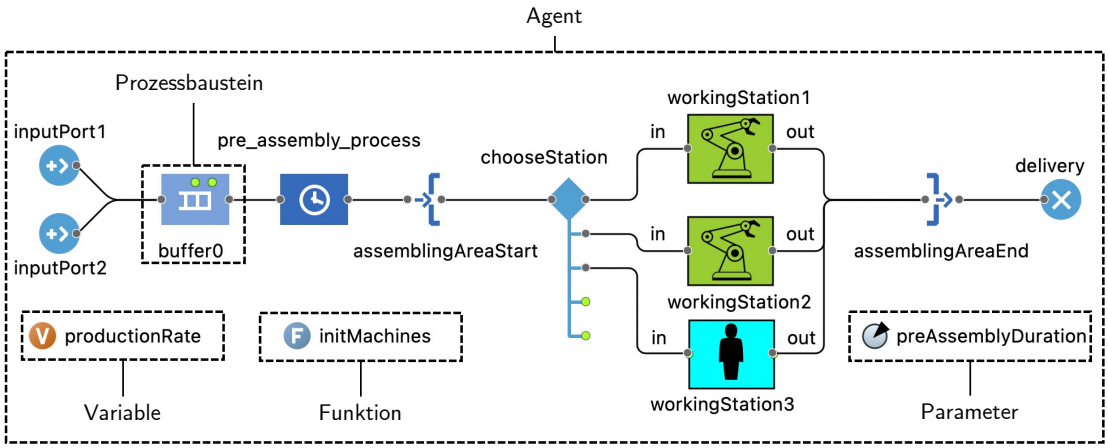


Abbildung 4.5: Modellierung eines Agenten in der Simulationsumgebung AnyLogic 8, in der die einzelnen Bestandteile der Modellierung in gestrichelten Kästen hervorgehoben sind, die im Rahmen des Migrationsansatzes in Komponenten überführt und durch diese produziert werden.

wurde in der Simulationsumgebung AnyLogic 8 erstellt und ist in der Abbildung 4.5 dargestellt. Der Agent *Main* stellt einen Simulationsbaustein dar, der durch eine Komponente produziert werden kann. Der Parameter `preAssemblyDuration` stellt einen Teil der Parametrisierung des Agenten dar, während die Funktion `initWorkingStations` und die Variable `productionRate` dem Quellcode zugeordnet werden. Überdies verfügt der Agent über eine textuelle Beschreibung sowie weitere Quellcodefragmente, die jedoch in dem abgebildeten Ausschnitt nicht sichtbar sind. Die Simulationsmethodik, hier die ereignisorientierte Simulation, ist den Metainformationen zu entnehmen.

Der Agent verfügt über einen Kontrollflussgraphen, der die Logik des Agenten mittels einer Komposition von Prozessbausteinen repräsentiert. Die Prozessbausteine des Kontrollflussgraphen wie *buffer0* werden nachfolgend ebenfalls durch Komponenten produziert. Bei den Prozessbausteinen ist zwischen vordefinierten Prozessbausteinen der Simulationsumgebung und Prozessbausteinen, die eigens implementierte Agenten repräsentieren, zu unterscheiden. So repräsentiert der Prozessbaustein *buffer0* einen vordefinierten Puffer, während der Prozessbaustein *workingStation1* eine ergänzte Arbeitsstation repräsentiert. Auch die Prozessbausteine verfügen über eine Parametrisierung und Quellcodefragmente, die in der Abbildung nicht sichtbar sind, da diese in der Simulationsumgebung AnyLogic 8 in einem separaten Fenster in der Benutzeroberfläche definiert werden.

Zuvor wurden die Anforderungen an die Simulationsbausteine im Kontext der Simulationsmodellgenerierung erarbeitet, während nun die Anforderungen hinsichtlich der kombinatorischen Logiksynthese mittels des Syntheseframeworks Combinatory Logic Synthesizer folgen. Die grundlegende Anforderung an die Komponenten besteht in der Verantwortung der Produktion der Agenten und Prozessbausteine. Hierbei sollen unterschiedliche Komponenten verschiedenartige Bestandteile eines Simulationsbausteins produzieren. Wie in der Vorstel-

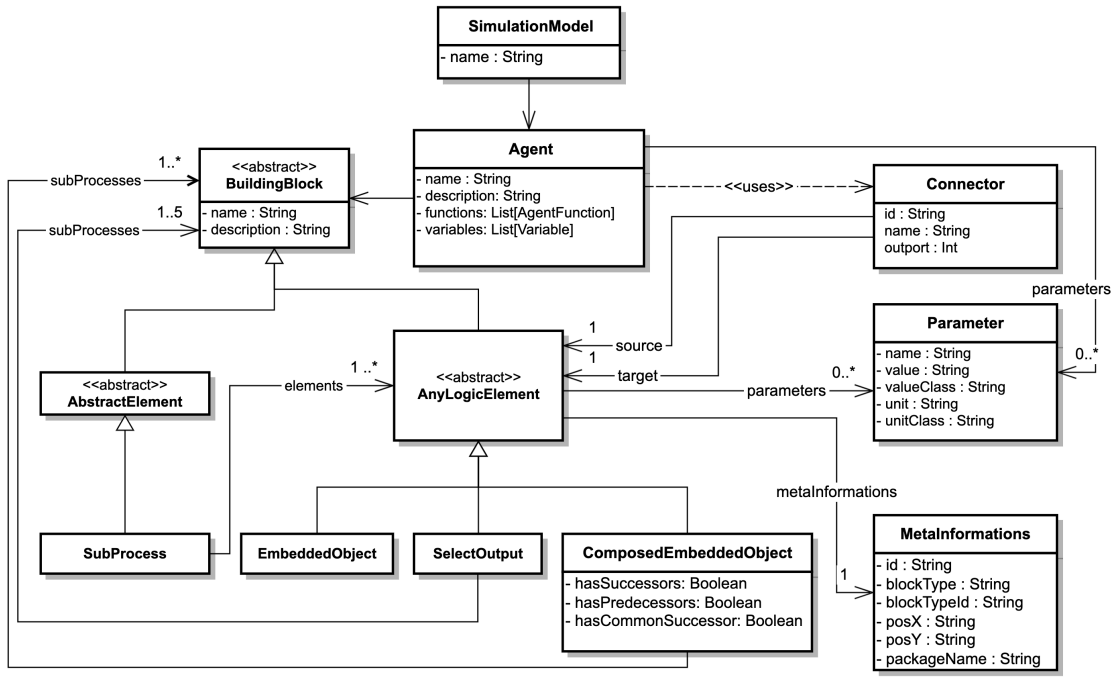


Abbildung 4.6: UML-Klassendiagramm der im Rahmen dieser Dissertationen konzeptionierten Datenstrukturen zur Repräsentation von agentenbasierten Simulationsmodellen in AnyLogic 8

lung des Combinatory Logic Synthesizer-Frameworks (vgl. Kapitel 3.4) erläutert, verfügen die Komponenten über eine Typspezifikation und Implementierungsdetails. Die Typspezifikation einer Komponente erlaubt die eindeutige Zuordnung, Beschreibung und Spezifikation der Schnittstellen eines zu synthetisierenden Simulationsbausteins. Die Implementierungsdetails ermöglichen die Produktion eines Agenten oder Prozessbausteins inklusive der Parametrisierung. Hierbei variiert das Produktionsergebnis und die Argumente von XML-Fragmenten in einem proprietären Format einer Simulationsumgebung (externe Generierung) bis hin zur Instantiierung von Datenobjekten unmittelbar im Simulationsmodell (interne Generierung).

4.4.2 Implementierung

In diesem Abschnitt werden die Datenstrukturen zur Realisierung der zuvor vorgestellten Agenten und Prozessbausteine vorgestellt. Weiterhin folgt die Implementierung der Komponenten, die diese produzieren. Abschließend erfolgt die Vorstellung eines Vorgehens zur Typisierung der Komponenten.

Das Klassendiagramm in der Abbildung 4.6 zeigt die Datenstrukturen zur Repräsentation agentenbasierter und ereignisorientierter Simulationsmodelle in AnyLogic 8. Die Konzeption und Implementierung dieser Datenstrukturen erfolgte im Kontext dieser Dissertation. Das Simulationsmodell wird durch die Klasse `SimulationModel` repräsentiert, die den Namen des Simulationsmodells enthält. Ferner verfügt die Klasse über eine Liste mit den Agenten

des Simulationsmodells. Die Agenten werden durch die Klasse `Agent` repräsentiert, die einen Namen, eine textuelle Beschreibung sowie eine Angabe der Simulationsmethodik aufweisen. Außerdem verfügt die Klasse über Listen mit den Parametern, Funktionen und Variablen des jeweiligen Agenten. Diese sind durch die Klassen `Parameter`, `Function` und `Variable` implementiert. Überdies verfügt die Klasse über eine Liste mit den Prozessbausteinen, die zur Repräsentation der Logik des Agenten verwendet werden.

Die Prozessbausteine werden durch Instanzen der abstrakten Klasse `BuildingBlock` repräsentiert. Die Klasse verfügt über zwei Variablen, die den Namen und die Beschreibung festlegen. Ferner stellt diese Klasse die Basisklasse der abstrakten Klassen `AbstractElement` und `AnyLogicElement` dar. Erstere repräsentiert einen abstrakten Prozessbaustein, der somit keinen tatsächlich vorhandenen Prozessbaustein repräsentiert. Beispielsweise repräsentiert die abgeleitete Klasse `SubProcess` eine Komposition von Prozessbausteinen zur Repräsentation eines Pfades in einem Kontrollflussgraphen. Hierfür verfügt die Klasse über eine Liste mit Prozessbausteinen, wobei angenommen wird, dass die Prozessbausteine in der Reihenfolge der Liste im Kontrollflussgraphen des Agenten vorkommen. Hingegen repräsentieren Instanzen der Klasse `AnyLogicElement` tatsächlich vorhandene Prozessbausteine. Daher umfasst die Klasse eine Variable des Datentyps `MetaInformations` zur Erfassung der Metainformationen eines Prozessbausteins. Zudem beinhaltet die Klasse eine Liste mit den Parametern des Prozessbausteins. Während Instanzen der abgeleiteten Klasse `EmbeddedObject` ausschließlich einzelne Prozessbausteine repräsentieren, umfassen Instanzen der abgeleiteten Klassen `ComposedEmbeddedObject` und `SelectOutput` zusätzliche eine Komposition von Prozessbausteinen. Daher verfügen letztere Datenstrukturen über jeweils eine Liste mit den Prozessbausteinen. Hierbei stellt jedes Element der Liste einen Pfad dar, der wiederum aus einem einzelnen Prozessbaustein oder einer Komposition bestehen kann. Ferner sei hervorzuheben, dass die Klasse `SelectOutput` einen Entscheidungsprozessbaustein repräsentiert. Dieser charakterisiert sich durch das Vorhandensein von Bedingungen, welche die Auswahl eines Pfades zur Simulationslaufzeit bestimmen. Die Bedingungen werden mittels der Parametrisierung des Prozessbausteins definiert. Die Klasse `ComposedEmbeddedObject` repräsentiert einen beliebigen Prozessbaustein und enthält Variablen, welche die Angabe der Existenz eines gemeinsamen Elements aller Pfade sowie die Positionierung der Pfade im Kontrollflussgraphen umfassen. Letzteres bezieht sich darauf, ob die Pfade vor oder nach dem Prozessbaustein platziert werden soll. Die Klassen `ComposedEmbeddedObject` und `SelectOutput` weisen Parallelen auf, sodass eine Zusammenlegung dieser realisierbar wäre. Jedoch wird nachfolgend darauf verzichtet, da eine Unterscheidung der Prozessbausteine in der Generierung der Simulationsmodelle notwendig ist. Zwar könnte die Unterscheidung durch die Ergänzung einer Variable realisiert werden, dies entspricht, aber nicht der empfohlenen Vorgehensweise in der objektorientierten Programmierung.

Nachfolgend werden die Datenstrukturen zur Repräsentation der Komponenten für die komponentenbasierte Softwaresynthese vorgestellt. Die Komponenten verantworten die Produktion der Agenten, Prozessbausteine und Simulationsmodelle. Jedoch erfolgt die Implementierung dieser Datenstrukturen derart, dass diese unabhängig von konkreten Simulations-

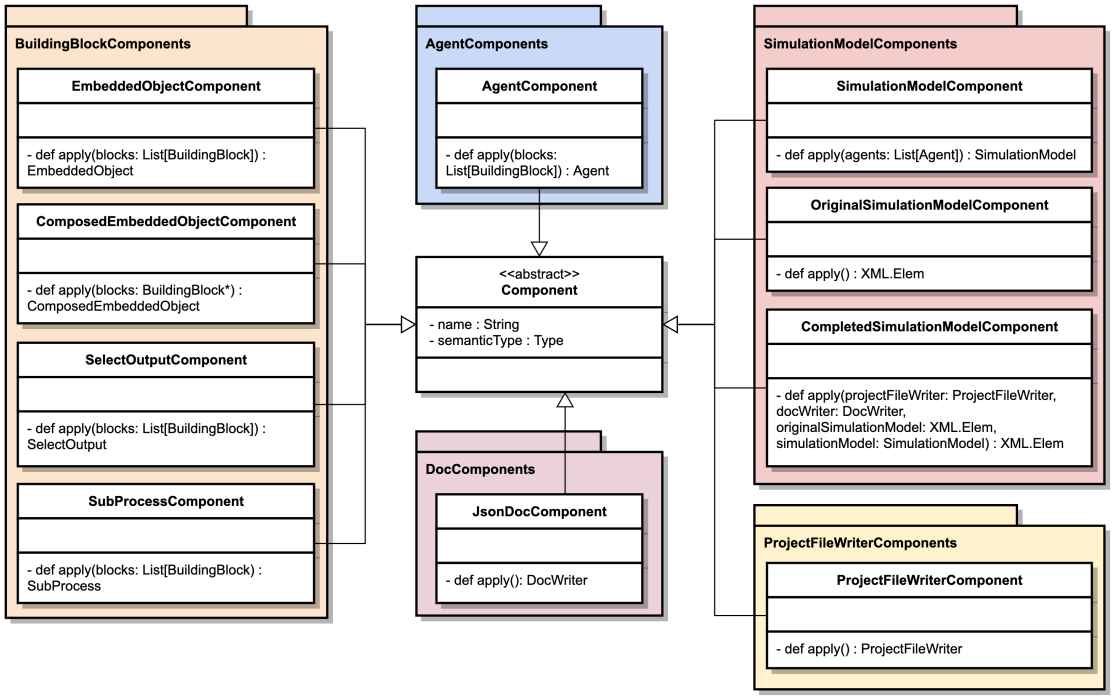


Abbildung 4.7: UML-Klassendiagramm der Datenstrukturen zur Repräsentation vordefinierter Komponenten, welche die Produktion unterschiedlicher Bestandteile eines Simulationsmodells verantworten. Diese Komponenten werden nach Bedarf instantiiert und dynamisch typisiert.

modellen, Agenten oder Prozessbausteinen ist. Damit kann eine einzelne Implementierung einer Komponente für die Produktion unterschiedlicher Elemente eingesetzt werden. Demzufolge erfolgt die Auswahl und die Instantiierung dieser Komponenten dynamisch. Die Komponenten unterscheiden sich hinsichtlich der Agenten- und Prozessbausteintypen, die sie produzieren, sowie in der Anzahl der erwarteten Argumente. Ferner sind Komponenten implementiert, die keine unmittelbaren Bestandteile des Simulationsmodells produzieren, sondern unter anderem die Übersetzung in proprietäre Formate oder die Generierung der Dokumentation verantworten. Diese Komponenten werden ebenso dynamisch instantiiert, jedoch ist die Implementierung der Komponenten hinsichtlich des Produktionsergebnisses und der Anzahl der Argumente fest. Das Klassendiagramm in der Abbildung 4.7 zeigt das Datenmodell dieser *vordefinierten* Implementierungen der Komponenten.

Die zentrale Instanz in diesem Datenmodell bildet die abstrakte Klasse `Component`, die über einen Namen und semantischen Typen verfügt und die Basisklasse aller anderen Klassen darstellt. Alle abgeleiteten Klassen haben eine `apply`-Methode, welche die Implementierungsdetails der jeweiligen Komponente beinhaltet. Die Methode erwartet Argumente, die im Rahmen der Produktion des Simulationsmodells, Agenten oder Prozessbaustein benötigt werden. Eine Vorstellung dieser Argumente erfolgt im Rahmen der Beschreibung der Typisierung der Komponenten. Durch das Vorhandensein eines Komponentennamen, semantischen

Typen und einer apply-Methode erfüllen die Implementierungen die Anforderungen an eine Komponente gemäß dem Syntheseframework CLS (vgl. Kapitel 3.4). Daher können die Instanzen dieser Datenstrukturen im Rahmen der Erstellung einer Komponentensammlung für die komponentenbasierte Synthese eingesetzt werden.

Die vordefinierten Implementierungen der Komponenten sind in Pakete gruppiert, die ähnliche Bestandteile eines Simulationsmodelles produzieren. Das Paket BuildingBlockComponents umfasst Komponenten zur Produktion von Prozessbausteinen eines Kontrollflussgraphen. Beispielsweise produziert eine Instanz der Komponente EmbeddedObjectComponent einen Prozessbaustein, der durch den Datentypen EmbeddedObject repräsentiert wird. Hingegen produziert eine Instanz der Komponente SubProcessComponent einen Pfad eines Kontrollflussgraphen und erwartet daher in der apply-Methode eine Reihe von Prozessbausteinen als Argumente. Das Paket AgentComponents beinhaltet die Komponente AgentComponent, welche die Produktion eines Agenten verantwortet. Im Paket SimulationModelComponents sind die Komponenten zur Produktion eines Simulationsmodells zusammengefasst. Hierbei verantwortet die Komponente OriginalSimulationModelComponent die Produktion des ursprünglichen Simulationsmodells, während die Komponente SimulationModelComponent eine Liste mit zuvor synthetisierten Agenten produziert. Die Komponente CompletedSimulationModelComponent stellt die Top-Level-Komponente dar, welche eine ausführbare Variante eines Simulationsmodells inklusive der entsprechenden Dokumentation produziert. Die Übersetzung der Datenstrukturen in ein proprietäres Format erfolgt mittels einer Instanz der Klasse ProjektFileWriter, die wiederum durch die Komponenten der Paketklasse ProjectFileWriterComponents erzeugt wird. Die Auslagerung der Übersetzer in Komponenten vereinfacht die Übersetzung in unterschiedliche proprietäre Formate, insofern als die Komponenten jeweils verschiedenartige Übersetzer zur Verfügung stellen können. Das Paket DocumentComponents bündelt Komponenten zur Produktion der Dokumentation. Beispielsweise produziert die Komponente JsonDocumentComponent eine Instanz einer Klasse zur Generierung der Dokumentation im JSON-Format.

Nach der Vorstellung des Datenmodells der Komponenten folgt die Erläuterung des Vorgehens der Typisierung. Hierbei lassen sich die nativen Zieltypen anhand der Argumentliste und des Rückgabetypen der jeweiligen apply-Methode ableiten. Der semantische Typ sowie die Argumentliste des nativen Typs werden bei den Komponenten dynamisch gebildet, die universal für unterschiedliche Instanzen eines Bestandteils eines Simulationsmodells verwendet werden. Die verbleibenden Komponenten verfügen über eine feste Typspezifikation, so insbesondere die Komponenten zur Übersetzung in die proprietären Formate. Die dynamische Erstellung des semantischen Typen erfolgt nach einem festen Schema, das in der Tabelle 4.2 skizziert ist. Ferner umfasst die Tabelle die Typspezifikationen der Komponenten mit einer nicht variablen Typisierung. In der Tabelle beinhaltet jede Tabellenzeile die Typisierung einer Komponente, wobei die eckigen Klammern den veränderlichen oder genauer gesagt den einzusetzenden Typen entsprechen.

In den ersten vier Zeilen ist das Vorgehen zur Typisierung der Komponenten aus dem Paket Bu-

Komponente	Nativer Typ	Semantischer Typ
`EmbeddedObject-Component`	`EmbeddedObject`	[Bausteinname] ∩ [Bausteintyp]
`ComposedEmbedded-ObjectComponent`	`BuildingBlock`$_0$ → ⋯ → `BuildingBlock`$_n$ → `ComposedEmbeddedObject`	[Argument$_0$] → ⋯ → [Argument$_n$] → [Bausteinname] ∩ [Bausteintyp]
`SelectOutput-Component`	`BuildingBlock`$_0$ → ⋯ → `BuildingBlock`$_n$ → `SelectOutputObject`	[Argument$_0$] → ⋯ → [Argument$_n$] → [Bausteinname] ∩ *SelectOutput*
`SubProcessComponent`	`BuildingBlock`$_0$ → ⋯ → `BuildingBlock`$_n$ → `SubProcessObject`	[Argument$_0$] → ⋯ → [Argument$_n$] → [Subprozessname] ∩ *SubProcess*
`AgentComponent`	`BuildingBlock`$_0$ → ⋯ → `BuildingBlock`$_n$ → `Agent`	[Argument$_0$] → ⋯ → [Argument$_n$] → [Agentenname] ∩ *Agent*
`ProjectFileWriter-Component`	`ProjectFileWriter`	[Simulationsumgebung] ∩ *ProjectFileWriter*
`DocumentWriter-Component`	`DocumentWriter`	[Zielformat] ∩ *DocumentWriter*
`OriginalSimulation-ModelComponent`	`Object`	*SimulationModel* ∩ *Original*
`CompletedSimulation-ModelComponent`	`Agent`$_0$ → ⋯ → `Agent`$_n$ → `SimulationModel`	[Argument$_0$] → ⋯ → [Argument$_n$] → *SimulationModel* ∩ *Completed*
`TranslatedSimulation-ModelComponent`	`ProjectFileWriter` → `DocumentWriter` → `Object` → `Object` → `Object`	*ProjectFileWriter* → *DocumentWriter* → *SimulationModel* ∩ *Original* → *SimulationModel* ∩ *Completed* → *SimulationModel* ∩ *Translated*

Tabelle 4.2: Schema zur Typisierung der Komponenten zur Produktion eines agentenbasierten Simulationsmodells. Die eckigen Klammern symbolisieren Platzhalter, die anwendungsfallabhängig durch Typen ersetzt werden.

ildingBlocksComponents dargestellt. Die Zieltypen dieser Komponenten setzen sich aus der Intersektion des Prozessbausteinnamens und -typs. Der Name bildet einen Teil des Zieltypen, da dieser in Simulationsmodellen häufig eindeutig ist und somit eine eindeutige Identifizierung der Komponente ermöglicht. Die Komponenten, die zusätzliche Prozessbausteine produzieren, (Zeile 2 bis 4) beinhalten in ihrer Typspezifikation darüber hinaus eine Argumentliste. Die Anzahl der Argumente n kann für jede Komponente zwischen 1 und einer beliebigen, aber festen Zahl variieren. Jedoch muss die Anzahl der Argumente im nativen und semantischen Typen übereinstimmen. Während sich die Argumentliste des nativen Typen lediglich in der Anzahl der Argumente unterscheidet, variieren die Argumente im semantischen Typ hinsichtlich der Typen. Die Platzhalter mit den Bezeichnern Argument_0 bis Argument_n werden ersetzt durch die Zieltypen entsprechender Komponenten, die Prozessbausteine produzieren. Daher werden in der Argumentliste des nativen Typs Objekte des Datentyps `BuildingBlock` erwartet. Dies wiederum ermöglicht die Verarbeitung beliebiger Prozessbausteine.

Die Typisierung der Komponente `AgentComponent` erfolgt analog. Auch werden n Argumente im semantischen und nativen Typen erwartet. Ebenso entsprechen die Argumente im semantischen Typen den Zieltypen von Komponenten, welche die Produktion von Prozessbausteinen verantworten. Die Argumentliste des nativen Typs umfasst Argumente des Datentyps `BuildingBlock`. Hingegen wird der semantische Zieltyp dieser Komponente mittels einer Intersektion des Agentennamen und des Typs *Agent* gebildet. Der native Zieltyp ist eine Instanz der gleichnamigen Klasse.

Die Komponente `ProjektFileWriterComponent` erwartet keine Argumente, sodass sich der semantische Typ ausschließlich aus der Intersektion des Namens der Simulationsumgebung, in deren proprietäres Format übersetzt werden soll, und des Typs *ProjectFileWriter* zusammensetzt. Der native Typ dieser Komponente ist eine Instanz der Klasse `ProjectFileWriter`. Ähnlich erfolgt die Typisierung der Komponente `DocumentWriterComponent`, wobei der semantische Zieltyp aus einer Intersektion des Typs *DocumentWriter* und dem Zielformat der Dokumentation besteht. Der native Zieltyp ist eine Instanz der Klasse `DocumentWriter`. Die Typisierung dieser Komponenten ist fest, da diese unabhängig vom zu synthetisierenden Simulationsmodell ist.

Der semantische Typ der Komponente `OriginalSimulationModelComponent` setzt sich aus einer Intersektion der Typen *SimulationModel* und *Original* zusammen. Mittels des Typs *Original* wird ausgedrückt, dass die Komponente das ursprüngliche Simulationsmodell produziert. Der native Typ ist der Datentyp `Object`, der die Elternklasse aller Klassen in Java und Scala darstellt. Damit können Simulationsmodelle in unterschiedlichen proprietären Formaten synthetisiert werden. In dieser Dissertation werden Simulationsmodelle im XML-Format synthetisiert. Die Komponente `SimulationModelComponent`, welche die Agenten produziert, verfügt über einen semantischen Zieltypen, der sich aus einer Intersektion der Typen *SimulationModel* und *FilledWithAgents* zusammensetzt. Der Typ *FilledWithAgents* spiegelt wider, dass das Produktionsergebnis eine Liste mit Agenten umfasst. Aufgrund dessen erwartet die Komponente in der Argumentliste die Zieltypen der Komponenten, welche

die geforderten Agenten synthetisieren. Dementsprechend umfasst die Argumentliste im nativen Typ Argumente des Datentyps `Agent`. Die Top-Level-Komponente `CompletedSimulationModel` verfügt über eine Intersektion der Typen *SimulationModel* und *Completed* als semantischen Zieltypen. Der Typ *Completed* betont, dass das Produktionsergebnis ein vervollständigtes Simulationsmodell darstellt. Die Argumentliste der Top-Level-Komponente umfasst die Zieltypen der Komponenten, die den Übersetzer in ein proprietäres Format, die Instanz zur Erzeugung der Dokumentation, das ursprüngliche Simulationsmodell sowie die Liste mit den Agenten produzieren. Hierbei fällt auf, dass die ersten beiden Argumente unterspezifiziert sind, sodass etwa darauf verzichtet wird, das proprietäre Format der Simulationsmodelle festzulegen. Dadurch werden bei der Synthese mehrere Lösungsvarianten produziert, die sich mindestens hinsichtlich des Formats unterscheiden. An dieser Stelle sei hervorzuheben, dass die Unterspezifizierung optional ist. Stattdessen kann im Rahmen der Typisierung der Top-Level-Komponente auch explizit das proprietäre Format spezifiziert werden.

Mit den zuvor vorgestellten vordefinierten Komponenten können diese dynamisch erzeugt und einer Komponentensammlung hinzugefügt werden, um unterschiedliche Simulationsmodelle zu synthetisieren. Im Rahmen der dynamischen Erstellung der Komponenten erwies sich die wechselnde Anzahl der Argumente bestimmter Komponenten als herausfordernd, da die verwendete Version des CLS-Frameworks keine variablen Argumentlisten in der Implementierung der Komponenten erlaubt. Dies wiederum resultiert in der Notwendigkeit von separaten Implementierungen für jede Komponente mit Argumenten, die sich wiederum hinsichtlich der Argumentanzahl unterscheiden. Beispielsweise müssten n Implementierungen der Komponente `ComposedEmbeddedObjectComponent` vorhanden sein, sofern diese 1 bis n Argumente unterstützen soll. Jedoch führt dieses Vorgehen zu Duplikaten im Quellcode, die wiederum zu einer schlechteren Wartbarkeit und höheren Fehleranfälligkeit führen [95].

Im Nachfolgenden wird eine Lösung dieser Problematik vorgestellt, welche die Argumentlisten der Komponenten auf ein Argument reduziert. Dieses einzelne Argument soll eine Komponente referenzieren, die eine Liste mit den ursprünglichen Argumenten produziert. Jedoch soll die Typisierung der Komponenten, welche die Bestandteile des Simulationsmodells produzieren, weiterhin gemäß der Beschreibung im vorherigen Abschnitt erfolgen. Daher wird nachfolgend ein Verfahren vorgestellt, das basierend auf der Argumentliste einer Komponente automatisiert Komponenten zur Produktion der Listen mit den Argumenten erstellt. Damit muss eine anwendende Person die Aufteilung der Argumentliste einer Komponente nicht selbst vornehmen.

Zur Erläuterung des Vorgehens werden zunächst die Komponenten zur Produktion der Listen vorgestellt. Auch hier werden vordefinierte Komponenten implementiert, die durch eine dynamische Auswahl und Typisierung unabhängig von konkreten Problemstellungen verwendbar sind. Die vordefinierten *Listenkomponenten* sind in der Tabelle 4.3 aufgelistet. Jede Zeile umfasst eine Komponente sowie das Schema zur Typisierung dieser. In Letzterer stellen die eckigen Klammern erneut Platzhalter dar. Die Komponente `EmptyListComponent` produziert eine leere Liste und erwartet aufgrund dessen keine Argumente. Die verbleibenden

Komponente	Semantischer Typ
`EmptyListComponent`	*BuildingBlockList* ∩ *IsEmpty*
`ListWithOneElementComponent`	[Argument$_0$-Zieltyp] → [Verschachtelte Liste] → *BuildingBlockList* ∩ [Listenkennung]
`ListWithTwoElementsComponent`	[Argument$_0$-Zieltyp] → [Argument$_1$-Zieltyp] → [Verschachtelte Liste] → *BuildingBlockList* ∩ [Listenkennung]
`ListWithThreeElementsComponent`	[Argument$_0$-Zieltyp] → [Argument$_1$-Zieltyp] → [Argument$_2$-Zieltyp] → [Verschachtelte Liste] → *BuildingBlockList* ∩ [Listenkennung]

Tabelle 4.3: Komponenten zur Produktion von Listen, die Simulationsbausteine enthalten. Jede Komponente produziert eine Liste mit einer festen Anzahl an Elementen. Die eckigen Klammern repräsentieren Platzhalter, die durch entsprechende Typen ersetzt werden. Der Platzhalter *Verschachtelte Liste* wird durch den Zieltypen einer Listenkomponente ersetzt, sodass eine Verbindung zwischen den Komponenten ermöglicht wird.

Listenkomponenten verantworten die Produktion von Listen, die jeweils ein, zwei oder drei Elemente beinhalten. Die ursprünglichen auszulagernden Argumente werden in der Listenkomponente durch das erste Argument, die ersten zwei oder die ersten drei Argumente der Typspezifikation der Listenkomponente produziert. Das letzte Argument der Listenkomponenten erwartet entspricht dem Zieltypen einer zusätzlichen Listenkomponente, die weitere auszulagernde Argumente enthält. Dieses Argument wird durch den Platzhalter mit der Bezeichnung *Verschachtelte Liste* repräsentiert. Durch dieses Argument wird eine Verbindung zwischen den Listenkomponenten realisiert. Dies wiederum ermöglicht, dass eine beliebige, aber feste Anzahl an Argumenten einer Komponente mittels einer festen Menge an vordefinierten Listenkomponenten ausgelagert werden kann. Zur Identifizierung und Verbindung der Listenkomponenten verfügt jede Komponente, die eine nicht leere Liste produziert, über eine eindeutige Kennung, die einen Teil des semantischen Zieltyps bildet.

Bevor das Vorgehen zur Aufteilung einer Argumentliste einer Komponente vorgestellt wird, sei zunächst auf eine softwaretechnische Herausforderung hingewiesen, die auf die verschiedenartigen nativen Typen der Argumente der aufzuteilenden Komponenten zurückzuführen ist. So umfasst die Argumentliste der Komponente `AgentComponent` den nativen Datentypen `BuildingBlock`, während etwa die Argumente der Komponente `CompletedSimulationModelComponent` Instanzen des nativen Typs `Agent` darstellen. Eine mögliche Lösung besteht in der Implementierung mehrerer Listenkomponenten zur Produktion von Listen, die jeweils Objekte eines Datentyps beinhalten. Jedoch führt dieses Vorgehen zu Duplikaten im Quellcode, sodass nachfolgend eine vom Datentyp unabhängige Implementierung vorgestellt wird, die das Konzept der generischen Klassen nutzt. Hierzu wird bei der Instantiierung einer Listenkomponente der Datentyp der enthaltenen Objekte als Typparameter übergeben. Zur Demonstration wird nachfolgend die Implementierung der Listenkomponente betrachtet,

```
1 class ListWithOneElementComponent[R](val semanticType: Type)
2   extends BuildingBlockListComponent[R] {
3     def apply(element: R, list: List[R]): List[R] = list :+ element
4 }
```

Listing 4.1: Implementierung einer Komponente, die eine einelementige Liste mit Elementen des generischen Typs `R` produziert

die eine einelementige Liste produziert und in Listing 4.1 dargestellt ist. Zunächst sei anzumerken, dass sämtliche Implementierungen der Listenkomponenten von einer abstrakten Klasse `ListComponent` abgeleitet sind. Letztere erwartet einen Typparameter `R`, der somit auch in den Implementierungen der Listenkomponenten erwartet wird. Bei der Instantiierung einer Listenkomponente muss dieser Typparameter explizit angegeben werden. Ferner erwartet der Konstruktor (vgl. Zeile 1) die Angabe eines Typs, der dem semantischen Typen der Listenkomponente entspricht. Die `apply`-Methode der Listenkomponente erwartet ein Objekt `element` des Datentyps `R` sowie eine Liste `list`, die Elemente des Datentyps `R` beinhalten kann. In der Implementierung der Methode wird das Objekt `element` der Liste `list` hinzugefügt. Die erweiterte Liste, die Elemente des Datentyps `R` umfasst, stellt das Produktionsergebnis dar. Der Datentyp `R` wird zur Laufzeit durch einen konkreten Datentypen ersetzt. Durch den Aufruf von `new ListWithOneElementComponent[Agent](semanticType)` kann etwa eine Listenkomponente instantiiert werden, die eine Liste mit Objekten des Datentyps `Agent` produziert.

Wie zuvor erwähnt, erfolgt die Aufteilung der Argumentliste einer Komponente in Listenkomponenten automatisiert. Das Vorgehen sieht eine schrittweise Aufteilung in mehreren Iterationen vor, wobei in jeder Iterationen eine festgelegte Anzahl an Argumenten ausgelagert wird, solange bis keine Argumente mehr auszulagern sind. Durch die Automatisierung der Aufteilung wird es der anwendenden Person ermöglicht, die Komponenten weiterhin gemäß der Tabelle 4.2 zu typisieren. Das Listing 4.2 zeigt den Algorithmus zur automatisierten Aufteilung. Dieser ist als eine Funktion implementiert, die ein Objekt des Datentyps `R` sowie einen semantischen Typen der Komponente, deren Argumentliste aufgeteilt werden soll, als Argumente erwartet. Letzterer wird in der Zeile 2 durch den Aufruf der Funktion `extractGoalTypes` in eine Liste `goalTypes` überführt, welche die aufzuteilenden Argumente enthält. An dieser Stelle sei zu wiederholen, dass die Argumente in Form von Typen vorliegen. In der Zeile 3 wird eine leere Liste, welche die erzeugten Listenkomponenten enthalten soll, instantiiert und der Variable `generatedListComponents` zugewiesen. Die Zeile 4 umfasst die Instantiierung der Komponente `EmptyListComponent`, welche die Produktion einer leeren Liste verantwortet. Auch hier wird der generische Typ `R` als Typparameter übergeben. Anschließend wird diese Listenkomponente in der fünften Zeile der Liste `generatedListComponents` ergänzt und in der Zeile 6 der Variable `previousList` zugewiesen. Letztere ermöglicht den Aufbau der Verbindungen zwischen den einzelnen Listenkomponenten.

In der Zeile 8 beginnt der Durchlauf der Argumentliste mittels einer Schleife. Das Abbruchkriterium der Schleife prüft nach jeder Iteration, ob weitere auszulagernde Argumente vorhanden sein. Trifft dies zu, wird in der Zeile 9 zunächst der Zieltyp, der zu erstellenden Listenkomponente, konstruiert. Hierzu wird eine zufällige, eindeutige Kennung mittels einer Funktion `generateID` generiert. Die Kennung bildet in einer Intersektion mit dem Typen `BuildingBlockList` den semantischen Zieltypen der neuen Listenkomponente. Danach folgt die Konstruktion der Argumentliste im semantischen Typen der Listenkomponente sowie die Instantiierung der vordefinierten Implementierung. Diese erfolgen basierend auf der Anzahl der auszulagernden Argumente in unterschiedlichen `if`-Quellcodeblöcken. Sofern die Liste `goalTypes` genau ein auszulagerndes Argument enthält, wird der Quellcode in den Zeilen 10 bis 14 ausgeführt. Die Aufteilung von zwei auszulagernden Argumenten erfolgt in den Zeilen 15 bis 20. Der dritte und letzte Fall tritt ein, sofern die Liste `goalTypes` mehr als zwei Elemente umfasst. Dann werden zunächst drei Argumente ausgelagert, während die verbleibenden Argumente den nächsten Iterationen vorbehalten bleiben. Die Festlegung auf mehr als ein Argument pro Listenkomponente erfolgt vor dem Hintergrund, dass eine möglichst geringe Anzahl an zusätzlichen Komponenten angestrebt wird, sodass die benötigte Zeitdauer zur Synthese nicht möglicherweise negativ beeinflusst wird. Der Wert von maximal drei Argumenten pro Listenkomponente erwies sich in den durchgeführten Experimenten diesbezüglich als praktikabel.

Die Auslagerung wird nachfolgend anhand der Auslagerung eines einzelnen Arguments erläutert. In diesem Fall wird die `if`-Bedingung in der Zeile 10 erfüllt. Zunächst erfolgt die Vervollständigung des semantischen Typs der Listenkomponente in der Zeile 11 gemäß dem Schema zur Typisierung (vgl. Tabelle 4.3). In der darauffolgenden Zeile 12 wird eine Instanz der Listenkomponente `ListWithOneElementComponent` erzeugt, die den generischen Datentypen `R` sowie den vervollständigten semantischen Typen als Argumente erhält. Die erzeugte Listenkomponente wird in der Zeile 13 der Liste `generatedListComponents` hinzugefügt. Abschließend wird das ausgelagerte Argument der Liste `goalTypes` entnommen und an den Kopf der Schleife gesprungen. Die Implementierungen zur Auslagerung von zwei oder drei Elemente sind analog realisiert und unterscheiden sich lediglich hinsichtlich der Anzahl auszulagernder Elemente sowie der Auswahl der vordefinierten Listenkomponente.

Zur Veranschaulichung des Vorgehens wird nachfolgend die Aufteilung der folgenden semantischen Typspezifikation demonstriert:

$$Arg1 \rightarrow Arg2 \rightarrow Arg3 \rightarrow Arg4 \rightarrow Komponente$$

Diese Typspezifikation wird dem Algorithmus aus Listing 4.2 als Parameter `semanticType` übergeben. Zunächst werden die Argumente *Arg1, Arg2, Arg3* und *Arg4* aus der Typspezifikation extrahiert und der Liste `goalTypes` hinzugefügt. Anschließend wird eine Instanz der Listenkomponente `EmptyListComponent` mit dem semantischen Typen $BuildingBlockList \cap IsEmpty$ erzeugt und der Variable `previousList` zugewiesen.

```
public splitInLists[R](semanticType: Type): Seq[ListComponent[R]] = {
  var goalTypes: Seq[Type] = extractGoalTypes(semanticTypeToSplit)
  var generatedListComponents: Seq[ListComponent[R]] = Seq.empty
  val emptyListComponent = new EmptyListComponent[R]()
  generatedListComponents = generatedListComponents :+ emptyListComponent
  var previousList: ListComponent[R] = emptyListComponent

  while (goalTypes.nonEmpty) {
    val listGoalType: Type = generateID() :&:
     ↪ SemanticTypes.BuildingBlockList
    if (goalTypes.size == 1) {
      val listSemanticType: Type = goalTypes(1) =>:
       ↪ previousList.semanticType =>: listGoalType
      val listComponent = new
       ↪ ListWithOneElementComponent[R](listSemanticType)
      generatedListComponents = generatedListComponents :+ listComponent
      goalTypes = goalTypes.drop(1)
    } else if (goalTypes.size == 2) {
      val listSemanticType: Type = goalTypes(1) =>: goalTypes(2) =>:
       ↪ previousList.semanticType =>: listGoalType
      val listComponent = new
       ↪ ListWithTwoElementComponent[R](listSemanticType)
      generatedListComponents = generatedListComponents :+ listComponent
      goalTypes = goalTypes.drop(2)
    } else {
      val listSemanticType: Type = goalTypes(1) =>: goalTypes(2) =>:
       ↪ goalTypes(3) =>: previousList.semanticType =>: listGoalType
      val listComponent = new
       ↪ ListWithThreeElementsComponent[R](newListSemanticType)
      generatedListComponents = generatedListComponents :+ listComponent
      goalTypes = goalTypes.drop(3)
    }
  }
  generatedListComponents
}
```

Listing 4.2: Algorithmus zur automatisierten Aufteilung der Argumentliste einer Typspezifikation `semanticType` auf unterschiedliche Komponenten, die Listen produzieren. Hierzu wird über die Argumentliste iteriert und in jeder Iteration eine Komponente erzeugt, die einen Teil der Argumentliste produziert. Die einzelnen Listenkomponenten sind mittels ihrer Typspezifikation miteinander verbunden, sodass die ursprüngliche Argumentliste rekonstruiert werden kann. Der Algorithmus gibt die erzeugten Listenkomponenten als Ergebnis zurück.

Darauf folgt der erstmalige Durchlauf der Schleife. Zunächst wird die eindeutige Kennung der zu erzeugenden Listenkomponente generiert, die für dieses Beispiel *1234* lautet. Damit ergibt sich der semantische Zieltyp *List* ∩ *1234* für die zu erstellende Listenkomponente. Danach folgt die Bestimmung der Anzahl der auszulagernden Elemente. Da die Liste `goalTypes` mehr als drei Elemente beinhaltet, folgt die Auslagerung von drei Elementen. Daher werden die Zeilen 20 bis 24 in Listing 4.2 ausgeführt, in denen zunächst der semantische Typ der Listenkomponente gemäß der Tabelle 4.3 konstruiert wird. Hierzu werden drei Argumente aus der Liste `goalTypes` entnommen und mittels des Funktionsoperators → verbunden. Das letzte Argument leitet sich anhand des semantischen Zieltyps der Listenkomponente, die in der Variable `previousList` referenziert ist, ab. Damit ergibt sich folgende semantische Typspezifikation für die zu erstellende Listenkomponente:

$$Arg1 \rightarrow Arg2 \rightarrow Arg3 \rightarrow List \cap IsEmpty \rightarrow List \cap 1234$$

Da drei Argumente ausgelagert werden, wird die Komponente `ListWithThreeElementsComponent` ausgewählt, instantiiert und anschließend der Liste `generatedComponents` hinzugefügt. Ferner wird die erstellte Listenkomponente der Variable `previousList` zugewiesen. Abschließend werden die ausgelagerten Elemente der Liste `goalTypes` entnommen, sodass diese ausschließlich das Argument *Arg4* umfasst.

Danach wird an den Kopf der Schleife gesprungen und die Schleifenbedingung auf ihre Erfüllbarkeit überprüft. Im aktuellen Beispiel wird diese erfüllt, sodass eine weitere Iteration folgt. Zunächst wird erneut der semantische Zieltyp, der zu erstellenden Listenkomponente, konstruiert. Es wird angenommen, dass die Kennung *5678* generiert wurde, sodass der Zieltyp *List* ∩ *5678* lautet. Da genau ein Element zur Auslagerung verbleibt, wird der Quellcodeblock in den Zeilen 10 bis 14 ausgeführt. Analog zur ersten Iteration wird eine Typspezifikation erzeugt, die wie folgt lautet:

$$Arg4 \rightarrow List \cap 1234 \rightarrow List \cap 5678$$

Auch hier wird Bezug auf die Listenkomponente, die in der Variable `previousList` referenziert ist, genommen. Nach der Konstruktion des semantischen Typs folgt die Instantiierung der Komponente `ListWithOneElementComponent` und das Hinzufügen zur Liste `generated-List-Components`. Abschließend wird das ausgelagerte Element aus der Liste `goalTypes` entnommen und an den Kopf der Schleife gesprungen.

Es wird erneut die Schleifenbedingung auf ihre Erfüllbarkeit überprüft. Jedoch kann diese nicht erfüllt werden, da die Liste auszulagernder Elemente leer ist, sodass die Schleife verlassen wird und die erzeugten Listenkomponenten, die in der Variable `generatedComponents` enthalten sind, zurückgegeben werden. Außerhalb des Algorithmus aus Listing 4.2 wird die semantische Typspezifikation der Komponente, deren Argumente ausgelagert wurden, angepasst. Hierbei werden die Argumente durch den Zieltypen der Listenkomponente, die in der letzten Iteration der Aufteilung erzeugt wurde, ersetzt. Somit ergibt sich für das Beispiel die

folgende semantische Typspezifikation:

$$List \cap 5678 \rightarrow Komponente$$

Dieses Vorgehen wird für alle Komponenten einer Komponentensammlung, die eine Argumentliste beinhalten, durchgeführt. In den durchgeführten Experimenten wirkte sich die Aufteilung auf Listenkomponenten nicht auf die Laufzeit der Synthese aus.

4.5 Softwarearchitektur

In diesem Unterkapitel erfolgt die Konzeption der Softwarearchitektur zur Realisierung der komponentenbasierten Synthese von ereignisorientierten Simulationsmodellen. Die Generierung der Simulationsmodelle erfolgt in dieser Dissertation als interner und externer Ansatz. Gemäß der Klassifikation von Bergmann und Straßburger [14] ist die interne Generierung innerhalb einer Simulationsumgebung eingebettet, während der externe Ansatz die Simulationsmodelle außerhalb einer Simulationsumgebung generiert. Da sich die Softwarearchitekturen der Implementierungen grundlegend unterscheiden, werden diese separat vorgestellt.

4.5.1 Externe Simulationsmodellgenerierung

Die Implementierung des externen Ansatzes erfolgt in Form einer Webanwendung nach dem Client-Server-Modell. Die Softwarearchitektur der Webanwendung ist in der Abbildung 4.8 visualisiert. Die anwendende Person (Client) greift über einen Webbrowser auf das Frontend zu, das auf einem Webserver betrieben wird und die Migration von Simulationsmodellen ermöglicht. Das Frontend erlaubt insbesondere das Hochladen eines zu migrierenden Simulationsmodells, die Markierung von Variabilitätspunkten in Form von Anpassungen an der Komponentensammlung und des Syntheseziels sowie das Herunterladen der synthetisierten Simulationsmodellvarianten. Durch das Vorhandensein einer grafischen Benutzeroberfläche bedarf es zur Durchführung der Synthese keinen von der anwendenden Person verfassten Quellcode.

Die Logik der Migration einschließlich der Synthese ist im Backend-System verortet, das auf einem Webserver betrieben wird. Das Backend-System umfasst die Aufteilung eines Simulationsmodells in Simulationsbausteine, die Überführung in getypte Komponenten, die Konstruktion eines Syntheseziels sowie die Durchführung der komponentenbasierten Synthese. Das Backend-System verfügt über eine Instanz des Syntheseframeworks CLS. Ferner ist die Anbindung einer Instanz einer cloudbasierten Simulationsumgebung zur unmittelbaren Ausführung der synthetisierten Simulationsmodellvarianten möglich. Die Kommunikation zwischen dem Frontend und dem Backend-System erfolgt über eine REST-API, die den Austausch mittels JavaScript Object Notation (JSON)-Dateien ermöglicht. Beim JSON-Format

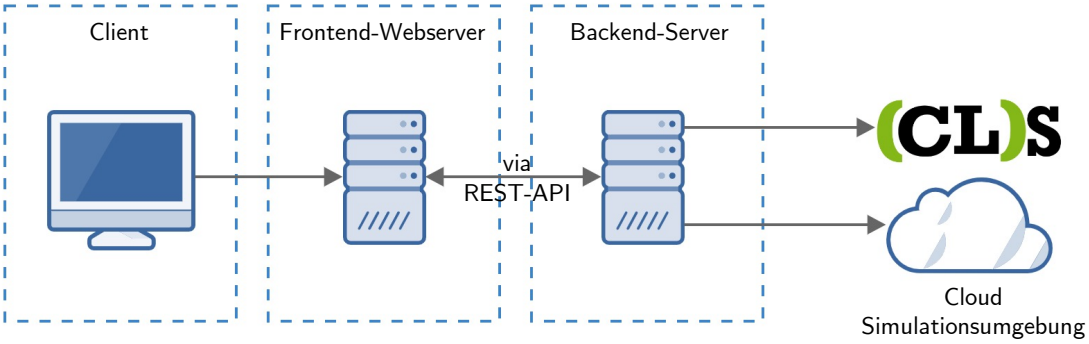


Abbildung 4.8: Softwarearchitektur des externen Ansatzes zur Simulationsmodellgenerierung. Die anwendende Person greift auf dem Client über einen Webbrowser auf eine Webanwendung zu, die auf dem Frontend-Webserver läuft. Letztere kommuniziert über eine REST-API mit einem Backend-Server, der die Programmlogik zur Migration und Synthese der Simulationsmodelle beinhaltet. Mittels einer Referenz auf das Syntheseframework CLS erfolgt die komponentenbasierte Synthese. Die Ausführung der Simulationsmodelle kann durch eine Referenz auf eine Cloud Simulationsumgebung automatisiert werden.

handelt sich um ein Datenaustauschformat, das unabhängig von Programmiersprachen ist und häufig Anwendung in Webschnittstellen oder der Konfiguration von Serversystemen findet. Eine JSON-Datei besteht aus einer Menge von Schlüssel-Wert-Zuweisungen. Die Schlüssel sind Zeichenketten, während die Werte unterschiedliche Ausprägungen wie Zeichenketten, Zahlenwerte, Objekte oder Arrays annehmen können. Mit Objekten werden ineinander geschachteltes JSON-Dateien bezeichnet. [10]

Da sowohl das Backend-System als auch das Frontend auf Webservern betrieben werden, kann der gewählte Ansatz als ein *as-a-Service-Modell* bezeichnet werden. Letzteres bezeichnet den Betrieb und das zur Verfügung stellen einer Anwendung oder Funktionalität, die eine anwendende Person ohne eine lokale Installation nutzen kann. Ferner kann durch die gewählte Softwarearchitektur die Implementierung des Frontends und des Backend-Systems in unterschiedlichen Technologien erfolgen. Auch können unterschiedliche Frontends entwickelt werden, die auf unterschiedliche Zielgruppen zugeschnitten sind. Im Nachfolgenden werden die Anforderungen an das Frontend und das Backend-System eruiert. Weiterhin wird die Struktur und der Inhalt der JSON-Dateien erarbeitet sowie der Datenaustausch zwischen den Systemen erläutert.

4.5.1.1 Bestandteile des Backend-Systems

Das Backend-System ist in drei Schichten unterteilt, die in der Abbildung 4.9 dargestellt sind. Die erste Schicht repräsentiert die Datenschicht, und beinhaltet die Implementierungen der Datenstrukturen zur Repräsentation der Simulationsbausteine sowie der vordefinierten Komponenten (vgl. Kapitel 4.4.2).

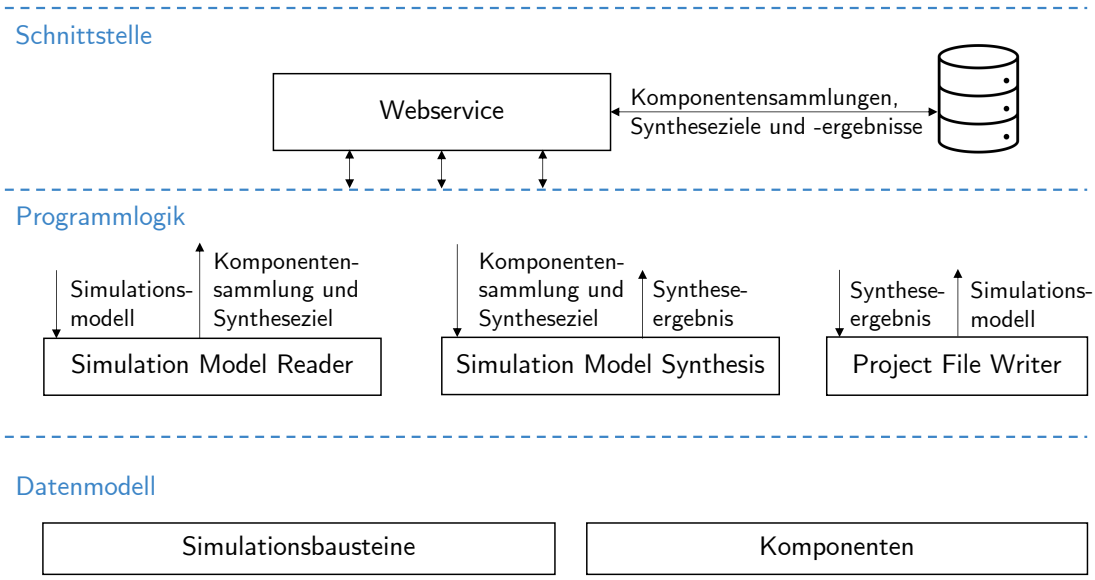


Abbildung 4.9: Softwarearchitektur des Backend-Systems des externen Ansatzes, das aus drei Schichten besteht: 1. das Datenmodell mit den Datenstrukturen zur Repräsentation der Simulationsbausteine und vordefinierter Komponenten, 2. die Programmlogik mit den Algorithmen zur automatisierten Überführung von Kontrollflussgraphen in getypte Komponente, Generierung der Simulationsmodellvarianten sowie Übersetzung in proprietäre Formate und 3. die Schnittstelle mit einer Datenbank, die Komponentensammlungen und synthetisierten Simulationsmodelle enthält, und einer REST-API zur externen Kommunikation.

Die zweite Schicht umfasst die Programmlogik, die sich wiederum aus mehreren Modulen zusammensetzt. Das Modul *SimulationModelReader* verantwortet das Extrahieren von Simulationsbausteinen aus existierenden Simulationsmodellen und die Überführung in getypte Komponenten (vgl. Tabelle 4.2). Ferner konstruiert das Modul ein Syntheseziel, mit dem das ursprüngliche Simulationsmodell synthetisiert werden kann. Somit stellt das Modul *SimulationModelReader* eine Alternative zur händischen Implementierung einer Komponentensammlung dar. Zudem erlaubt das Modul den Export der Komponentensammlung und des Syntheseziels als JSON-Datei, die über die REST-Schnittstelle vom Frontend angefordert werden kann. Letztere ermöglicht der anwendenden Person die Anpassung der extrahierten Komponentensammlung. Daher erfolgt in diesem Modul noch keine Aufteilung der Argumentlisten der Komponenten auf unterschiedliche Listenkomponenten, da eine verschachtelte Struktur der Argumente die Anpassung der Komponentensammlung erschweren würde. Eine detaillierte Beschreibung des Vorgehens zur automatisierten Erstellung der Komponentensammlung erfolgt in Kapitel 6, da in diesem Anwendungsfall die Funktionalität verwendet wird. Im ersten Anwendungsfall (vgl. Kapitel 5) wird die Komponentensammlung händisch erstellt.

Das Modul *SimulationModelSynthesis* verantwortet die Generierung der Simulationsmodellvarianten mittels der komponentenbasierten Synthese. Daher erwartet das Modul eine Kom-

ponentensammlung sowie ein Syntheseziel als Eingaben. Hierbei ist es unerheblich, ob die Komponentensammlung händisch implementiert, automatisiert generiert oder bereits durch die anwendende Person angepasst wurde. Sofern die Komponentensammlung in Form einer JSON-Datei vorliegt (z. B. nach einer Anpassung der Komponenten), wird zunächst eine leere Komponentensammlung instantiiert, ehe die Komponenten dynamisch erzeugt werden. An dieser Stelle erfolgt die Aufteilung der Argumente der Komponenten auf Listenkomponenten. Erst danach werden die aufgeteilten und angepassten Komponenten der Komponentensammlung hinzugefügt.

Anschließend folgt die Durchführung der komponentenbasierten Synthese, gegeben dem übergebenen Syntheseziel. Sofern die Inhabitationsanfrage erfolgreich ist, wird das Syntheseergebnis als ein Objekt des Datentyps `InhabitationResult` zurückgegeben. Letztere ist eine Datenstruktur des Syntheseframeworks CLS, das die synthetisierten Lösungen beinhaltet. Es sei anzumerken, dass die Datenstruktur die Lösungen erst bei einem Abruf interpretiert beziehungsweise erzeugt. Das Modul *SimulationModelSynthesis* bildet eine Basisimplementierung, die im Rahmen der Migration der Anwendungsfälle dieser Dissertation erweitert wird. So werden in Kapitel 5 SMT-Techniken zur Filterung und Sortierung der Lösungen integriert. In Kapitel 6 wird das Modul, um die komponentenbasierte Synthese von Kontrollstrategien ergänzt. Die Programmlogikschicht umfasst weiterhin ein Modul `ProjectFileWriter`, das ein synthetisiertes Simulationsmodell in ein proprietäres Format einer Simulationsumgebung übersetzt. Im Kontext dieser Dissertation wird ein Übersetzer zur Erzeugung von Simulationsmodellen der Simulationsumgebung AnyLogic 8 implementiert.

Die dritte Schicht bildet das *Webservice*-Modul, das die Schnittstelle zwischen den Modulen der Programmlogikschicht bildet. So ermöglicht der Webservice die Kommunikation zwischen den einzelnen Modulen mittels des Austauschs von Nachrichten über die entsprechende REST-Schnittstelle. Das REST-Paradigma wird verwendet, da dieses im Bereich der Webanwendungen eine gängige Methode darstellt. Dabei handelt es sich um eine Programmierschnittstelle, die den Austausch von Ressourcen fokussiert. Das Erstellen, Lesen, Aktualisieren und Löschen von Ressourcen erfolgt mittels entsprechender HTTP-Methoden. Beispielsweise erfolgt die Erstellung einer Ressource über die HTTP-Methode `POST`, während die Aktualisierung durch die HTTP-Methode `PUT` realisiert wird. Die Methoden werden über den HTTP-Aufruf von festgelegten URLs angestoßen. Das REST-Paradigma sieht dabei vor, dass die URL in Verbindung mit der HTTP-Methode selbsterklärend sein sollte. So sollte der Aufruf der URL `/users` mittels der HTTP-Methode `POST` die Erstellung eines neuen Benutzenden anstoßen. Die Ressourcen werden als Nachrichten über das HTTP-Protokoll ausgetauscht. In dem Beispiel enthält die Nachricht unter anderem Informationen über das zu erstellende Benutzerkonto. Die Nachrichten verfügen über einen Rumpf, die häufig eine JSON-Datei zur Repräsentation der Ressource umfassen. Ferner verfügen die Nachrichten über einen Kopf mit Metainformationen bezüglich der Nachricht (z. B. Datenformat der Nachricht) sowie der Kommunikation (z. B. Schlüssel für nicht öffentliche Schnittstelle). [56]

Außerdem verfügt die Schnittstellen-Schicht über eine Instanz einer Datenbank, welche

die hochgeladenen Simulationsmodelle sowie die extrahierten Komponentensammlungen persistiert. Überdies beinhaltet die Datenbank die synthetisierten Simulationsmodelle. Durch das Vorhandensein einer Kommunikation erfordert die Kommunikation der Module über die REST-Schnittstelle keinen zwingenden Austausch aller Daten. Beispielsweise genügt beim Aufruf einer REST-Schnittstelle die Angabe einer eindeutigen Kennung eines hochgeladenen Simulationsmodells als URL-Parameter. Die Implementierung fragt anhand der übergebenen Kennung automatisiert die benötigten Daten aus der Datenbank ab.

Die Realisierung des *WebService*-Moduls erfolgt mittels des Frameworks *Scalatra*[1], das zur Implementierung der REST-Schnittstellen verwendet wird. Ferner wird das Framework *Swagger*[2] eingesetzt, um die Dokumentation der REST-Schnittstellen automatisiert zu generieren. In der Tabelle 4.4 ist eine Auswahl der implementierten Schnittstellen des *WebService*-Moduls aufgelistet. Mittels eines `:` werden Platzhalter in einer URL dargestellt. So stellt `:id` einen Platzhalter dar, der die Kennung eines Simulationsmodells repräsentiert und beim Aufruf REST-Schnittstelle durch eine existierende Kennung ersetzt wird. Die Schnittstellen umfassen den gesamten Ablauf des Migrationsansatzes, beginnend vom Hochladen eines Simulationsmodells bis zum Herunterladen der synthetisierten Lösungen.

4.5.1.2 Umsetzung des webbasierten Frontends

Das Frontend bildet die Schnittstelle zwischen einer anwendenden Person und der Programmlogik im Backend-System. Das Frontend wird derart gestaltet, dass eine benutzerfreundliche und intuitive Verwendung der kombinatorischen Logiksynthese möglich ist, gänzlich ohne Vorkenntnisse der theoretischen Grundlagen. Es wird lediglich ein prinzipielles Verständnis der komponentenorientierten Komposition von Lösungen vorausgesetzt. Die Umsetzung des Frontends erfolgt als eine Webanwendung, die auf beliebigen netzwerkfähigen Geräten, die über einen Internetbrowser verfügen, verwendet werden kann. Die Implementierung des Frontend-Systems erfolgt in JavaScript unter Verwendung der Frameworks npm[3], Vue.js[4] und React.js[5].

Das Frontend leitet durch den Migrationsprozess in vier Schritten. Im ersten Schritt wird ein Simulationsmodell hochgeladen oder ein zuvor hochgeladenes Simulationsmodell ausgewählt. Im zweiten Schritt wird die extrahierte Komponentensammlung in einer tabellarischen Sicht angezeigt, die in der Abbildung 4.10 dargestellt ist. In dieser Tabelle entspricht jede Zeile einer Komponente, die dupliziert, entfernt oder angepasst werden können.

Nach der Anpassung der Komponentensammlung kann im dritten Schritt eine Komponentisierung der Kontrollstrategien eines Simulationsmodells vorgenommen werden, die in Kapi-

[1] Scalatra - https://scalatra.org/
[2] Swagger für Scala - https://github.com/swagger-api/swagger-scala-module
[3] npm - https://www.npmjs.com/
[4] Vue.js - https://vuejs.org/
[5] React.js - https://reactjs.org/

HTTP-Methode	URL	Beschreibung
`POST`	`/`	Hochladen eines zu migrierenden Simulationsmodells, das im Backend-System eine eindeutige Kennung erhält und persistiert wird. Die Schnittstelle gibt die Kennung zur nachfolgenden Referenzierung des Simulationsmodells zurück.
`GET`	`/:id/repository`	Abrufen der Komponentensammlung und des Syntheseziels für das Simulationsmodell mit der angefragten Kennung.
`PUT`	`/:id/repository`	Aktualisieren der Komponentensammlung und des Syntheseziels.
`GET`	`/:id/synthesize`	Anstoßen der Inhabitationsanfrage. Nach Abschluss wird die Anzahl der synthetisierten Lösungen zurückgegeben.
`GET`	`/:id/solutions/:number`	Abrufen einer konkreten Lösungsvariante für ein Simulationsmodell (erst nach einer Inhabitationsanfrage möglich). Der URL-Parameter `:number` umfasst eine Ziffer n mit $n \geq 0$, die der n-ten synthetisierten Lösungsvariante entspricht. Daher muss die Anzahl der synthetisierten Lösungen größer gleich dieser Ziffer sein.

Tabelle 4.4: Auswahl der REST-Schnittstellen, die im Backend-System implementiert sind und eine Kommunikation mit externen Systemen ermöglichen.

tel 6 erläutert wird. Nachdem im Frontend die Durchführung der Synthese angestoßen und abgeschlossen wurde, werden im vierten Schritt die synthetisierten Simulationsmodelle in tabellarischer Form angezeigt. Letztere ist in der Abbildung 4.11 dargestellt. Jede Tabellenzeile repräsentiert eine synthetisierte Simulationsmodellvariante. Es kann eine einzelne Lösung oder eine archivierte Datei mit allen Lösungen heruntergeladen werden. Überdies umfasst die archivierte Datei ein ausführbares Skript, das die synthetisierten Simulationsmodelle automatisiert in einer Simulationsumgebung öffnet.

Neben der tabellarischen Ansicht der Komponentensammlung verfügt das Frontend über einen grafischen Editor, der ebenso Anpassungen an den Komponenten ermöglicht. Dieser wurde im Rahmen einer betreuten studentischen Abschlussarbeit [107] konzeptioniert und implementiert. Der Editor stellt einen weiteren Schritt zur Senkung der Einstiegshürde dar, als dieser eine grafische Anpassung und Spezifikation der Typen ermöglicht. Die Abbildung 4.12 zeigt einen Screenshot des Editors. In dem Editor werden die Komponenten durch grüne Rechtecke und die Argumente dieser durch graue Rechtecke dargestellt. Durch das Hinziehen einer Komponente aus der Liste der Komponenten (vgl. linke Spalte) in die grünen Rechtecke können Argumente ergänzt werden. Analog resultiert das Herausziehen eines Arguments aus einem grünen Rechteck in der Entfernung des Arguments aus der Argument-

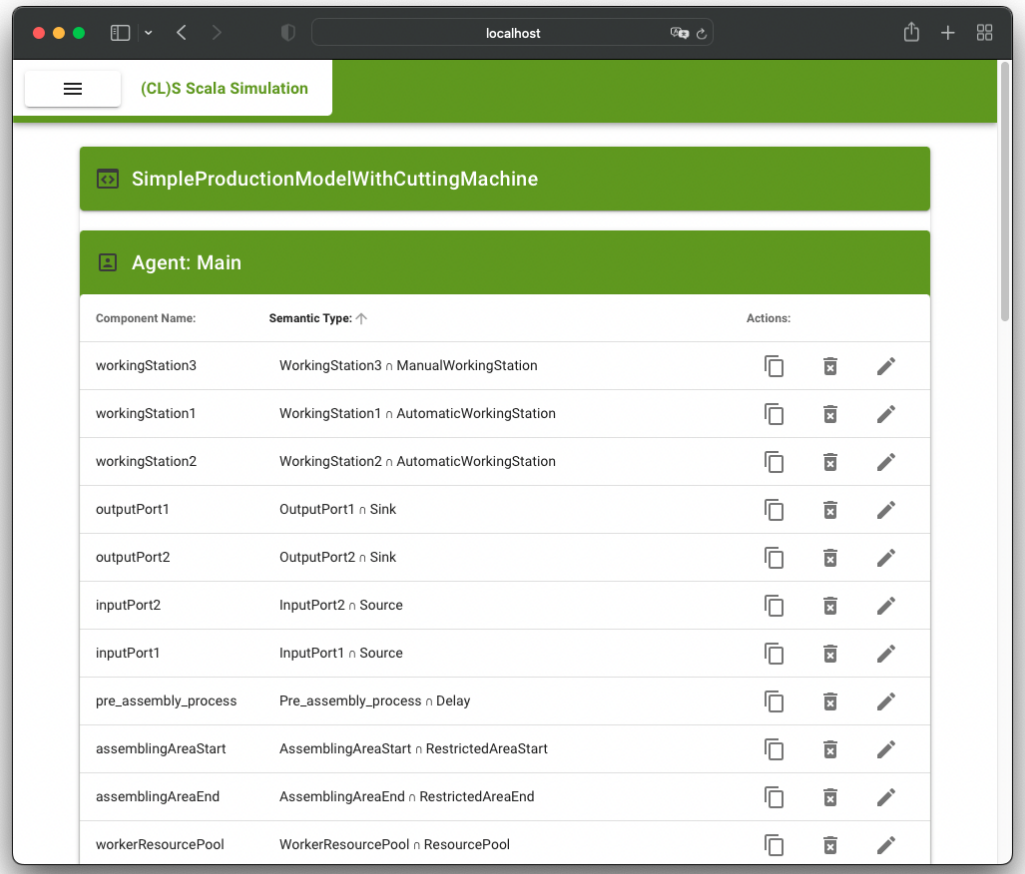


Abbildung 4.10: Screenshot des Frontends, das eine Komponentensammlung für ein exemplarisches Simulationsmodell zeigt. Die Tabellenzeilen repräsentieren Komponenten mit dem Namen, semantischen Typen und Schalter zum Duplizieren, Entfernen und Anpassen einer Komponente (v. l. n. r.).

liste einer Komponente. Die Typspezifikationen der Komponenten werden im Hintergrund automatisch angepasst. Sofern ein Argument einer Komponente zur Bildung von Varianten unterspezifiziert werden soll, kann der semantische Typ eines Arguments händisch angepasst werden. Ferner erlaubt der Editor die Parametrisierung eines Prozessbausteins oder Agenten, der durch eine Komponente produziert werden, sowie die Erstellung und das Löschen von Komponenten. Der Editor ist im Frontend integriert, sodass zu jedem Zeitpunkt zwischen der tabellarischen Ansicht und dem Editor gewechselt werden kann. Der Editor stellt eine beispielhafte Erweiterung des Frontends dar. Überdies kann das Frontend durch anwendungsfallspezifische Erweiterungen ergänzt werden. Beispielsweise wird im Rahmen des ersten Anwendungsfalls eine Benutzeroberfläche ergänzt, die eine benutzerfreundliche Angabe von Nebenbedingungen erlaubt.

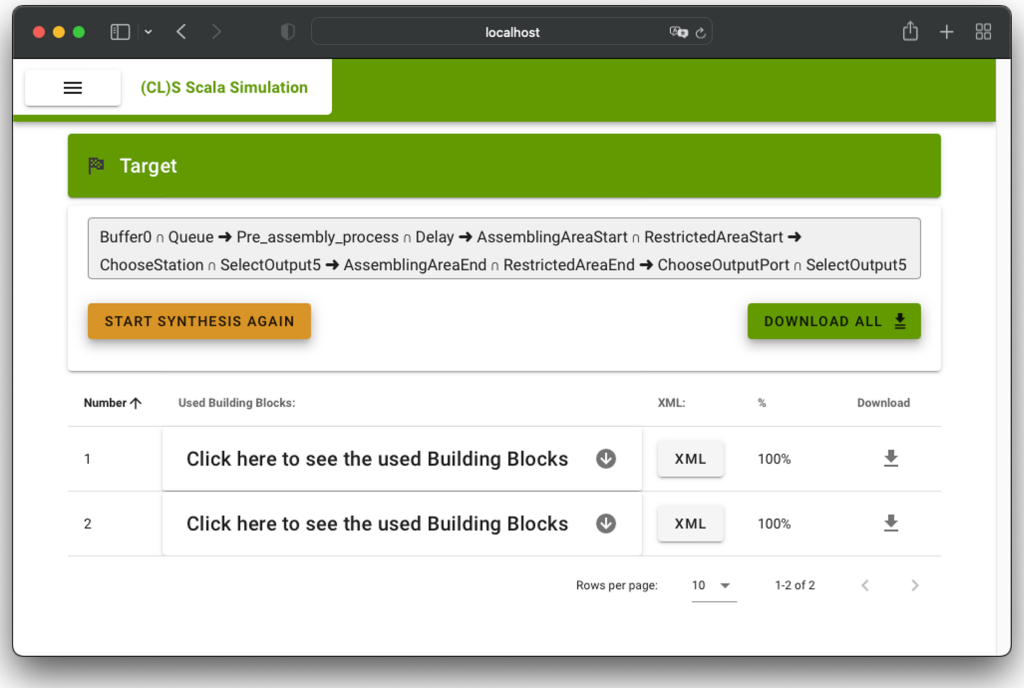


Abbildung 4.11: Screenshot des Frontends der entwickelten Technologie, das die synthetisierten Simulationsmodelle in einer tabellarischen Ansicht zeigt. Die Tabellenzeilen entsprechen einzelnen Lösungen.

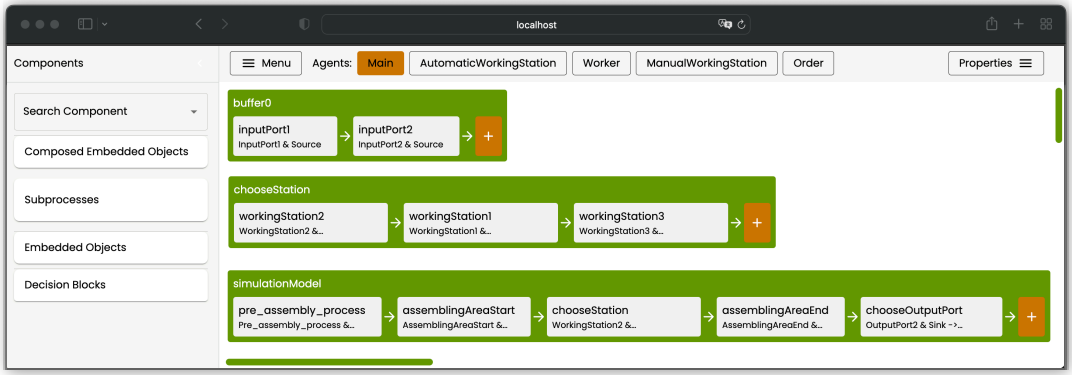


Abbildung 4.12: Screenshots des Editors, der eine grafische Anpassung der Typspezifikationen der Komponenten sowie des Syntheseziels ermöglicht. Die grünen Kästen repräsentieren Komponenten und die darin platzierten Kästen entsprechen den Argumenten einer Komponente. In der linken Spalte sind die vorhandenen oder ergänzten Komponenten aufgelistet.

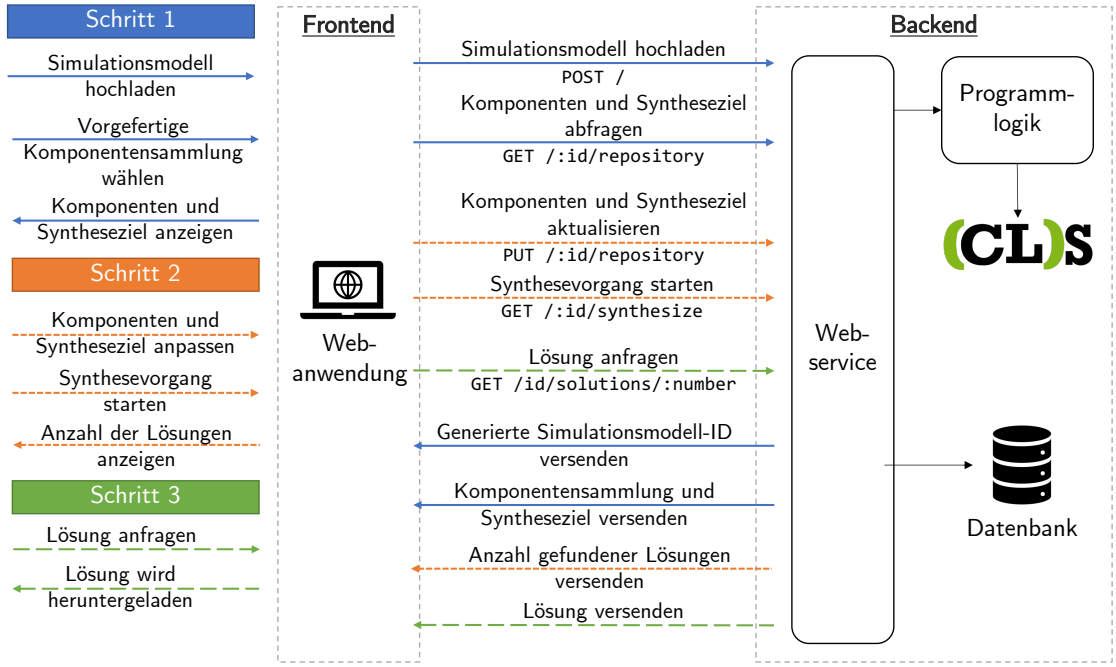


Abbildung 4.13: Ablauf und Softwarearchitektur der Implementierung des externen Ansatzes zur Synthese der Simulationsmodelle (Quelle: In Anlehnung an Kallat et al. [59]). Die Migration verläuft in drei aufeinanderfolgenden Schritte, die im Frontend angestoßen werden. Die Abbildung zeigt die aufgerufenen REST-Schnittstellen des Backend-Systems sowie die Antworten.

4.5.1.3 Datenaustausch mittels JSON über REST-API

Im Folgenden wird der Datenaustausch zwischen dem Frontend und dem Backend über die REST-Schnittstelle thematisiert. Die Abbildung 4.13 skizziert die Kommunikationswege, beginnend vom Hochladen der Simulationsmodelle bis zum Herunterladen synthetisierter Lösungsvarianten. Die einzelnen Schritte werden nachfolgend erläutert.

Der erste Schritt (vgl. blau gestrichelte Anfragen und Antworten in der Abbildung 4.13) sieht das Hochladen eines Simulationsmodells vor, das im Backend-System in eine Komponentensammlung überführt und ein Syntheseziel konstruiert wird. Nach der Überführung antwortet das Backend-System über die REST-Schnittstelle mit dem Statuscode 200, der eine erfolgreiche Antwort repräsentiert. Weiterhin beinhaltet die Antwort die zugewiesene Kennung des hochgeladenen Simulationsmodells. Eine beispielhafte Antwort ist in Listing 4.3 dargestellt. Das hochgeladene Simulationsmodell, die Komponentensammlung sowie das Syntheseziel sind über die Kennung im Schlüssel `id` referenzierbar, in diesem Beispiel lautet die Kennung *bd7...b7f*. Alternativ kann eine händisch implementierte Komponentensammlung ausgewählt werden. Hierzu kann die eindeutige Kennung des geforderten Simulationsmodells mittels der REST-Schnittstelle `GET /` aufgerufen werden. Letztere gibt sämtliche Kennungen der bereits hochgeladenen Simulationsmodelle zurück. In beiden Fällen bildet die eindeutige Kennung einen Bestandteil der URL der nachfolgenden REST-Anfragen.

```
1 {
2     "id": "bd742299-9876-4352-81e7-39d7419d8b7f"
3 }
```

Listing 4.3: Antwort des Backend-Systems nach dem Hochladen eines Simulationsmodells über die REST-API. Der Wert `id` kann nachfolgend zur Referenzierung des hochgeladenen Simulationsmodells verwendet werden.

Nach dem Hochladen oder der Auswahl eines Simulationsmodells wird die Komponentensammlung und das Syntheseziel über die entsprechende REST-Schnittstelle abgerufen. Das Ergebnis dieses Aufrufs ist eine JSON-Datei, die aus JSON-Objekten besteht, die wiederum die Komponentensammlungen inklusive der Syntheseziele der einzelnen Agenten enthalten. Das Listing 4.4 zeigt eine beispielhafte Antwort auf die REST-Anfrage `GET /bd7...b7f/repository` für das zuvor exemplarisch hochgeladene Simulationsmodell. In der JSON-Dateien entspricht die Zeichenkette `->` dem Funktionsoperator $\rightarrow$, während `&` der Intersektion $\cap$ entspricht. Die Komponentensammlungen sind nach Agenten aufgeteilt, da für jeden Agenten eine separate Komponentensammlung vorgesehen ist. Somit erfolgt auch die Durchführung der Synthese für die Agenten unabhängig voneinander. Dieses Vorgehen ermöglicht die Parallelisierung der Synthese einzelner Agenten sowie eine reduzierte Rechenzeit, da die Kombinatorik eines einzelnen Agenten geringer ist im Vergleich zu einer Komponentensammlung, die Komponenten für die Produktion sämtlicher Agenten umfasst.

Die JSON-Objekte zur Repräsentation der Agenten umfassen den Namen des Agenten sowie das Syntheseziel. Letzterer ist im Schlüssel `target` enthalten und repräsentiert die Typspezifikation der Top-Level-Komponente zur Produktion des Agenten. Ferner umfassen die JSON-Objekte ein Objekt mit dem Schlüssel `components`, das verschiedene Arrays mit den Komponenten zur Produktion der Prozessbausteine enthält. Jedes Array umfasst JSON-Objekte, die einer vordefinierten Komponente entsprechen. So enthält das Array mit dem Schlüssel `selectOutputs` die Komponenten des Datentyps `SelectOutputComponent`, welche die Produktion von Entscheidungsprozessbausteine und den dazugehörigen Pfaden verantworten. Die JSON-Objekte in den einzelnen Arrays repräsentierten die einzelnen Komponenten und bestehen mindestens aus einem eindeutigen Namen und einer semantischen Typspezifikation. Die weiteren Bestandteile der JSON-Objekte unterscheiden sich je nach vordefinierter Komponente. So enthalten die Komponenten zur Produktion von Prozessbausteinen die Metainformationen der Prozessbausteine. Beispielsweise sind diese Metainformationen für die Komponente *inputPort1* im Objekt mit dem Schlüssel `properties` (vgl. Zeile 10 bis 16) hinterlegt. Ferner verfügt das JSON-Objekt über ein Array mit dem Schlüssel `parameters` mit den Parametern des Prozessbausteins. Das Vorhandensein der Metainformationen und der Parametrisierung ermöglicht die Anpassung dieser.

An dieser Stelle sei anzumerken, dass die JSON-Repräsentation der Komponenten, die zusätzlich die angebundenen Pfade eines Prozessbausteins produzieren, nicht die Metainformatio-

nen aller auf dem Pfad liegenden Prozessbausteine enthalten. Ferner sei zu erwähnen, dass die JSON-Repräsentation der Komponenten des Datentyps `ComposedEmbeddedObjectComponent` über Felder verfügt, welche die Positionierung des zu produzierenden Pfades sowie das Vorhandensein eines gemeinsamen Elements der Pfade umfassen (vgl. Zeile 27 bis 29). Die JSON-Datei wird in der Benutzeroberfläche verarbeitet und in der tabellarischen Ansicht und im Editor verarbeitet.

Im zweiten Schritt erfolgen die Anpassungen an den Komponentensammlungen sowie der Syntheseziele (vgl. orange gestrichelte Anfragen und Antworten in der Abbildung 4.13). Diese Anpassungen führt die anwendende Person im Frontend durch und werden vom Frontend über die REST-Schnittstelle `PUT /:id/repository` an das Backend weitergeleitet. Hierzu wird eine angepasste Komponentensammlung (vgl. Listing 4.4) versendet und das Backend-System ersetzt damit die vorherige Version der Komponentensammlung. Nach der Bestätigung der erfolgreichen Aktualisierung des Backend-Systems mittels einer Antwort, die den Statuscode 200 enthält, kann die anwendende Person die Durchführung der Synthese anstoßen. Hierzu ruft das Frontend die Schnittstelle `GET /:id/synthesize` auf, die nach der Beendigung der Synthese im Backend-System eine Antwort zurückgibt, welche die Anzahl der gefundenen Lösungen umfasst. Das Listing 4.5 zeigt eine beispielhafte Antwort für den Fall, dass das Backend-System zehn Lösungen synthetisierte.

Somit umfasst die unmittelbare Antwort des Backend-Systems keine konkreten Lösungen. Letztere werden im dritten Schritt (vgl. grün gestrichelte Anfragen und Antworten in der Abbildung 4.13) durch entsprechende Anfragen an das Backend-System angefragt. Hierbei können sowohl einzelne als auch sämtliche Lösungen angefordert werden. Sofern eine einzelne Lösung angefragt wird, wird die entsprechende Nummer der Lösung angegeben. Wird etwa die zweite Lösung einer Lösungsmenge angefordert, so wird die REST-Schnittstelle `GET/:id/solutions/1` aufgerufen. Die 1 in der URL entspricht der zweiten Lösung gemäß der gängigen Nummerierung in der Informatik. Weiterhin sei zu erwähnen, dass erst bei der Anforderung einer Lösung diese auch tatsächlich interpretiert und das ausführbare Simulationsmodelle produziert wird. In der Antwort einer Lösungsanfrage ist das synthetisierte Simulationsmodell als Zeichenkette in der JSON-Datei enthalten. Ferner enthält die Antwort eine Auflistung der ausgewählten Komponente einer synthetisierten Lösung. Das Frontend verfügt über die notwendige Funktionalität, um aus der gegebenen Zeichenkette eine Datei zu konstruieren und zum Herunterladen anzubieten. Sofern die synthetisierten Simulationsmodelle in einer Cloud-Simulationsumgebung ausgeführt werden können, würde diese durch das Backend-System ausgeführt werden. In diesem Szenario könnte das Frontend über REST-Schnittstellen die Ausführung anstoßen sowie die Ergebnisse nach der Simulation abrufen. Ferner sei zu erwähnen, dass die (De-)Serialisierung sowohl im Frontend als auch im Backend-System mittels entsprechender Bibliotheken automatisiert erfolgt.

```
{
 "agents": [
   {
    "name": "Main",
    "target": "Buffer0 & Queue -> Pre_assembly_process & Delay -> ... ->
     ↪ Delivery & Sink",
    "components": {
      "embeddedObjects": [
       {
        "name": "inputPort1",
        "properties": {
          "id": "1585901908879",
          "blockType": "Source",
          "posX": "160",
          "posY": "110",
          "packageName": "com.anylogic.libraries.processmodeling"
        },
        "parameters": [...],
        "semanticType": "InputPort1 & Source"
       }
       ...
      ],
      "composedEmbeddedObjects": [
       {
        "name": "buffer0",
        "properties": {...},
        "parameters": [...],
        "hasPredecessors": true,
        "hasSuccessors": false,
        "hasCommonSuccessor": false,
        "semanticType": "InputPort1 & Source -> InputPort2 & Source ->
         ↪ Buffer0 & Queue"
       }
       ...
      ],
      "selectOutputs": [...],
      "subProcesses": [...]
   },
   ...
 ]
}
```

Listing 4.4: Gekürzte Antwort des Backend-Systems für die Anfrage der Komponentensammlung und des Syntheseziels für ein hochgeladenes Simulationsmodell über die REST-API. Die Antwort der Anfrage besteht aus einer JSON-Datei, die für jeden Agenten des Simulationsmodells das Syntheseziel `target` sowie die entsprechenden Komponenten `components` zur Produktion der Prozessbausteine umfasst.

```
{
    "numberSolutions": 10
}
```

Listing 4.5: Antwort des Backend-Systems nach der Ausführung des Synthesevorgangs, das die Anzahl der gefundenen Lösung enthält.

4.5.2 Interne Simulationsmodellgenerierung

Bei der internen Simulationsmodellgenerierung findet die Generierung des Simulationsmodells innerhalb einer Simulationsumgebung statt. Hierfür wird vorausgesetzt, dass die verwendete Simulationsumgebung die Einbindung externer Bibliotheken oder eine externe Kommunikation außerhalb der Systemgrenzen des Simulationsmodells erlaubt. Letzteres wäre etwa mittels des Zugriffs auf das Dateisystem oder durch HTTP-Anfragen möglich. Die Steuerung der internen Simulationsmodellgenerierung kann mittels von Konfigurationsdateien oder grafischen Benutzeroberflächen innerhalb eines Simulationsmodells erfolgen.

Ein Vorteil der internen Simulationsmodellgenerierung besteht darin, dass keine Integration einer weiteren Anwendung in eine Systemlandschaft notwendig ist. Ferner erlaubt die interne Simulationsmodellgenerierung, dass in einigen Fällen die lokale Installation einer Simulationsumgebung entfallen kann. Dies ist dann der Fall, wenn eine Simulationsumgebung das Exportieren der Simulationsmodelle als ausführbare Programme ermöglicht, die ohne die entsprechende Simulationsumgebung ausgeführt werden können. Dieses Szenario ist insofern relevant, als die Nutzung einer Simulationsumgebung häufig mit hohen Lizenzkosten verbunden sind und dies wiederum zur Folge hat, dass nicht jede Arbeitskraft über eine entsprechende Lizenz verfügt. Jedoch ist Letztere bei der externen Simulationsmodellgenerierung erforderlich, da die Simulationsmodelle in den proprietären Formaten der Simulationsumgebungen zur lokalen Ausführung erzeugt werden.

Als eine mögliche Lösung kann der Einsatz von Cloud-Simulationsumgebungen zur Ausführung der generierten Simulationsmodelle genannt werden. Da die cloudbasierten Simulationsumgebungen oftmals ausschließlich die Ausführung eines Simulationsmodells ermöglichen, fallen die Lizenzkosten häufig geringer aus im Vergleich zur lokalen Simulationsumgebung. In diesem Fall würden die Simulationsmodelle nach der Generierung in der Cloud-Simulationsumgebung zur Verfügung gestellt werden. Jedoch setzt dies voraus, dass die generierten Simulationsmodelle ohne Verwendung einer lokalen Simulationsumgebung in die Cloud-Simulationsumgebung hochgeladen werden können. Letzteres ist allerdings nicht in jeder Cloud-Simulationsumgebung möglich. Weiterhin erweist sich der Einsatz einer cloudbasierten Lösung nicht in jedem Szenario als praktikabel. Zum Beispiel, wenn die Simulationsmodelle vertrauliche Informationen verarbeiten oder eine anwendende Person über keine stabile und ausreichend schnelle Internetverbindung verfügen.

Als ein Nachteil der internen Simulationsmodellgenerierung können die Einschränkungen

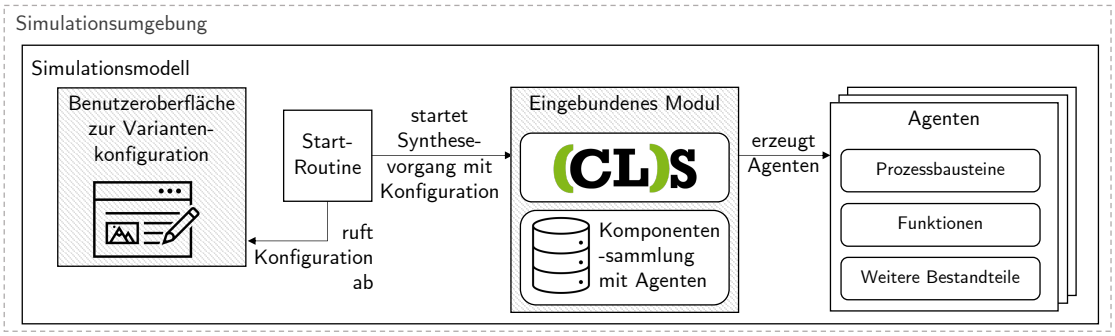


Abbildung 4.14: Ablauf und Softwarearchitektur der Implementierung des internen Ansatzes zur komponentenbasierten Synthese von Simulationsmodellen mittels CLS. Der Ansatz ist innerhalb eines Simulationsmodells eingebettet und besteht aus folgenden Elementen: 1. Bedienoberfläche zur Konfiguration der geforderten Variantenbildung, 2. Modul mit einer anwendungsfallspezifischen Komponentensammlung und einer Referenz auf das Syntheseframework CLS sowie 3. Startroutine, die das Modul zur Synthese aufruft, die synthetisierten Bestandteile dem Simulationsmodell hinzufügt und die Simulation startet.

genannt werden, die sich durch das Ökosystem der Simulationsumgebung ergeben und Einfluss auf die Implementierung nehmen. Dies betrifft etwa die Programmiersprache oder die verwendete Version der Laufzeitumgebung einer Simulationsumgebung. Beispielsweise läuft die Simulationsumgebung AnyLogic 8 auf der Java Virtual Machine und verwendet die Programmiersprache Java, sodass ein Ansatz zur Simulationsmodellgenerierung ebenso in einer Programmiersprache der Java Virtual Machine implementiert sein sollte.

Gleichwohl stellt die interne Simulationsmodellgenerierung häufig eine sinnvolle Lösung dar. Daher untersucht diese Dissertation auch die Umsetzung der internen Generierung im Kontext der Migration eines Simulationsmodells. Die Abbildung 4.14 skizziert den schematischen Ablauf der internen Simulationsmodellgenerierung im Kontext der kombinatorischen Logiksynthese. Der äußere Rahmen symbolisiert die Einbettung in das Ökosystem einer Simulationsumgebung. In dieser Einbettung ist eine grafische Benutzeroberfläche zur Konfiguration der zu synthetisierenden Simulationsmodelle integriert. Die Konfiguration ermöglicht insbesondere die Festlegung von Variabilitätspunkten. An dieser Stelle sei anzumerken, dass die Konfiguration und die Benutzeroberfläche vom jeweiligen Anwendungsfall abhängig sind.

Basierend auf der Konfiguration werden Eingabedaten mit den notwendigen Informationen zur Durchführung der komponentenbasierten Synthese erzeugt, die in der Startroutine des Simulationsmodells verarbeitet werden. Letztere umfasst eine Reihe von Befehlen, die beim Start eines Simulationsmodells ausgeführt und häufig durch Funktionen in einer Programmiersprache implementiert werden. Die Startroutine stellt einen zentralen Ausgangspunkt dar, da diese die komponentenbasierte Synthese anstößt und die Ergebnisse der Synthese verarbeitet. Hierfür instantiiert die Startroutine zunächst ein Modul, bestehend aus einer Instanz des Syntheseframeworks CLS und einer Komponentensammlung. Ebenso wie die Konfiguration und die Benutzeroberfläche ist die Implementierung dieses Modul auf den

Anwendungsfall zugeschnitten. Das Modul wird als eine Abhängigkeit des Simulationsmodells in das Ökosystem integriert. Dadurch kann die Implementierung des Moduls auch außerhalb der Simulationsumgebung in externen Entwicklungsumgebungen erfolgen. Die händisch implementierte Komponentensammlung umfasst Komponenten zur Produktion der Bestandteile des Simulationsmodells wie Agenten oder Prozessbausteine. Die Startroutine ruft eine Schnittstelle des Moduls mit der Konfiguration als Argument auf und das Modul führt die komponentenbasierte Synthese durch. Nach der Beendigung der Synthese werden die synthetisierten Lösungen an die Startroutine zurückgegeben. Letztere fügt die synthetisierten Bestandteile aus den Lösungen dynamisch in das Simulationsmodell ein. Abschließend wird die Simulation gestartet und die anwendende Person kann nach der Beendigung der Simulation die Evaluation der Simulationsergebnisse durchführen. Der dritte Anwendungsfall (vgl. Kapitel 7) in dieser Dissertation implementiert die Migration und komponentenbasierte Synthese von Simulationsmodellen als einen internen Ansatz. Daher umfasst das Kapitel 7 eine detaillierte Beschreibung des Vorgehens.

4.6 Auswahl der Simulationsumgebung

Während in den vorherigen Abschnitten keine Festlegung auf eine konkrete Simulationsumgebung erfolgte, erfolgt diese nun anhand der Auswertung verschiedener Kriterien. Im Rahmen dieser Dissertation wird die Simulationsumgebung AnyLogic 8 verwendet, da diese die nachfolgenden Kriterien erfüllt.

Das erste Kriterium bezieht sich die Verbreitung der Simulationsumgebung in der Forschung und in der industriellen Anwendung. Hierbei wird insbesondere die Domäne der Produktion und Logistik berücksichtigt. Dieses Kriterium wird von der Simulationsumgebung AnyLogic 8 erfüllt, da diese in Industrieprojekten als auch in der Forschung im Kontext der genannten Domäne verwendet wird [31, 85].

Das zweite Kriterium nimmt Bezug auf technische Gesichtspunkte. Im Kontext der externen Generierung ist ein strukturiertes und maschinenlesbares Format vorteilhaft, um eine automatisierte Verarbeitung und Generierung von Simulationsmodellen zu ermöglichen. Dies Kriterium erfüllt AnyLogic 8 insofern, als die Simulationsmodelle im XML-Format vorliegen. Dieses Format kann in zahlreichen Programmiersprachen mittels entsprechender Bibliotheken programmatisch verarbeitet werden. Ein weiterer technischer Gesichtspunkt betrifft die Fähigkeit, bei der Modellierung eines Simulationsmodells die Kontrollstrategien mittels Funktionen in höheren Programmiersprachen zu realisieren. Dadurch ist ein Einsatz der komponentenbasierten Synthese zur Generierung der Kontrollstrategien möglich. Auch diese Anforderung erfüllt AnyLogic 8, da die Programmiersprachen Java und Python bei der Modellierung verwendet werden können. Überdies erweist sich die Unterstützung der Programmiersprache Java hinsichtlich der internen Simulationsmodellgenerierung als vorteilhaft. Dies ermöglicht eine unaufwändige Integration des Syntheseframeworks in das Simulations-

modell, da die Simulationsumgebung und die Implementierung dieser Dissertation auf der Java Virtual Machine lauffähig sind.

Das dritte Kriterium umfasst das Vorhandensein einer cloudbasierten Erweiterung, welche die parallele Ausführung der synthetisierten Simulationsmodelle erlaubt. Ferner ist eine cloudbasierte Ausführung in Bezug auf maschinelle Lernverfahren von Bedeutung, da damit eine Vielzahl unterschiedlicher Lösungsvarianten automatisiert ausgeführt, evaluiert und die Simulationsergebnisse zum Lernen verwendet werden können. Dieses Kriterium wird durch die Erweiterung der AnyLogic Cloud [22] auch von der Simulationsumgebung erfüllt.

Abschließend sei anzumerken, dass die nachfolgenden Vorgehensweisen und Implementierungen derart konzeptioniert sind, dass eine Übertragbarkeit auf andere agentenbasierte Simulationsumgebungen denkbar ist.

4.7 Zusammenfassung

In diesem Kapitel erfolgte die Begründung zur Migration von bereits existierenden Simulationsmodellen in Produktlinien. Danach wurde der grundlegende Ansatz der Migration mittels der komponentenbasierten Synthese skizziert. Anschließend wurde eine Integration des Migrationsansatzes in bestehende Vorgehensmodelle zur Durchführung von Simulationsstudien erarbeitet. Darauf folgte die Eruierung der Anforderungen an die Simulationsbausteine und Komponenten. Hierbei repräsentieren die Simulationsbausteine die Bestandteile eines Simulationsmodells, während die Komponenten die Bestandteile oder Kompositionen dieser produzieren. Basierend auf diesen Anforderungen erfolgte die Konzeption der Umsetzung dieser Simulationsbausteine und Komponenten. Im Anschluss daran wurde die Softwarearchitektur zur Realisierung des Migrationsansatzes vorgestellt. Hierbei wurden unterschiedliche Architekturen präsentiert, die jeweils die externe als auch interne Simulationsmodellgenerierung adressieren. Das Kapitel wurde mit einer Begründung für die Verwendung der Simulationsumgebung AnyLogic 8 abgeschlossen.

Kapitel 5

Filterung von Lösungsvarianten mittels SMT-Techniken

In diesem Kapitel wird ein Simulationsmodell einer blechverarbeitenden Fabrik in eine Produktlinie zur Synthese von Simulationsmodellvarianten migriert. Das Fabriksystem umfasst mehrere Maschinen, die unterschiedlich konfiguriert werden können und sich beispielsweise hinsichtlich der Bearbeitungsgeschwindigkeit oder der kompatiblen Maße des Rohmaterials unterscheiden. In diesem Anwendungsfall bilden die Konfigurationen die Variabilitätspunkte. Dabei wird insbesondere untersucht, wie die kombinatorische Logiksynthese und Techniken des Constraint-Solvings kombiniert werden können, um die synthetisierten Lösungsvarianten unter der Berücksichtigung von numerischen Randbedingungen (z. B. Kosten und Durchlaufzeit) zu filtern. Hierzu wird ermittelt, wie diese numerischen Randbedingungen in die Komponentensammlungen integriert und die Techniken des Constraint-Solvings in den Prozess der Synthese integriert werden können. Zunächst folgt eine Beschreibung des Anwendungsfalls.

5.1 Beschreibung des Anwendungsfalls

Der Anwendungsfall entstand in einer Zusammenarbeit mit dem Unternehmen TRUMPF Werkzeugmaschinen SE & Co. KG. Aus dieser Zusammenarbeit entstand ein Konferenzbeitrag [58], in dem die Forschungsergebnisse aus diesem Kapitel bereits teilweise veröffentlicht wurden (vgl. Kapitel 1.5). Der Anwendungsfall umfasst eine Produktionsstätte mit Werkzeugmaschinen zur Verarbeitung von Blechen. Ein Simulationsmodell dieser Produktionsstätte wurde durch das Unternehmen als agentenbasiertes Simulationsmodell in der Simulationsumgebung AnyLogic 8 entwickelt. Die Modellierung des Simulationsmodells erfolgte unter dem Einsatz eines Simulationsbaukastens, der vorgefertigte Bausteine zur Modellierung unterschiedlicher Produktionsstätten umfasst. Die Abbildung 5.1 zeigt die 3D-Visualisierung

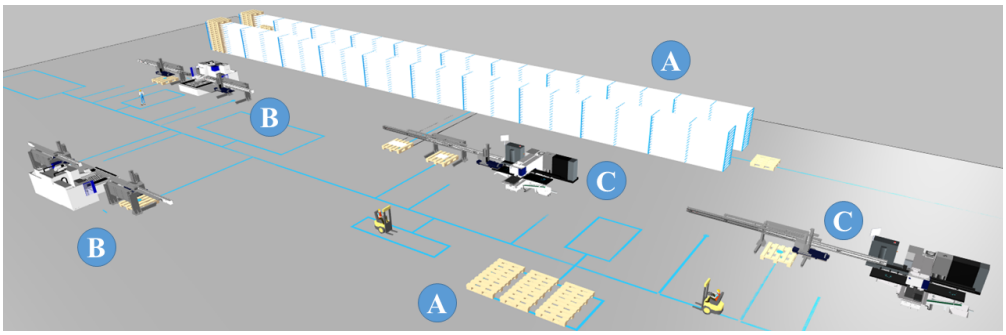


Abbildung 5.1: 3D-Visualisierung des Simulationsmodells einer blechverarbeitenden Fabrik mit einem Lager für die Bleche und leere Paletten (A), Biegemaschinen (B) sowie Schneidemaschinen (C) (Quelle: Kallat et al. [58]).

dieses Simulationsmodells. Die nachfolgende Migration verwendet das Simulationsmodell der Produktionsstätte.

Der 3D-Visualisierung ist zu entnehmen, dass die Produktionsstätte aus jeweils einem Lager für die Bleche und die leeren Paletten (A) sowie je zwei Biege- (B) und Schneidemaschinen (C) besteht. Die mittlere Schneidemaschine ist unmittelbar mit dem Lagersystem verbunden, während die rechte Schneidemaschine und die Biegemaschinen mittels Gabelstaplern mit dem Rohmaterial versorgt werden. Der Arbeitsablauf in dieser Produktionsstätte sieht zunächst das Schneiden eines Bleches in mehrere Teile und das anschließende Biegen der Teile vor, ehe diese in das Lagersystem eingelagert werden.

Nachfolgend werden die Variabilitätspunkte in diesem Anwendungsfall beschrieben. Die Schneidemaschine kann hinsichtlich des Automatisierungsgrades der Entladung konfiguriert werden, die automatisiert oder manuell durch einen Werker erfolgen kann. Überdies kann eine Schneidemaschine in ihrer Schneidegeschwindigkeit variiert werden, die wiederum niedrig, mittel oder hoch sein kann. Ebenso variiert die Biegegeschwindigkeit einer Biegemaschine, wobei diese lediglich die Ausprägungen mittel und hoch umfasst. Beide Maschinentypen können in Bezug auf die kompatible Blechgröße variiert werden, die in diesem Anwendungsfall vereinfacht in klein, mittelgroß und groß klassifiziert sind. Die kontinuierlichen Parameter sind in Intervalle gruppiert, um eine überabzählbare Menge an Konfigurationen zu vermeiden. In diesem Anwendungsfall werden keine Umrüst- oder Transportzeiten berücksichtigt, sodass sich die Durchlaufzeiten auf die reine Verarbeitung durch die Maschinen beziehen. Durch die aufgezählten Variabilitätspunkte ergeben sich für eine Schneidemaschine durch die Konfiguration der Schneidegeschwindigkeit, kompatiblen Blechgröße und Entladung insgesamt $3 * 3 * 2 = 18$ Konfigurationen. Die Biegemaschine umfasst durch die Variation der Biegeschwindigkeit und der unterstützten Blechgröße $2 * 3 = 6$ Konfigurationen. Damit umfasst die Produktionsstätte, mit jeweils zwei Schneide- und Biegemaschinen, $18 * 18 * 6 * 6 = 11664$ unterschiedliche Konfigurationen. Zum Vergleich der Konfigurationen werden nachfolgend die durchschnittliche Durchlaufzeit sowie die Kosten betrachtet. Hierbei gilt, dass eine kürzere Durchlaufzeit durch hohe Verarbeitungsgeschwindigkeiten höhere Kosten verursacht.

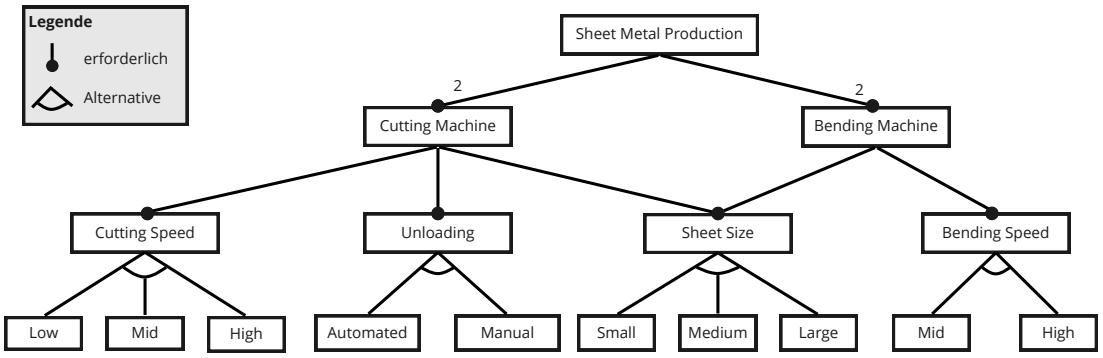


Abbildung 5.2: Feature-Diagramm, das die Variabilitätspunkte des ersten Anwendungsfalls umfasst (Quelle: Kallat et al. [58]). Ferner umfasst dieses die Kardinalität der Features Cutting Machine und Bending Machine (siehe Beschriftung der entsprechenden Kanten).

5.2 Aufbau der Komponentensammlung

In diesem Unterkapitel wird das bereits existente Simulationsmodell der Produktionsstätte in getypte Komponenten aufgeteilt. Im ersten Schritt wird zunächst ein Feature-Modell erstellt, das die zuvor beschriebenen Variabilitätspunkte systematisch erfasst. In der Abbildung 5.2 ist das Feature-Diagramm des erstellten Feature-Modells für diesen Anwendungsfall dargestellt. Das Feature-Modell dient nachfolgend als Basis für die Konzeption und Entwicklung der Komponentensammlung. In diesem Anwendungsfall korrespondieren die Features im Feature-Modell zu den Komponenten der Komponentensammlung. Dies trifft jedoch nicht auf jeden Anwendungsfall zu. Ein Kriterium lautet beispielsweise, ob die einzelnen Features sinnvoll durch jeweils eine Komponente produziert werden können. Dies ist in diesem Fall gegeben, da die Blätter im Feature-Diagramm Parametern und die Features *Cutting Machine* und *Bending Machine* Prozessbausteine darstellen, die durch entsprechende Komponenten produziert werden können.

Nachfolgend wird ein Top-Down-Ansatz vorgestellt, der ausgehend von der Wurzel sämtliche Ebenen im Feature-Diagramm durchläuft und für jede Ebene die korrespondierenden Komponenten erzeugt. Die Wurzel stellt in dem Ansatz die Top-Level-Komponente der Komponentensammlung dar, also die Komponente, welche die Produktion des ausführbaren Simulationsmodells verantwortet. Dieser Ansatz verläuft nach dem erarbeiteten Schema, das in der Tabelle 5.1 dargestellt ist. Das Schema unterscheidet zwischen zwei Fällen, die von der Verbindung eines Features mit den Sub-Features abhängt. Der erste Fall umfasst ein Feature A mit *zwingend-notwendig*-Verbindungen zu den n Sub-Features $A_1 \dots A_n$. In diesem Fall werden $n+1$ Komponenten erstellt, deren semantische Zieltypen den Bezeichnungen der Features und Sub-Features entsprechen. Dabei verantworten n Komponenten die Produktion eines Sub-Features, während eine Komponente das Feature A produziert. Die Typspezifikation letzterer Komponente umfasst eine Argumentliste mit n Argumenten, die durch den Funktionsoperator $\rightarrow$ miteinander verbunden sind. In dieser Argumentliste entspricht jedes

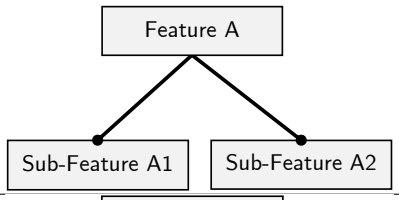
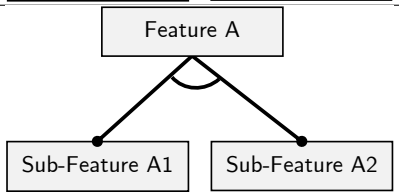
Verbindungstyp	Feature-Diagramm	Resultierende Komponentensammlung
zwingend erforderlich		$\Gamma = \{$ a : $A1 \rightarrow A2 \rightarrow A$, a1 : $A1$, a2 : $A2$ $\}$
alternativ		$\Gamma = \{$ a1 : $A \cap A1$, a2 : $A \cap A2$ $\}$

Tabelle 5.1: Vorgehen zum strukturellen Aufbau der Komponentensammlung für den ersten Anwendungsfall, basierend auf dem dazugehörigen Feature-Diagramm. Je nach Verbindungstyp eines Features zu seinen Sub-Features erfolgt die Erzeugung und die Typisierung der Komponenten.

Argument einem semantischen Zieltypen einer Komponente, die ein zugehöriges Sub-Feature produziert. Der zweite Fall umfasst ein Feature A, das mittels *entweder-oder*-Verbindungen mit n Sub-Features verbunden ist. In diesem Fall stellen die Sub-Features Ausprägungen des übergeordneten Features dar, sodass n Komponenten erstellt werden. Jede dieser Komponenten verantwortet die Produktion einer Ausprägung des Features. Die semantischen Zieltypen setzen sich aus der Intersektion der Bezeichnung des Features und des Sub-Features zusammen. Gesetzt den Fall, dass ein Sub-Feature $A1$ über eine *zwingend-notwendig*-Verbindung mit einem übergeordneten Feature A verbunden ist und das Sub-Feature $A1$ ebenso über Sub-Features verfügt, so wird das Vorgehen wiederholt. Möglicherweise ändert sich in diesem Fall die Typspezifikation der Komponente, die das Sub-Feature $A1$ produziert. In der nachfolgenden Überführung des Feature-Modells in eine Komponentensammlung tritt dieser Fall ein und wird an diesem Punkt näher erläutert.

Nachfolgend wird der Top-Down-Ansatz zur Konstruktion einer Komponentensammlung basierend auf dem Feature-Diagramm aus der Abbildung 5.2 demonstriert. Im ersten Schritt wird die Wurzel des Feature-Diagramms in eine Komponente überführt, in diesem Fall das Feature *SheetMetalProduction*. Gemäß dem Vorgehen wird die Verbindungsstruktur dieses Features zu den Sub-Features betrachtet. Da das Feature mit den Sub-Features *CuttingMachine* und *BendingMachine* über eine *zwingend-notwendig* Verbindung verbunden ist, erfolgt die Komponentenerstellung gemäß dem ersten Fall und es werden drei Komponenten erstellt:

- cuttingMachine : *CuttingMachine*
- bendingMachine : *BendingMachine*
- sheetMetalProduction : *CuttingMachine* → *CuttingMachine* → *BendingMachine* →

BendingMachine → *SheetMetalProduction*

Letztere umfasst vier Argumente, da die *zwingend-notwendig*-Verbindungen eine Multiplizität von zwei aufweisen. Daher soll die Komponente *sheetMetalProduction* jeweils zwei Instanzen beider Maschinentypen produzieren.

Im nächsten Schritt wird die Ebene, mit den Features *Schneidemaschine* und *Biegemaschine*, in Komponenten überführt. Auch wird die Verbindungsstruktur zu den jeweiligen Sub-Features betrachtet. Da das Feature *CuttingMachine* mittels einer *zwingend-notwendig*-Verbindung mit den Sub-Features *CuttingSpeed*, *Unloading* und *SheetSize* verbunden ist, werden die folgenden vier Komponenten erstellt:

- cuttingSpeed : *CuttingSpeed*
- unloading : *Unloading*
- sheetSize : *SheetSize*
- cuttingMachine : *CuttingSpeed* → *Unloading* → *SheetSize* → *CuttingMachine*

An dieser Stelle wird die Typspezifikation der Komponente cuttingMachine aus dem vorherigen Schritt überschrieben oder genauer gesagt ersetzt. Das Feature *BendingMachine* ist mit den Sub-Features *BendingSpeed* und *SheetSize* über eine *zwingend-notwendig*-Verbindung verbunden. Daher werden zur Produktion des Features drei Komponenten erstellt:

- bendingSpeed : *BendingSpeed*
- sheetSize : *SheetSize*
- bendingMachine : *BendingSpeed* → *SheetSize* → *CuttingMachine*

Da eine Komponente zur Produktion des Sub-Features *sheetSize* bereits vorhanden ist, wird das Duplikat der Komponentensammlung nicht hinzugefügt. Abschließend wird die letzte Ebene im Feature-Diagramm mittels des Vorgehens in Komponenten überführt. In dieser Ebene sind die Features über eine *entweder-oder*-Verbindung mit den Sub-Features verbunden, sodass der zweite Fall des Vorgehens eintritt. Dieser wird nachfolgend anhand des Features *CuttingSpeed* exemplarisch demonstriert. So verfügt das Feature über die Sub-Features *Low*, *Mid* und *High*, die eine Ausprägung des Features darstellen. Dementsprechend werden drei Komponenten erstellt:

- cuttingSpeedLow : *CuttingSpeed* ∩ *Low*
- cuttingSpeedMid : *CuttingSpeed* ∩ *Mid*
- cuttingSpeedHigh : *CuttingSpeed* ∩ *High*

Diese Komponenten ersetzen die zuvor erstellte Komponente cuttingSpeed. Die verbleibenden Features dieser Ebene werden analog in Komponenten überführt.

Nachfolgend werden Typvariablen und eine Substitutionskarte der Komponentensammlung hinzugefügt. Die Substitutionskarte ermöglicht das Mapping von Typvariablen auf Typen ohne Variablen. Damit kann beispielsweise der Typ $a \rightarrow a$ einer Identitätsfunktion in die Typen Int → Int oder String → String überführt werden, abhängig von der gewählten Substitution [17]. Dadurch wird eine Möglichkeit zur expliziten Forderung von Features in den synthetisierten Fabrikkonfigurationen realisiert. In diesem Anwendungsfall ist die Angabe von kompatiblen Blechgrößen gefordert. Hierzu werden zunächst die Typspezifikation der Komponenten, die eine Konfiguration einer Schneide- oder Biegemaschine produzieren, derart angepasst, dass diese eine Typvariable γ enthalten. Diese Typvariable γ kann gemäß einer Substitutionskarte durch die Typen *Small*, *Medium* und *Large* substituiert werden. Letztere repräsentieren jeweils eine mögliche Ausprägung einer Blechgröße und entsprechen einem Teil des Zieltypen der jeweiligen Komponenten, welche die Parametrisierung der Blechgröße produzieren. Damit wird die Typspezifikation der Komponente cuttingMachine wie folgt erweitert:

$$\mathit{CuttingTime} \rightarrow \mathit{Unloading} \rightarrow \mathit{SheetSize} \cap \gamma \rightarrow \mathit{CuttingMachine} \cap \gamma$$

Analog wird die Typvariable γ auch in der Typspezifikation der Komponente bendingMachine ergänzt. Der Wert der Typvariable γ wird bei der Komposition einer Maschinenkonfiguration berücksichtigt, insofern, als dieser einen Teil der Typspezifikation des Arguments bildet, das die Parametrisierung der Blechgröße produziert. Hierbei kann der Wert der Typvariable γ explizit bei der Referenzierung der Komponente *cuttingMachine* oder *bendingMachine* angegeben werden. Sofern kein Wert angegeben ist, übernimmt das Syntheseframework CLS die Ersetzung und ersetzt die Typvariable γ gemäß der Substitutionskarte. Im nächsten Schritt wird die Typspezifikation der Top-Level-Komponente *sheetMetalProduction* angepasst, da diese die zuvor angepassten Komponenten als Argumente erwartet. So werden die Typvariablen γ_1 und γ_2 ergänzt, die dieselben Werte wie γ annehmen können. Diese werden im Zieltypen sowie in der Argumentliste ergänzt, sodass sich die folgende Typspezifikation für die Top-Level-Komponente ergibt:

$$\begin{aligned}\mathit{CuttingMachine} \cap \gamma_1 \rightarrow \mathit{CuttingMachine} \cap \gamma_2 \rightarrow \mathit{BendingMachine} \cap \gamma_1 \rightarrow \\ \mathit{BendingMachine} \cap \gamma_2 \rightarrow \mathit{SheetMetalProduction} \cap \gamma_1 \cap \gamma_2\end{aligned}$$

Die Typspezifikation erfüllt die Anforderung, dass jeweils ein Paar aus Schneide- und Biegemaschine dieselbe Blechgröße verarbeitet. Dies wird dadurch gewährleistet, dass jeweils zwei Argumente zur Produktion unterschiedlicher Maschinenkonfigurationen, um eine identische Typvariable ergänzt werden. Mittels der Ergänzung der Typvariable im Zieltypen der Top-Level-Komponente kann bei der Angabe des Syntheseziels die Konfiguration der kompatiblen Blechgröße gesteuert werden. Beispielseweise wäre der Zieltyp

$$\mathit{SheetMetalProduction} \cap \mathit{Small} \cap \mathit{Medium}$$

ein valider Zieltyp, der fordert, dass in den synthetisierten Fabrikkonfigurationen ein Paar einer Schneide- und Biegemaschine kleine Bleche verarbeitet, während das andere Paar mittelgroße Bleche verarbeitet. Der Zieltyp wird aus der Konfiguration im Web-Frontend abgeleitet und automatisch konstruiert, sodass eine anwendende Person den semantischen Typ der Top-Level-Komponente nicht händisch im Quellcode anpassen muss.

Ferner wird eine Erweiterung umgesetzt, die es ermöglicht, dass Konfigurationen von Maschinen hinsichtlich der Blechgrößen abwärtskompatibel sein sollen. Damit können Maschinen, die große Bleche verarbeiten können, gleichzeitig auch mittelgroße und kleine Bleche verarbeiten. Jedoch soll diese Erweiterung optional sein. Weiterhin sei anzumerken, dass diese Anforderung keiner tatsächlichen Anforderung des realen Produktionssystems entspricht, sondern lediglich zu Demonstrationszwecken in diesem Anwendungsfall dient. Die Anforderung wird nachfolgend mittels von Subtypen im Kontext der komponentenbasierten Synthese realisiert. Hierzu wird eine Subtyp-Beziehung eingeführt, die besagt, dass der Typ *Large* ein Subtyp von *Medium* ist und dieser wiederum ein Subtyp von *Small* ist. Damit produziert die Synthese gegeben dem zuvor exemplarisch aufgeführten Syntheseziel auch eine Fabrikkonfiguration mit Maschinen, die mit großen Blechgrößen kompatibel sind. Letztere Konfiguration ermöglicht ebenso die Verarbeitung der geforderten kleinen und mittelgroßen Blechen.

Nach der Gestaltung der Komponentensammlung werden die Komponenten um die Implementierungsdetails ergänzt, die eine Produktion eines ausführbaren Simulationsmodells ermöglichen. Als Basis für die Implementierung der Komponentensammlung dient das bereits existierende Simulationsmodell. Durch die agentenbasierte Modellierung kann das Simulationsmodell mit einem überschaubarem Aufwand in Komponenten aufgeteilt werden. So werden die Schneide- und Biegemaschine durch separate Agenten repräsentiert, die als Prozessbausteine im Hauptagenten des Simulationsmodells vorhanden sind. Die Konfiguration der Maschinen erfolgt mittels der Parametrisierung der Prozessbausteine. Daher wird zur Repräsentation der Prozessbausteine und Parameter die erarbeitete Datenstruktur aus dem Kapitel 4.4 verwendet. Diese Datenstrukturen werden nachfolgend in den Implementierungsdetails der Komponenten instantiiert, sodass diese den jeweiligen Bestandteilen des Simulationsmodells entsprechen.

Zunächst wird die XML-Datei des ursprünglichen Simulationsmodells ausgelesen, die von der Simulationsumgebung AnyLogic 8 zur Persistierung der Simulationsmodelle erzeugt wird. Aus dieser können die Bestandteile eines Simulationsmodells entnommen werden. Letztere sind in einer strukturierten Weise auf unterschiedliche XML-Fragmente aufgeteilt. Die geforderten Bestandteile, genauer gesagt die korrespondierenden XML-Fragmente werden mittels der in der Benutzeroberfläche der Simulationsumgebung vergebenen Kennungen der Prozessbausteine oder Agenten ermittelt. Das Listing 5.1 zeigt einen Ausschnitt eines XML-Fragments, das eine Schneidemaschine repräsentiert. So wird die Definition des Prozessbausteins durch das XML-Tag `<EmbeddedObject>` eingeleitet. In der Zeile 3 findet sich der Name des Prozessbausteins im gleichnamigen XML-Tag. Die Zeilen 4 bis 13 beinhalten Metainformationen, während die Zeilen 14 bis 20 die Parameter des Prozessbausteins umfassen.

```
<EmbeddedObject>
  <Id>123456789123456789</Id>
  <Name>cuttingMachine</Name>
  <X>-200</X><Y>320</Y>
  <Label><X>-15</X><Y>-15</Y></Label>
  <PublicFlag>false</PublicFlag>
  <PresentationFlag>true</PresentationFlag>
  <ShowLabel>true</ShowLabel>
  <ActiveObjectClass>
    <PackageName>com.anylogic.libraries.processmodeling</PackageName>
        <ClassName>RackSystem</ClassName>
  </ActiveObjectClass>
  ...
  <Parameters>
    <Parameter>
          <Name>CuttingSpeed</Name>
          <Value Class="CodeValue"><Code>{ 1.0 }</Code></Value>
        </Parameter>
        ...
  </Parameters>
  ...
</EmbeddedObject>
```

Listing 5.1: XML-Fragment im proprietären Format der Simulationsumgebung AnyLogic 8 zur Repräsentation eines Prozessbausteins einer Schneidemaschine inklusive der Parametrisierung.

Nachfolgend wird die Umsetzung der Implementierungsdetails erläutert. Zunächst wird für jede Komponente die native Typspezifikation erarbeitet. Hierzu werden die Komponenten gemäß ihrem Produktionsergebnis in folgende Gruppen aufgeteilt:

1. Parameter,
2. Prozessbaustein und
3. Simulationsmodell mit ersetzten Prozessbausteinen.

Für jede Komponente einer Gruppe wird in den Implementierungsdetails die entsprechende Datenstruktur gemäß dem Datenmodell aus Kapitel 4.4 verwendet. Die erste Gruppe entspricht den Blättern des Feature-Diagramms aus der Abbildung 5.2, da diese die unterschiedlichen Ausprägungen der Eigenschaften einer Maschine umfassen. Diese Eigenschaften werden durch die Parametrisierung des entsprechenden Prozessbausteins bestimmt. Daher produzieren die Komponenten, die zu den Features der Blätter korrespondieren, Parameter. Gemäß dem Datenmodell ist hierfür die Datenstruktur `PlainParameter` vorgesehen, sodass

diese den nativen Zieltypen dieser Komponenten darstellt. Die zweite Gruppe entspricht den Komponenten, die jeweils einen Prozessbaustein einer Schneide- und Biegemaschine produzieren. Hierbei wird die Datenstruktur `EmbeddedObject` verwendet, die den Zieltypen der korrespondieren Komponenten bildet. Die dritte Gruppe umfasst die Komponente *sheetMetalProduction*, da diese die Top-Level-Komponente in der Komponentensammlung darstellt und für die Produktion des ausführbaren Simulationsmodells verantwortlich ist. Hierbei stellt die Datenstruktur `Agent` eine geeignete Lösung dar, da diese eine Liste von Prozessbausteinen umfasst. An dieser Stelle wird die Datenstruktur jedoch nicht verwendet, sondern es soll unmittelbar eine XML-Datei generiert werden. Daher bildet die Scala-Datenstruktur `XML.Elem` den nativen Zieltypen der Top-Level-Komponente. Die nativen Typen in den Argumentlisten der Komponenten ergeben sich gemäß der nativen Zieltypen der geforderten Komponenten. An dieser Stelle sei zu erwähnen, dass im Rahmen dieses Anwendungsfalls die verwendeten Datenstrukturen um eine Funktion `fromXML(xml: XML.Elem)` erweitert werden, welche die entsprechenden Datenstrukturen gemäß eines gegebenen XML-Fragments instantiieren. Die Tabelle 5.2 fasst die Komponenten mitsamt ihrer nativen und semantischen Typspezifikation zusammen.

Im Folgenden wird für jede Gruppe die Implementierung einer exemplarischen Komponente vorgestellt. Zunächst wird die Komponente *cuttingLowEnd* vorgestellt, welche die Parametrisierung einer niedrigen Schneidegeschwindigkeit produziert. Das Listing 5.2 zeigt die Implementierungsdetails dieser Komponente. In der Zeile 2 findet sich der semantische Zieltyp gemäß der vorgenommenen Typisierung. In den Zeilen 5 bis 10 wird der Variable `xml` das XML-Fragment zugewiesen, das aus dem ursprünglichen Simulationsmodell stammt. Im XML-Fragment wird Parametrisierung der Schneidegeschwindigkeit *CuttingSpeed* auf einen relativen Wert 0.9 gesetzt, der einer niedrigen Schneidegeschwindigkeit in der Modellierung entspricht. In der Zeile 11 folgt die Instantiierung der Klasse `PlainParameter`. Hierzu wird die Methode `fromXML` derselben Klasse aufgerufen, die ein XML-Fragment als Argument erwartet und darauf basierend eine Instanz der Klasse erzeugt.

Nun wird die Implementierung der Komponente *cuttingMachine* zur Produktion eines Prozessbausteins vorgestellt, die einen Teil der zweiten Gruppe darstellt. Die Implementierungsdetails dieser Komponente sind in Listing 5.3 dargestellt. Auch hier enthält die Zeile 2 den semantischen Typen der Komponente. Die `apply`-Methode in der Zeile 4 erhält gemäß dem semantischen Typen drei Argumente, die jeweils einem Parameter entsprechen. In der Zeile 5 wird der Variable `xml` das extrahierte XML-Fragment zur Repräsentation einer Schneidemaschine zugewiesen. In der Zeile 6 wird basierend auf dem XML-Fragment eine entsprechende Instanz der Klasse `EmbeddedObject` erzeugt. Basierend auf dieser Instanz wird in der Zeile 7 die Methode `replaceParameters` aufgerufen. Letztere stellt ebenso eine anwendungsspezifische Erweiterung der Datenstruktur dar, welche die Parameter eines Prozessbausteins ersetzt und eine Instanz des Prozessbausteins mit den ersetzten Parametern zurückgibt. Letztere bildet das Produktionsergebnis dieser Komponente.

Abschließend wird die Implementierung der Top-Level-Komponente *sheetMetalProducti-*

Komponentenname	Nativer Typ	Semantischer Typ
sheetProduction	`EmbeddedObject` →	*CuttingMachine* $\cap\gamma_1$ →
	`EmbeddedObject` →	*CuttingMachine* $\cap\gamma_2$ →
	`EmbeddedObject` →	*BendingMachine* $\cap\gamma_1$ →
	`EmbeddedObject` →	*BendingMachine* $\cap\gamma_2$ →
	`scala.xml.Elem`	*SheetProduction* $\cap\gamma_1 \cap \gamma_2$
cuttingMachine	`PlainParameter` →	*CuttingTime* →
	`PlainParameter` →	*Unloading* →
	`PlainParameter` →	*SheetSize* $\cap\gamma$ →
	`EmbeddedObject`	*CuttingMachine* $\cap\gamma$
bendingMachine	`PlainParameter` →	*BendingTime* →
	`PlainParameter` →	*SheetSize* $\cap\gamma$ →
	`EmbeddedObject`	*BendingMachine* $\cap\gamma$
cuttingLowEnd	`PlainParameter`	*CuttingTime* $\cap$ *Low*
cuttingMidEnd	`PlainParameter`	*CuttingTime* $\cap$ *Mid*
cuttingHighEnd	`PlainParameter`	*CuttingTime* $\cap$ *High*
bendingMidEnd	`PlainParameter`	*BendingTime* $\cap$ *Mid*
bendingHighEnd	`PlainParameter`	*BendingTime* $\cap$ *High*
manualUnloading	`PlainParameter`	*Unloading* $\cap$ *Manual*
automatedUnloading	`PlainParameter`	*Unloading* $\cap$ *Automated*
smallSheetSize	`PlainParameter`	*SheetSize* $\cap$ *Small*
mediumSheetSize	`PlainParameter`	*SheetSize* $\cap$*Medium*
largeSheetSize	`PlainParameter`	*SheetSize* $\cap$ *Large*

Tabelle 5.2: Komponentensammlung zur Synthese von Simulationsmodellen einer blechverarbeitenden Fabrik. Die Komponenten sind nach ihrem Produktionsergebnis gruppiert: Die erste Gruppe mit der Komponente sheetProduction produziert das ausführbare Simulationsmodell, die zweite Gruppe mit den Komponenten cuttingMachine und bendingMachine verantworten die Produktion der entsprechenden Prozessbausteine und die verbleibenden Komponenten produzieren die Parameter der Prozessbausteine.

on erläutert, deren Implementierungsdetails in Listing 5.4 dargestellt sind. In der Zeile 2 der Implementierung ist der semantische Typ mit insgesamt vier Argumenten der Variable `semanticType` zugewiesen. Demgemäß umfasst die Argumentliste der `apply`-Methode in der Zeile 4 ebenso vier Argumente, die jeweils einem Prozessbaustein des Datentyps `EmbeddedObject` entsprechen. Letztere werden in den Zeilen 6 bis 9 hinsichtlich ausgewählter Metainformationen aktualisiert. Hierzu wird eine Methode `replaceMetaInformations` aufgerufen, die gegebene Metainformationen wie die interne Kennung, den Prozessbausteinnamen und die Positionierung für einen Prozessbaustein ersetzt. Die Methode stellt ebenso eine anwendungsspezifische Erweiterung dar. Die Notwendigkeit der Aktualisierung der Metainformationen ist darauf zurückzuführen, dass die Komponentensammlung jeweils genau eine Komponente zur Produktion der Maschinen-Prozessbausteine umfasst. Dadurch enthalten die synthetisierte Prozessbausteine eines Maschinentyps die identischen Metainformationen, da diese

```
@combinator object cuttingLowEndComponent {
  val semanticType: Type = CuttingTime :&: Low

  def apply: PlainParameter = {
    val xml = <Parameter>
                <Name><![CDATA[CuttingSpeed]]></Name>
                <Value Class="CodeValue">
                  <Code><![CDATA[0.9]]></Code>
                </Value>
              </Parameter>
    PlainParameter.fromXML(xml)
  }
}
```

Listing 5.2: Implementierung einer Komponente zur Produktion der Parametrisierung der Schneidegeschwindigkeit.

in den Implementierungsdetails der jeweiligen Komponenten festgelegt sind. Eine mögliche Lösung stellt die Ergänzung weiterer Komponenten dar, welche die Prozessbausteine mit den individuellen Metainformationen produzieren. Konkret würde dies in zwei Komponenten zur Produktion der Prozessbausteine zur Repräsentation der Schneidemaschinen resultieren. Jedoch wird in dieser Umsetzung darauf verzichtet, da die Aktualisierung der Metainformationen mit geringerem Aufwand in der Top-Level-Komponente durchführbar ist. Die aktualisierten Prozessbausteine werden anschließend in der Zeile 11 in eine gemeinsame Liste `updatedEmbeddedObjects` zusammengefasst. Diese Liste wird in der Zeile 12 iteriert und für jeden Prozessbaustein wird die Methode `embeddedObjectToAnyLogic` der Klasse `AnyLogicProjectFileWriter` aufgerufen, die ein Datenobjekt in die XML-Repräsentation übersetzt. In der Zeile 13 wird das ursprüngliche Simulationsmodell als XML-Datei geladen und der Variable `xmlFile` zugewiesen. Aus dieser XML-Datei wird in der Zeile 14 der Hauptagent aus dem Simulationsmodell extrahiert und der Variable `mainAgent` zugewiesen. In der Zeile 15 werden zunächst mittels der Hilfsfunktion `findEmbeddedObjects` sämtliche Prozessbausteine des ursprünglichen Simulationsmodells extrahiert, wobei die Prozessbausteine herausgefiltert werden, die einem synthetisierten Prozessbaustein entsprechen. Das Filtern erfolgt mittels der Funktion `filter` und einem Vergleich anhand der Prozessbausteinnamen. Abschließend werden in der Zeile 17 die synthetisierten und die bereits zuvor existenten Prozessbausteine in das ursprüngliche Simulationsmodell eingesetzt. Hierzu wird die Methode `replaceNodeSeqInXML` verwendet, die in einer XML-Datei XML-Fragmente anhand ihres XML-Tags ermittelt und ersetzt. In diesem Fall werden XML-Fragmente mit dem XML-Tag `<EmbeddedObject>` ermittelt und ersetzt. Die Methode gibt nach der Ersetzung ein XML-Fragment zurück, das der ursprünglichen XML-Datei mit den ersetzten Fragmenten entspricht.

Damit sind alle Implementierungsdetails der Komponenten zur Synthese der Simulationsmodelle in diesem Anwendungsfall beschrieben. Nachfolgend wird der Einsatz des SMT-

```
@combinator object cuttingMachineComponent {
  val semanticType: Type = CuttingTime =>: Unloading =>: SheetSize :&:
   ↪ gamma =>: CuttingMachine :&: gamma

  def apply(cuttingTime: PlainParameter, unloading: PlainParameter,
   ↪ sheetSize: PlainParameter): EmbeddedObject = {
      val xml: XML.Elem = <EmbeddedObject> ... </EmbeddedObject>
      val cuttingMachine: EmbeddedObject = EmbeddedObject.fromXML(xml)
      cuttingMachine.replaceParameters(cuttingTime, unloading, sheetSize)
    }
}
```

Listing 5.3: Implementierung einer Komponente zur Produktion eines Prozessbausteins, der eine konfigurierte Schneidemaschine repräsentiert.

Constraint-Solvings untersucht zur weiteren Eingrenzung der Lösungsmenge.

5.3 Kombination von SMT-Techniken und CLS zum Filtern von Lösungen

Die erstellte Komponentensammlung ermöglicht die komponentenbasierte Synthese von 11664 Varianten des ursprünglichen Simulationsmodells. Jedoch stellt die Simulation und Evaluation dieser hohen Anzahl an Lösungsvarianten eine zeit- und kostenintensive Aufgabe dar. Daher wird nachfolgend ein Ansatz vorgestellt, der die Anzahl der Lösungsvarianten basierend auf numerischen Randbedingungen reduziert. Das Ziel ist eine Filterung der weniger vielversprechenden Lösungen. In diesem Anwendungsfall ist dieses Vorhaben insbesondere möglich, da die Variabilitätspunkte mit Kosten und einer benötigten Verarbeitungsdauer verknüpft werden können. Die Umsetzung erfolgt mittels des Einsatzes des SMT-Constraint-Solving, das im nachfolgenden Unterkapitel eingeführt wird.

5.3.1 Einführung in das SMT-Constraint-Solving

In der Informatik können Probleme häufig auf Formeln in einer bestimmten Logik und deren Erfüllbarkeit reduziert werden. So kann die Formulierung in der Aussagenlogik und die Überprüfung der Erfüllbarkeit durch einen SAT-Solver erfolgen. Überdies existieren Problemstellungen, in denen die Prädikatenlogik eine kompaktere und natürlichere Formulierung ermöglicht. Ferner ist die Prädikatenlogik durch das Vorhandensein von nicht-booleschen Variablen, Funktionen, Prädikaten und Quantoren in ihrer Ausdrucksstärke stärker als die Aussagenlogik [9]. Weiterhin wird die Ausdrucksstärke durch die Verwendung von Hintergrundtheorien verstärkt. Letztere erlauben etwa den Einsatz von Arrays oder Bit-Vektoren.

```
1  @combinator object sheetProductionComponent {
2    val semanticType: Type = CuttingMachine =>: CuttingMachine =>:
     ↪  BendingMachine =>: BendingMachine =>: SheetProduction
3
4    def apply(cuttingMachine1: EmbeddedObject, cuttingMachine2:
     ↪  EmbeddedObject, bendingMachine1: EmbeddedObject, bendingMachine2:
     ↪  EmbeddedObject): XML.Elem = {
5
6      val updatedCuttingMachine1: EmbeddedObject =
       ↪  cuttingMachine1.replaceMetaInformation("15...20",
       ↪  "cuttingMachine1", (470, 1350))
7      val updatedCuttingMachine2: EmbeddedObject =
       ↪  cuttingMachine2.replaceMetaInformation("15...87",
       ↪  "cuttingMachine2", (470, 1960))
8      val updatedBendingMachine1: EmbeddedObject =
       ↪  bendingMachine1.replaceMetaInformation("15...86",
       ↪  "bendingMachine1", (480, 2660))
9      val updatedBendingMachine2: EmbeddedObject =
       ↪  bendingMachine2.replaceMetaInformation("15...40",
       ↪  "bendingMachine2", (480, 3120))
10
11     val updatedEmbeddedObjects: Seq[EmbeddedObject] =
       ↪  Seq(updatedCuttingMachine1, updatedCuttingMachine2,
       ↪  updatedBendingMachine1, updatedBendingMachine2)
12     val nodes: Seq[Node] =
       ↪  embeddedObjects.map(AnyLogicProjectFileWriter.embeddedObjectToAnyLogic)
13     val xmlFile: Elem = XML.loadFile("[path to simulation model file]")
14     val mainAgent: Node = findMainAgent(xmlFile)
15     val originalEmbeddedObjects: Seq[EmbeddedObject] =
       ↪  findEmbeddedObjects(mainAgent).filter(currEmbeddedObject =>
       ↪  updatedEmbeddedObjects.exists(_.name == currEmbeddedObject.name))
16
17     AnyLogicProjectFileWriter.replaceNodeSeqInXML(xmlFile,
       ↪  Map("EmbeddedObjects" -> originalEmbeddedObjects ++
       ↪  updatedEmbeddedObjects))
18   }
19 }
```

Listing 5.4: Implementierung der Top-Level-Komponente zur Produktion des ausführbaren Simulationsmodells in Form eines XML-Fragments. Hierzu erwartet die Komponente Argumente, die parametrisierte Maschinen-Prozessbausteine repräsentieren. Diese werden in den Zeilen 6 bis 9 hinsichtlich ihrer Metainformationen aktualisiert. In den Zeilen 11 bis 15 werden die synthetisierten und aktualisierten Prozessbausteine in einer Liste zusammengefasst sowie die Prozessbausteine aus dem ursprünglichen Simulationsmodell extrahiert, wobei die Prozessbausteine, die synthetisiert wurden, herausgefiltert werden. In der Zeile 17 werden die Prozessbausteine in das ursprüngliche Simulationsmodell eingefügt.

Die prädikatenlogischen Formeln können mittels Satisfiability Modulo Theories (SMT)-Solver auf ihre Erfüllbarkeit modulo der verwendeten Hintergrundtheorien überprüft werden. Bei einer erfüllbaren Formelmenge generieren die SMT-Solver zudem entsprechende Modelle mit erfüllenden Variablenbelegungen. Das Einsatzgebiet der SMT-Solver umfasst unter anderem die statische Programmanalyse, Programmverifikation, das Model-Checking oder auch die Dynamic Symbolic Execution [28]. Beispielhafte Vertreter der SMT-Solver sind Alt-Ergo, Beaver, Boolector, CVC4, MathSAT5, openSMT, SMTInterpol, Sonolar, STP, veriT, Yices und Z3 [118]. Eine wesentliche Rolle bei der Weiterentwicklung der SMT-Solver lässt sich auf die Austragung des jährlichen SMT-Solver-Wettbewerb *SMT-COMP* zurückführen, der von der internationalen Initiative SMT-LIB organisiert wird. Für die Austragung des Wettbewerbs wurde das Format SMT-LIB konzeptioniert, das ein Ein- und Ausgabeformat für SMT-Solver darstellt und für Benchmarks verwendet werden kann. Jedoch hat sich das Format als ein Standardformat im Bereich der SMT-Solver etabliert. [124] Auch in der Umsetzung dieses Anwendungsfalls wird das SMT-LIB Format als Ein- und Ausgabeformat für die Kommunikation mit dem SMT-Solver eingesetzt. Daher wird das Format nachfolgend eingeführt.

Der Standard SMT-LIB stellt eine Sprache zur Verfügung, das die Formulierung von Entscheidungsproblemen erlaubt. Die Syntax dieser Sprache orientiert sich an der Sprache Common Lisp. Bei der Verwendung eines SMT-Solvers und der Sprache sind die folgenden Elemente festzulegen: die zugrundeliegende Theorie, die Hintergrundtheorie(n) und das Skript mit den Eingabe-Formeln.

Das erste Element, die zugrundeliegende Theorie, bezieht sich auf das verwendete logische System (z. B. Prädikaten-, Modal- oder Temporallogik). Das zweite Element, die Hintergrundtheorie, wird gemäß dem Standard SMT-LIB zwischen *grundlegend* und *kombiniert* unterschieden. Eine grundlegende Hintergrundtheorie ist ein Teil des Theorienkatalogs des Standards SMT-LIB. Letztere stellt eine Liste mit Hintergrundtheorien dar, die jeder SMT-LIB-kompatible SMT-Solver unterstützten sollte. Beispielhafte Theorien dieses Katalogs stellen die Theorie der Arithmetik, Arrays, Bit-Vektoren oder die Kern-Theorie dar. Letztere umfasst die Konnektoren zur Formulierung von aussagenlogischen Formeln sowie die Theorien der Gleichheit und der uninterpretierten Funktionen. Die zuletzt Genannte ermöglicht die Deklaration von Funktionen, deren Interpretation zunächst nicht festgelegt wird. Bei diesen Funktionen erfolgt die Definition der Interpretation durch das Aufstellen von Randbedingungen, welche die Ein- und Ausgabe der Funktion spezifizieren. Die kombinierten Hintergrundtheorien stellen eine Komposition von mehreren Hintergrundtheorien dar. Das dritte Element, das Skript, stellt die Eingabe des SMT-Solvers dar und liegt in der Syntax der SMT-LIB Sprache vor. Dieses Skript enthält die Festlegung auf die zugrundeliegende Theorie und die verwendeten Hintergrundtheorien. Ferner enthält das Skript die Konfiguration des SMT-Solvers. Beispielsweise kann das Zeitlimit einer Erfüllbarkeitsprüfung limitiert werden. Der Kern dieses Skript stellen jedoch die Formeln dar, welche die Randbedingungen zur Formulierung des Entscheidungsproblems umfassen. Nachfolgend werden die Elemente der Sprache vorgestellt, die für die Formulierung eines Skripts zum Filtern von synthetisierten Lösungsvarianten relevant sind. Eine vollständige Spezifikation der Sprache findet sich in der dazugehörigen Dokumentation

des Standards in [8].

In der Sprache SMT-LIB kann mittels des Schlüsselworts `declare-const` eine Variable deklariert werden. Hierbei muss die Bezeichnung und die Sorte einer Variable angegeben werden. Die Sorte bezeichnet den Typen einer Variable oder die Rückgabe einer Funktion. Die Sprache SMT-LIB umfasst unter anderem die Sorten `Bool` für boolesche Werte, `Int` für reelle Zahlen oder `Array` für Felder. Eine Funktion wird über das Schlüsselwort `declare-fun` deklariert. Dabei folgt nach der Angabe einer Funktionsbezeichnung, die Auflistung der Sorten der Argumente sowie der Rückgabe der Funktion. In Listing 5.5 erfolgt in der Zeile 1 die Deklaration einer Variable c der Sorte σ und in der Zeile 2 wird eine Funktion f mit n Argumenten, die eine Ausgabe der Sorte σ erzeugt, deklariert.

```
1 (declare-const c σ)
2 (declare-fun f (σ_1,...,σ_n) σ)
```

Listing 5.5: Deklaration einer Konstante und einer Funktion in SMT-LIB.

Die deklarierten Variablen und Funktionen verfügen in dieser Form über keine Interpretation. Die Erweiterung um eine Interpretation erfolgt durch das Aufstellen von Randbedingungen. Diese werden in der Sprache SMT-LIB durch das Kommando `(assert φ)` eingeführt mit φ als eine beliebige logische Formel. Die Konstruktion einer logischen Formel kann mit den aussagenlogischen Konnektoren der Kern-Theorie erfolgen, die in der Tabelle 5.3 aufgelistet sind. Weiterhin können Funktionen der ausgewählten Hintergrundtheorien verwendet werden, wie die Funktion + aus der Theorie der Arithmetik. Diese erlaubt in einem Ausdruck `(+ a b)` die Addition zweier Zahlen a und b. Ferner ist dem beispielhaften Ausdruck zu entnehmen, dass die Sprache eine Präfix-Notation vorsieht, bei der zuerst der Operator und dann die Funktionsargumente folgen.

Boolescher Konnektor	Interpretation	Verwendung in SMT-LIB
$\neg A$	nicht Aussage A	`(not a)`
$A \implies B$	aus Aussage A folgt Aussage B	`(=> a b)`
$A \wedge B$	Aussage A und Aussage B	`(and a b)`
$A \vee B$	Aussage A oder Aussage B (oder beide)	`(or a b)`
$A \oplus B$	entweder Aussage A oder Aussage B	`(xor a b)`

Tabelle 5.3: Konnektoren der Kern-Theorie von SMT-LIB zur Formulierung aussagenlogischer Formeln.

In einer Formel φ einer Randbedingung können Variablen verwendet werden, die durch die Verwendung eines Existenz- oder Allquantors gebunden werden. Ein Existenzquantor wird durch das Schlüsselwort `exists` und der Allquantor durch `forall` eingeleitet. Nach dem Schlüsselwort folgt eine Auflistung der Variablen in Form von Tupeln mit x_i als Variablenname und σ_i als Sorte der Variable, wobei $1 \leq i \leq n$ mit n als Anzahl der Variablen. Nach der Auflistung der Variablen folgt eine Formel Ψ, welche die Variablen verwendet. Das Listing 5.6

demonstriert die Verwendung der Quantoren. In der Zeile 1 werden Variablen mittels einer Allquantors gebunden, während in der Zeile 2 der Existenzquantor verwendet wird. Ferner ist in der Zeile 2 eine Kurzform der Verwendung eines Quantors dargestellt, bei der die gebundenen Variablen in einer gemeinsamen Liste zusammengefasst sind. Hierbei wird also nicht für jede Variable das entsprechende Schlüsselwort des Quantors aufgeführt (vgl. Zeile 1).

```
1 (assert (forall ((x_1 σ_1)) (forall ((x_2 σ_2)) ... (forall ((x_n σ_n)) Ψ))))
2 (assert (exists ((x_1 σ_1) (x_2 σ_2) ... (x_n σ_n)) Ψ))
```

Listing 5.6: Verwendung des All- und Existenzquantors in vollständiger und abgekürzter Form in SMT-LIB.

Die Quantoren können insbesondere zur Definition der Interpretation einer Funktion verwendet werden, wie in Listing 5.7 demonstriert. In der Zeile 1 erfolgt die Deklaration einer Funktion f mit n Argumenten. In der darauffolgenden Zeile wird eine Randbedingung hinzugefügt, die n Variablen mittels eines Allquantors bindet. Die Variablen entsprechen den Argumenten der Funktion f hinsichtlich der Sorten. Die gebundenen Variablen werden in derselben Zeile verwendet, um das Ergebnis der Funktion f zu definieren. So soll mit den gegebenen Variablen als Argumente das Ergebnis der Funktion einer Formel φ entsprechen. Die Sprache SMT-LIB erlaubt die Deklaration einer Funktion und das Hinzufügen der entsprechenden Randbedingungen in einem einzelnen Kommando. Hierfür ist das Kommando `(define-fun` f `((`x_1 σ_1`),...,(`x_n σ_n`))` σ Ψ`)` vorgesehen. In dieser abgekürzten Schreibweise folgen dem Funktionsnamen f die Funktionsargumente, die Sorte σ der Funktionsrückgabe sowie die Formel Ψ, welche die Randbedingung umfasst.

```
1 (declare-fun f (σ_1,...,σ_n) σ)
2 (assert (forall ((x_1 σ_1) (x_2 σ_2) ... (x_n σ_n)) (= (f x_1 ... x_n) φ)))
```

Listing 5.7: Deklaration einer Funktion (vgl. Zeile 1) und Definition der Interpretation mittels Randbedingungen (vgl. Zeile 2) in SMT-LIB.

Nach der Vervollständigung eines Skripts kann dieses mit einem SMT-Solver auf die Erfüllbarkeit überprüft werden. Letztere wird durch das Kommando `(check-sat)` im Skript angestoßen. Für die Überprüfung der Erfüllbarkeit erfolgt eine Konjunktion der Randbedingungen durch den SMT-Solver. Sofern diese Konjunktion erfüllbar ist, antwortet der Solver mit `sat`, während bei einer unerfüllbaren Formelmenge `unsat` zurückgegeben wird. Eine weitere Antwort lautet `unknown`, die genau dann zurückgegeben wird, wenn nach Ablauf eines Zeitlimits keine Antwort ermittelt werden konnte. Sofern eine Formelmenge erfüllbar ist, kann mit dem Kommando `(get-model)` ein Modell mit einer erfüllenden Variablenbelegung angefordert werden. Hierbei sei anzumerken, dass ein SMT-Solver häufig ein einzelnes Modell zurückgeben, auch bei wiederholten Durchläufen.

Zur Demonstration der Verwendung der Sprache SMT-LIB werden nachfolgend eine aussa-

genlogische Formel sowie eine prädikatenlogische Formel als Skripte formuliert und auf ihre Erfüllbarkeit überprüft. Zunächst wird die Formel 5.1 betrachtet.

$$(a \vee b) \wedge \neg a \tag{5.1}$$

Das korrespondierende Skript ist in Listing 5.8 dargestellt. Zunächst erfolgt in den ersten zwei Zeilen die Deklaration der aussagenlogischen Variablen a und b der Sorte `Bool`. In der Zeile 3 wird die Randbedingung ergänzt, die unter Verwendung der aussagenlogischen Konnektoren aus der Tabelle 5.3 die Formel 5.1 konstruiert wurde. Abschließend folgt in den Zeilen 4 und 5 die Überprüfung der Erfüllbarkeit sowie die Anfrage nach einem Modell.

```
1 (declare-fun a () Bool)
2 (declare-fun b () Bool)
3 (assert (and (or a b) (not a)))
4 (check-sat)
5 (get-model)
```

Listing 5.8: Konstruktion einer aussagenlogischen Formel in SMT-LIB.

Das Listing 5.9 zeigt die Antwort des SMT-Solvers für das Beispiel. Zunächst ist der Zeile 1 zu entnehmen, dass die Formel erfüllbar ist. Die darauffolgenden Zeilen zeigen eine Variablenbelegung, mit der die Formelmenge erfüllt werden kann. Letztere wird in Form von Funktionsdefinitionen ausgegeben. Aus diesen kann abgeleitet werden, dass die Zuweisungen der Variable a mit dem Wahrheitswert *falsch* und b mit dem Wahrheitswert *wahr* zu einer Erfüllung der Formel führen.

```
1 sat
2 (model
3   (define-fun a () Bool false)
4   (define-fun b () Bool true)
5 )
```

Listing 5.9: Antwort des SMT-Solvers nach einer Erfüllbarkeitsüberprüfung einer aussagenlogischen Formel

Im zweiten Beispiel wird eine prädikatenlogische Formel betrachtet, die in der Formel 5.2 definiert ist.

$$0 < x \leq 10 \tag{5.2}$$

Das korrespondierendes SMT-LIB-Skript ist in Listing 5.10 dargestellt. In der Zeile 1 wird die Variable x als Variable der Sorte `Int` deklariert. In der Zeile 2 erfolgt die Konstruktion der Formel unter Verwendung der arithmetischen Operatoren der Theorie der Arithmetik. Analog zum vorhergehenden Beispiel wird die Überprüfung der Erfüllbarkeit und die Anforderung

eines Modells durch die Kommandos in den darauffolgenden Zeilen angestoßen.

```
1 (declare-const x Int)
2 (assert (and (< 0 x) (<= x 10)))
3 (check-sat)
4 (get-model)
```

Listing 5.10: Konstruktion einer prädikatenlogischen Formel in SMT-LIB.

Das Listing 5.11 zeigt die Antwort des SMT-Solvers. Auch diese Formelmenge ist erfüllbar, und zwar mit der Belegung $x = 1$. Wie bereits erwähnt, führt die wiederholte Ausführung des Skripts zur Rückgabe eines identischen Modells. Sofern weitere Belegungen berechnet werden sollen, können die bisher berechneten Modelle in einer negierter Form der Formelmenge hinzugefügt werden. In diesem Beispiel führt dies zur Formel 5.3.

$$0 < x \leq 10 \wedge x \neq 1 \tag{5.3}$$

Dies wiederum resultiert in einer zusätzlichen Randbedingung `(assert (not (= x 1)))`, die vor der Erfüllbarkeitsprüfung in der Zeile 4 in Listing 5.8 ergänzt werden sollte. Damit berechnet der SMT-Solver ein weiteres Modell.

```
1 sat
2 (model
3   (define-fun x () Int 1))
```

Listing 5.11: Antwort des SMT-Solvers nach einer Erfüllbarkeitsüberprüfung einer prädikatenlogischen Formel.

In dieser Dissertation wird der SMT-Solver Z3 von Microsoft Research verwendet. Der Solver wurde im Jahr 2007 veröffentlicht und belegte bei der ersten Teilnahme am Wettbewerb SMT-COMP viermal den ersten Platz sowie siebenmal den zweiten Platz [82]. Auch in jüngeren Austragungen des Wettbewerbs erreicht der SMT-Solver Z3 Höchstplatzierungen [124]. Die Kommunikation mit Z3 kann sowohl mittels Skripten im SMT-LIB Format als auch über Programmierschnittstellen in den Programmiersprachen Java, Python oder C++ erfolgen. Die Architektur des SMT-Solvers setzt sich zusammen aus einem SAT-Solver, einem Kern-Theorie-Solver, mehreren Satelliten-Solvern (für Hintergrundtheorien wie die Arithmetik oder Arrays) sowie einer *E-Matching abstract machine*, die die Quantifizierung von Variablen ermöglicht. [82] Aufgrund der hervorragenden Performanz in den SMT-COMP Wettbewerben wird dieser in dieser Dissertation verwendet. Da in dieser Dissertation die Kommunikation mittels eines SMT-LIB-Skripts erfolgt, kann der SMT-Solver mit überschaubarem Aufwand ersetzt werden.

5.3.2 Eignung hinsichtlich des Filterns von Lösungen

Nachfolgend wird diskutiert, inwiefern das SMT-Constraint-Solving in Kombination mit der kombinatorischen Logiksynthese sinnvoll zur Filterung von Lösungen eingesetzt werden kann. Die Motivation für den Einsatz der SMT-Techniken begründet sich darin, dass in der kombinatorischen Logiksynthese die Berücksichtigung von numerischen Randbedingungen schwierig zu realisieren ist [131]. Jedoch sind SMT-Solver in der Lage mit numerischen Randbedingungen umzugehen. So verwenden Winkels et al. [131] einen SMT-Solver ergänzend zur kombinatorischen Logiksynthese, um Lösungsvarianten anhand numerischer Randbedingungen zu filtern. In der Forschungsarbeit wird ebenfalls das CLS-Framework verwendet. Dabei werden Pläne zur Synchronisation der Fabrik- und Bauplanung synthetisiert. In diesem Zusammenhang beziehen sich die numerischen Randbedingungen etwa auf eine maximale Tragkraft eines Bodens.

Ebenso stellen die SMT-Techniken eine sinnvolle Erweiterung in Bezug auf die Erweiterung der Ausdrucksstärke zur Berücksichtigung von domänenspezifischen Wissen dar. So wird in der Forschungsarbeit von Kallat et al. [60] festgestellt, dass die Ausdrucksstärke des domänenspezifischen Wissens durch das unterliegende Typsystem des CLS-Frameworks limitiert ist, da die Intersektionstypen nicht explizit die logischen Konnektoren Konjunktion, Disjunktion und Negation berücksichtigen. Ferner kann im kombinatorischen Ansatz durch die Typspezifikation einer Komponente kein unmittelbarer Einfluss auf die globale Struktur eines resultierenden Terms genommen werden. Letzteres wäre etwa notwendig, um explizit zu fordern, dass sofern eine Komponente c_0 in einem Term vorhanden ist, dann auch eine Komponente c_1 vorhanden sein muss. Ferner kann nicht verhindert werden, dass Komponenten mehrfach hintereinander in einem Term vorkommen. Jedoch sei anzumerken, dass die genannten Limitierungen nicht für alle Problemstellungen relevant sind, sodass der Einsatz der SMT-Techniken anwendungsfallabhängig zu entscheiden ist. In der Forschungsarbeit in [60] werden die SMT-Techniken in Kombination mit der kombinatorischen Logiksynthese eingesetzt, um bei der Synthese von Pfadplanungen zu verhindern, dass in einem Pfad nach einem Schritt in eine bestimmte Richtung der darauffolgende Schritt in die entgegengesetzte Richtung erfolgt.

5.3.3 Konzeption und Realisierung eines Vorgehens

Nachfolgend wird das Vorgehen zur Filterung von Lösungsvarianten mittels der SMT-Techniken im Kontext der kombinatorischen Logiksynthese dargestellt. Das Vorgehen wird in Teilen bereits in den Veröffentlichungen [60, 58] präsentiert, in denen der Autor dieser Dissertation in der Autorenschaft vertreten ist. In diesem Anwendungsfall besteht das grundlegende Ziel in einer Reduzierung der Lösungsmenge durch das Herausfiltern von wenig vielversprechenden Lösungen. Hierbei wird die Güte einer Lösung, die eine Fabrikkonfiguration umfasst, anhand der Durchlaufzeit oder der Maschinenkosten bestimmt. Beim Filtern werden sämtliche Lösungen, die eine festgelegte Durchlaufzeit oder Kosten überschreiten, als nicht vielversprechend

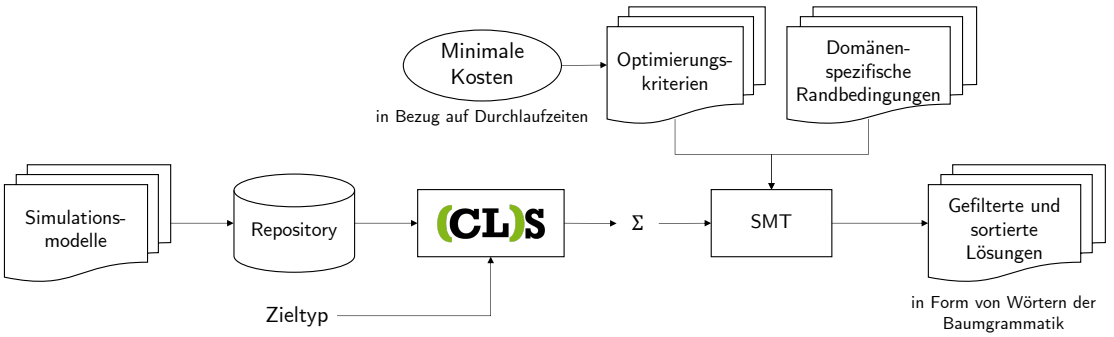


Abbildung 5.3: Integration der SMT-Techniken im Kontext der kombinatorischen Logiksynthese von Simulationsmodellen (Quelle: In Anlehnung an Kallat et al. [58]). Hierfür ist nach der Synthese durch das Framework CLS eine Übersetzung der Baumgrammatik Σ in SMT-Formeln sowie die Formulierung von domänenspezifischem Wissen und Optimierungskriterien als SMT-Formeln ergänzt. Durch den Einsatz eines SMT-Solvers können Wörter der Baumgrammatik berechnet werden, die das ergänzte domänenspezifische Wissen berücksichtigen.

markiert. Ferner erfolgt in diesem Zusammenhang eine Optimierung von Kennzahlen. Sofern etwa eine Optimierung der Durchlaufzeit erfolgt, werden die Lösungen nach der geringsten Durchlaufzeit sortiert. Die Abbildung 5.3 fasst die einzelnen Schritte zur Realisierung des beschriebenen Vorgehens zusammen.

Die ersten Schritte in diesem Vorgehen entsprechen dem Vorgehen zur Migration von Simulationsmodellen. Zunächst wird das existierende Simulationsmodell in eine Komponentensammlung überführt (vgl. Kapitel 5.2), ehe die Synthese gegeben der Komponentensammlung und dem Syntheseziel mittels des CLS-Frameworks erfolgt. Nach diesem Schritt sind die SMT-Techniken verortet. So wird das Syntheseergebnis in Form einer Baumgrammatik Σ in ein Skript mit SMT-Formeln übersetzt. Anschließend kann dieses Skripts mittels eines SMT-Solvers ausgeführt werden. Hierbei erfolgt zunächst eine Erfüllbarkeitsprüfung der Formelmenge. Sofern die Formelmenge erfüllbar ist, kann ein Modell berechnet, das einem Wort der Baumgrammatik gleichkommt. Dieses Wort wiederum entspricht einem validen Inhabitanten und somit einer Fabrikkonfiguration. Im nächsten Schritt wird das Skript um weitere SMT-Formeln ergänzt, welche die domänenspezifische Randbedingungen und Optimierungskriterien (z. B. minimale Kosten) umfassen. Eine Ausführung des ergänzten Skripts durch den SMT-Solver prüft erneut die Erfüllbarkeit. Sofern diese gegeben ist, wird ein Modell berechnet, das einem Wort der Grammatik entspricht und die ergänzten domänenspezifischen Randbedingungen und Optimierungskriterien berücksichtigt. An dieser Stelle sei anzumerken, dass nach jeder erfolgreichen Berechnung eines Modells, das Modell in negierter Form dem Skript hinzugefügt wird. Damit wird erreicht, dass in einer nachfolgenden Erfüllbarkeitsprüfung ein weiteres Modell berechnet wird, dass ungleich dem aktuellen Modell ist. Sofern es kein weiteres Modell gibt, antwortet der SMT-Solver mit `unsat` und das Vorgehen endet. Die berechneten Modelle, die den Wörtern der Grammatik Σ entsprechen, werden in einem nachfolgenden Schritt durch das CLS-Framework interpretiert. In diesem Schritt werden die

ausführbaren Simulationsmodelle produziert und der anwendenden Person zur Verfügung gestellt.

5.3.3.1 Übersetzung der Baumgrammatik in SMT-Formeln

In diesem Unterkapitel wird die Übersetzung der vom CLS-Framework synthetisierten Baumgrammatik in ein Skript mit SMT-Formeln vorgestellt. Diese Übersetzung dient als Basis zur Aufstellung von SMT-Formeln, welche die domänenspezifischen Randbedingungen und Optimierungskriterien umfassen. Die Übersetzung der Baumgrammatik erfolgt durch das Framework CLS-SMT. Nachfolgend wird die Übersetzung einer Baumgrammatik kompakt vorgestellt, da der Aufbau und die Struktur der Übersetzung im Kontext der Ergänzung weiterer SMT-Formeln relevant ist. Eine detaillierte Beschreibung des CLS-SMT-Frameworks findet sich in [60].

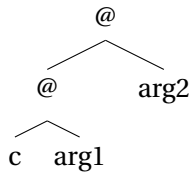
Der Kern des Frameworks CLS-SMT bildet die Datenstruktur *InhabitantTree*, die einen Binärbaum mit ganzen Zahlen als Werte der Knoten darstellt. Diese Datenstruktur repräsentiert applikative Terme und wird zur Übersetzung in korrespondierende SMT-Formeln verwendet. Die Datenstruktur ist in der Definition 11 definiert. [60]

Definition 11 (Inhabitant Tree) Sei n eine endliche Zahl, welche die Anzahl der Komponenten in der Baumgrammatik umfasst, und $C \subset \mathbb{N}$ eine nicht-leere Menge mit $\{1, \ldots, n\}$. Gegeben sei $c \in C$, dann kann ein Inhabitant Tree wie folgt definiert werden:

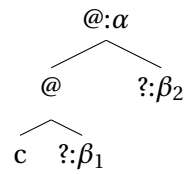
$$\textit{inhabTree} = 0 \ \textit{(leftChild inhabTree) (rightChild inhabTree)} \mid c \qquad \square$$

Gemäß der Definition 11 ergibt sich das Alphabet der Knotenbeschriftungen Σ_V durch $\{0\} \cup C$. Knoten mit der Beschriftung 0 bezeichnen Applikationsknoten und werden mit dem Symbol @ hervorgehoben. Ein Applikationsknoten stellt einen speziellen Knoten dar, der genau zwei Kinder hat. Das linke Kind des Knotens umfasst eine Funktion, während das rechte Kind das entsprechende Argument enthält. Die verbleibenden Knoten der Menge C stellen Konstanten dar. Hierdurch ergibt sich, dass eine Komponente mit n Argumenten durch einen Baum mit mindestens n Applikationsknoten repräsentiert wird. Das Symbol einer Komponente findet sich im äußersten linken Blatt des Baums, während das n-te Argument einer Komponente sich im rechten Kind des n-ten Elternknoten befindet. [60]

Nachfolgend wird demonstriert, wie ein Term *((c (arg1)) arg2)* durch einen *Inhabitant Tree* repräsentiert werden kann. Der beispielhafte Term umfasst die Anwendung der Komponente *c* auf die Argumente *arg1* und *arg2*. Die Datenstruktur *Inhabitant Tree* codiert die Knoten durch Zahlen. Daher wird nachfolgend angenommen, dass *c* durch 0, *arg1* durch 1 und *arg2* durch 2 repräsentiert werden. Gegeben der Codierung ergibt sich der *Inhabitation Tree = 0 (leftChild (0 (leftChild 1) (rightChild 2)) rightChild 3)*, der als Baum wie folgt visualisiert werden kann:



Zur Realisierung der Übersetzung einer Produktionsregel der Baumgrammatik in entsprechende SMT-Formeln stellt das Framework CLS-SMT zwei SMT-Funktionen zur Verfügung. Die erste Funktion $inhabitant : V \mapsto \Sigma_V$ bildet von der endlichen Menge der Knoten V auf die Menge der Knotenbeschriftungen Σ_V ab. Damit wird eine totale Repräsentation des Baums unter Berücksichtigung der Definition 11 ermöglicht. Die zweite Funktion ist die partielle Funktion $ty : V \mapsto N$, die die Knoten des Baums auf Nichtterminale der Menge N abbildet. Die Nichtterminale der Menge N repräsentieren Typen. Mittels beider Funktionen können eine lückenhafte Baum-Repräsentation sowie entsprechende Randbedingungen für einen Teilbaum konstruiert werden, die vom SMT-Solver vervollständigt werden. Zur Demonstration wird die Produktionsregel $\{\alpha \mapsto \{(c(\beta_1, \beta_2)\}\}$ betrachtet, die durch den folgenden unvollständigen Baum und den assoziierten Nichtterminalen repräsentiert werden kann:



Aus der Produktionsregel können die folgenden Randbedingungen gefolgert werden: Sei i die Wurzel einer applikativen Komposition einer Komponente und deren Argumente. Sofern der Knoten i über einen Typen verfügt, der durch das Nichtterminal-Symbol α repräsentiert wird, dann umfasst *(leftChild (leftChild i))* den Knoten c. Dann befindet sich das erste Argument der Komponente an der Position *(rightChild (leftChild i))* und hat den Typen β_1, während das zweite Argument an der Position *(rightChild i)* zu finden ist und den Typen β_2 hat. Die Randbedingungen für die Teilbäume von β_1 und β_2 werden analog definiert. Eine detaillierte Ausführung der Übersetzung sowie Fallbeispiele sind in der entsprechenden Veröffentlichung, in der das Framework CLS-SMT in [60] vorgestellt wird, nachzulesen.

5.3.3.2 Ergänzung domänenspezifischen Wissens mittels SMT-Formeln

In diesem Abschnitt werden die Randbedingungen vorgestellt, die das domänenspezifische Wissen sowie die Optimierungskriterien für den Anwendungsfall der blechverarbeitenden Produktionsstätte umfassen. Die Basis für die nachfolgenden Randbedingungen bildet die Übersetzung der Baumgrammatik, die das Syntheseframework CLS für die Komponentensammlung aus dem Kapitel 5.2 als Syntheseergebnis produziert, mittels des Frameworks CLS-SMT. Einige der nachfolgend vorgestellten Randbedingungen nehmen Bezug auf Knoten in der Baumrepräsentation und beeinflussen so die Selektion der Komponenten in einer Lösungsvariante. Die nachfolgenden Randbedingungen werden zunächst formal definiert,

ehe die Vorstellung der korrespondierenden SMT-Formeln folgt.

Die einzelnen Konfigurationen der Schneide- und Biegemaschinen werden jeweils durch die folgenden Vektoren dargestellt:

$$\overrightarrow{cm} = (cm_{cspeed}, cm_{size}, cm_{unloading}) \text{ und } \overrightarrow{bm} = (bm_{bspeed}, bm_{size})$$

Die möglichen Konfigurationen der Schneide- und Biegemaschinen werden durch folgende Mengen beschrieben:

$$CM = \{\overrightarrow{cm}^1, \overrightarrow{cm}^2, \ldots, \overrightarrow{cm}^{18}\} \quad \text{und} \quad BM = \{\overrightarrow{bm}^1, \overrightarrow{bm}^2, \ldots, \overrightarrow{cm}^6\}$$

An dieser Stelle sei zu wiederholen, dass insgesamt sechs unterschiedliche Konfigurationen einer Biegemaschine und 18 verschiedene Konfigurationen einer Schneidemaschine im Anwendungsfall möglich sind. Im Nachfolgenden werden die Funktionen t und c definiert, die eine Maschinenkonfiguration als Argument erwarten und die entsprechende Durchlaufzeit beziehungsweise die Kosten einer Konfiguration zurückgeben. Weiterhin wird angenommen, dass jeweils ein Paar, bestehend aus einer Schneide- und Biegemaschine, derart konfiguriert ist, dass das erste Paar x und das zweite Paar y Bleche derselben Größe verarbeitet. Gegeben der zu verarbeitenden Anzahl an Blechen je Paar x, y berechnen sich die Gesamtkosten *totalCosts* sowie die Durchlaufzeit *totalTime* einer Fabrikkonfiguration mit $0 < i, k \leq 18$ und $0 < j, l \leq 6$ wie folgt:

$$\begin{aligned} totalCosts &= x * (c(\overrightarrow{cm}^i) + c(\overrightarrow{bm}^j)) + y * (c(\overrightarrow{cm}^k) + c(\overrightarrow{bm}^l)), \\ totalTime &= x * (t(\overrightarrow{cm}^i) + t(\overrightarrow{bm}^j) + y * (t(\overrightarrow{cm}^k) + t(\overrightarrow{bm}^l)) \end{aligned}$$

Die *totalTime* umfasst somit nicht die Zeitspanne zur Verarbeitung der Bleche, sondern die kumulierte Verarbeitungszeit aller Maschinen. Daraufhin können Randbedingungen ergänzt werden, welche die Gesamtkosten oder die Durchlaufzeit einer Konfiguration einschränken oder nach einem dieser Werte optimieren. Die Randbedingung

$$totalTime \leq 400 \wedge min(totalCosts)$$

fordert beispielsweise, die Filterung von Konfigurationen mit einer höheren Durchlaufzeit als 400 Minuten und die Optimierung nach den Kosten.

Im nachfolgenden Abschnitt werden die vorgestellten Randbedingungen als SMT-Formeln in der Sprache SMT-LIB formuliert. In der Implementierung dieses Anwendungsfalls erfolgt die Formulierung von SMT-Formeln mittels der Scala-Bibliothek *Scala SMT-LIB*[1], die es erlaubt SMT-Formeln in der Programmiersprache Scala zu formulieren und als SMT-LIB-Skript zu exportieren. Dies ermöglicht eine dynamische Parametrisierung der SMT-Formeln, die etwa eine dynamische Zuweisung der Kosten und der benötigten Zeit einzelner Komponenten erlaubt. Ferner können eine geforderte minimale Durchlaufzeit oder maximale Kosten dynamisch

[1] Scala SMT-LIB - https://github.com/regb/scala-smtlib

gesetzt werden. Die Bibliothek *Scala SMT-LIB* ermöglicht sowohl das Parsing von Kommandos, die in der Sprache SMT-LIB formuliert sind, als auch die Konstruktion von Formeln mittels zur Verfügung gestellter Datenstrukturen. Letzteres wird im Rahmen der Implementierung dieses Anwendungsfalls verwendet. Jedoch wird nachfolgend die Darstellung der SMT-Formeln in der Syntax der Sprache SMT-LIB gewählt, da diese eine kompaktere Darstellung ermöglicht.

```
1 (declare-fun costs (Int) Int)
2 (declare-fun time (Int) Int)
3 (declare-const totalCosts Int)
4 (declare-const totalTime Int)
5 (declare-const x Int)
6 (declare-const y Int)
```

Listing 5.12: Deklaration der Funktionen und Konstanten zur Filterung von Fabrikkonfiguration. Die ersten vier Zeilen umfassen die Kosten und die benötigte Verarbeitungsdauer der Komponenten und der Fabrikkonfigurationen. Die letzten zwei Zeilen umfassen die Anzahl der zu verarbeitenden Bleche je Maschinenpaar.

Wie bereits erwähnt, bildet die Übersetzung der Baumgrammatik durch das Framework CLS-SMT die Ausgangsbasis für die nachfolgenden Randbedingungen. Innerhalb des Skripts werden die einzelnen Komponenten durch Ziffern codiert, die in den Randbedingungen zur Referenzierung einer Komponente verwendet werden. Die Decodierung einer Ziffer ist mittels des Frameworks CLS-SMT möglich. Das resultierende Skript setzt sich somit aus der Übersetzung und den nachfolgenden Randbedingungen zusammen. Zunächst werden die notwendigen zusätzlichen Funktionen und Variablen deklariert, die in Listing 5.12 abgebildet sind. Die SMT-Funktionen `costs` und `time` entsprechen den Funktionen c und t. Die Funktionen erwarten jedoch jeweils eine Komponente, die einen Teil einer Maschinenkonfiguration produziert, und geben die Kosten oder die benötigte Verarbeitungszeit zurück. Dabei werden standardisierte Werte angenommen, sofern für eine Komponente keine Kosten oder eine Zeitspanne angegeben wurde. Die verbleibenden Variablen in den Zeilen 3 bis 6 entsprechen den gleichnamigen, zuvor vorgestellten Variablen.

```
1 (assert (= (costs  c_id) c_costs))
2 (assert (= (time  c_id) c_time))
3 (assert (= x  e_1))
4 (assert (= y  e_2))
```

Listing 5.13: Zuweisung der Gewichtung in Form von Kosten und geschätzter Zeit einer Komponente sowie der Anzahl zu verarbeitender Bleche je Maschinenreihe.

Im nächsten Schritt erfolgt die Zuweisung von Gewichtungen der Komponenten in Form von Kosten und einer geschätzten Verarbeitungsdauer. Die Zuweisung erfolgt mittels von Randbedingungen, die den uninterpretierten Funktionen *costs* und *time* die entsprechenden Werte zuweisen. In Listing 5.13 erfolgt in Zeile 1 und 2 die Zuweisung für eine Komponente c

mit c_{costs} als Kosten und c_{time} als Verarbeitungszeit mittels einer Gleichung. Die Komponente wird durch die Ziffer c_{id} in der SMT-LIB-Repräsentation encodiert. Analog erfolgt die Zuweisung der zu verarbeitenden Bleche in diesem Anwendungsfall. Diese ist in den Zeilen 3 und 4 dargestellt, wobei e_1, e_2 der Anzahl zu verarbeitenden Bleche durch das erste und zweite Paar entspricht.

```
1  (assert
2    (= totalCosts
3      (+
4        (* x
5          (+ (costs (inhabitant1 11))
6          (+ (costs (inhabitant1 21))
7          (+ (costs (inhabitant1 35))
8          (+ (costs (inhabitant1 69))
9             (costs (inhabitant1 137))))))
10       )
11       (* y
12         (+ (costs (inhabitant1 7))
13         (+ (costs (inhabitant1 13))
14         (+ (costs (inhabitant1 19))
15         (+ (costs (inhabitant1 37))
16            (costs (inhabitant1 73)))))))))
```

Listing 5.14: Randbedingungen zur Berechnung der Gesamtkosten sowie der Durchlaufzeit einer Fabrikkonfiguration für eine gegebene Anzahl an Blechen. Letztere wird durch die Konstanten x und y repräsentiert, die im Skript zuvor definiert werden. Die Argumente der Funktion inhabitant1 entsprechen den Positionen der Komponenten in der Baumrepräsentation.

Die Berechnung der Gesamtkosten mittels einer Randbedingung ist in Listing 5.14 dargestellt. Hierbei ist der strukturierte Aufbau der unvollständigen Baumrepräsentation relevant, da diese eine gezielte Selektion der Argumente einer Komponente ermöglicht. Dadurch können die Positionen der Argumente ermittelt werden, die einen Einfluss auf die Gesamtkosten einer Konfiguration nehmen. Für die nachfolgende Formulierung der Randbedingungen werden diese Positionen berechnet, sodass absolute Werte und nicht die SMT-Funktionen `rightChild` und `leftChild` verwendet werden. Die Gesamtkosten werden in Listing 5.14 separat für die erste (Zeile 4 bis 10) sowie für die zweite Maschinenreihe (Zeile 11 bis 16) berechnet. Die Komponente, die die Konfiguration der Schneidemaschine des ersten Paars produziert, befindet sich an der Position 17. Um diese Position zu ermitteln, wurde zunächst der Typspezifikation der Top-Level-Komponente entnommen, dass die zuvor genannte Maschinenkonfigurationen das erste Argument darstellt. Gemäß der Übersetzung befindet sich das Argument somit an

folgender Position in der Baum-Repräsentation:

rightChild(leftChild(leftChild(leftChild 1)))
= rightChild(leftChild(leftChild 2))
= rightChild(leftChild 4)
= rightChild 8
= 17

Die Argumente der Komponente, welche die Konfiguration einer Schneidemaschine produziert, werden analog ermittelt. Diese sind an den Positionen 35, 69 und 137 vorzufinden und werden in den Zeilen 7 bis 9 verwendet, um mittels der SMT-Funktion `inhabitant1` die synthetisierte Komponente, genauer gesagt deren Codierung abzufragen. Daraufhin wird die Funktion `costs` aufgerufen, um die Kosten dieser Komponente abzurufen. Analog erfolgt die Summierung der Kosten der Konfiguration der Biegemaschine. Ebenso werden die Kosten für die zweite Maschinenreihe addiert. Die summierten Kosten werden anschließend in Zeile 2 mittels einer Gleichung der Variable `totalCosts` zugewiesen. Die Berechnung der Gesamtdurchlaufzeit für eine Fabrikkonfiguration erfolgt auf derselben Weise.

```
1 (assert (< totalTime (* 400 60)))
2 (minimize totalCosts)
```

Listing 5.15: Randbedingungen zur Begrenzung der Gesamtdurchlaufzeit auf 400 Minuten sowie der Minimierung der Kosten als Optimierungskriterium

Abschließend werden die Randbedingungen zur Limitierung der Kosten oder der Durchlaufzeit sowie das Optimierungskriterium hinzugefügt. Das Listing 5.15 zeigt exemplarisch die Begrenzung der Gesamtdurchlaufzeit auf 400 Minuten, in dem in der Zeile 1 gefordert wird, dass der Wert der Variable `totalTime` kleiner als $400 \times 60 = 24000$ Sekunden sein sollte. In der Zeile 2 wird mittels des Schlüsselworts *minimize* gefordert, dass die Gesamtkosten minimal gehalten werden sollen.

5.4 Durchführung von Experimenten

Nachfolgend wird die Durchführung einer Simulationsstudie mit dem beschriebenen Vorgehen vorgestellt, die das Synthetisieren von möglichst vielversprechenden Fabrikkonfigurationen in Bezug auf die Kosten und Durchlaufzeit fokussiert. Für dieses Simulationsexperiment wurde eine gesonderte Seite innerhalb des Web-Frontends der Implementierung dieser Dissertation ergänzt, die eine bedienungsfreundliche Verwendung der Synthese und Filterung ermöglicht. So können die Gewichtungen der Komponenten, die Grenzwerte der Durchlaufzeit

	Schneide-geschwindigkeit			Biege-geschwindigkeit		Entladung		Blechgröße		
	Niedrig	Mittel	Hoch	Mittel	Hoch	Auto	Manuell	Klein	Mittel	Groß
Zeit	72	60	48	120	96	30	180	0	10	20
Kosten	5	10	15	10	15	5	1	2	4	6

Tabelle 5.4: Kosten und Zeit (in Sekunden) als Gewichtung für die Variabilitätspunkte im ersten Anwendungsfall

und Gesamtkosten sowie das Optimierungskriterium gesetzt werden, ohne den Quellcode der Implementierung anzupassen. In der Abbildung 5.4 ist ein Screenshot dieser Seite dargestellt.

Auf dieser Seite können die Synthese und die Filterung gemäß der vorgenommenen Konfiguration angestoßen werden. Hierbei wird die Konfiguration dem Backend-System in Form einer JSON-Datei über eine REST-API übermittelt. Eine Vorstellung dieser JSON-Datei erfolgt im Rahmen der Beschreibung der Durchführung der Experimente. Das Backend-System führt die Synthese und Filterung durch und übermittelt anschließend die synthetisierten Lösungen, sofern vorhanden, über die REST-API an das Frontend. Dieses visualisiert die einzelnen Fabrikkonfigurationen und ermöglicht das Herunterladen der synthetisierten Simulationsmodelle. Die Abbildung 5.5 zeigt einen Screenshot der Ergebnisdarstellung.

Im Rahmen dieser beispielhaften Simulationsstudie werden zwei Experimente durchgeführt. Im ersten Experiment wird die maximale Bearbeitungszeit numerisch beschränkt und eine Minimierung der Kosten gefordert, während das zweite Experiment eine Beschränkung der Kosten und eine Minimierung der Bearbeitungszeit umfasst. Die Tabelle 5.4 zeigt die Gewichtungen der Ausprägungen der Features, die durch die einzelnen Komponenten produziert werden. Beispielsweise kann der Tabelle entnommen werden, dass eine mittlere Schneidegeschwindigkeit in einer benötigten Verarbeitungsdauer von 60 Sekunden resultiert und zehn Kostenpunkte verursacht. Die Werte aus der Tabelle sind zwar fiktiv, orientieren sich jedoch an realen Werten. Eine Anpassung der Gewichtungen an reale Werte ist in der Webanwendung möglich.

Die Gesamtkosten und -bearbeitungszeit werden gemäß der Beschreibung der entsprechenden SMT-Formeln (vgl. Kapitel 5.3) berechnet. In den nachfolgenden Experimenten wird eine Verarbeitung von 40 kleinen Blechen und 60 mittelgroßen Blechen angenommen. Daraus resultiert, gegeben den Gewichtungen aus der Tabelle 5.4, dass die Konfiguration mit der kürzesten rechnerischen Bearbeitungszeit von 310 Minuten ausschließlich hohe Schneide- und Biegegeschwindigkeiten sowie ein automatisiertes Entladen vorsieht und 4140 Kostenpunkte verursacht. Die kostengünstigste Konfiguration verursacht 2240 Kostenpunkte und umfasst die jeweils niedrigste Verarbeitungsgeschwindigkeit sowie ein manuelles Entladen der Schneidemaschine und benötigt rechnerisch 640 Minuten. An dieser Stelle sei hervorzuheben,

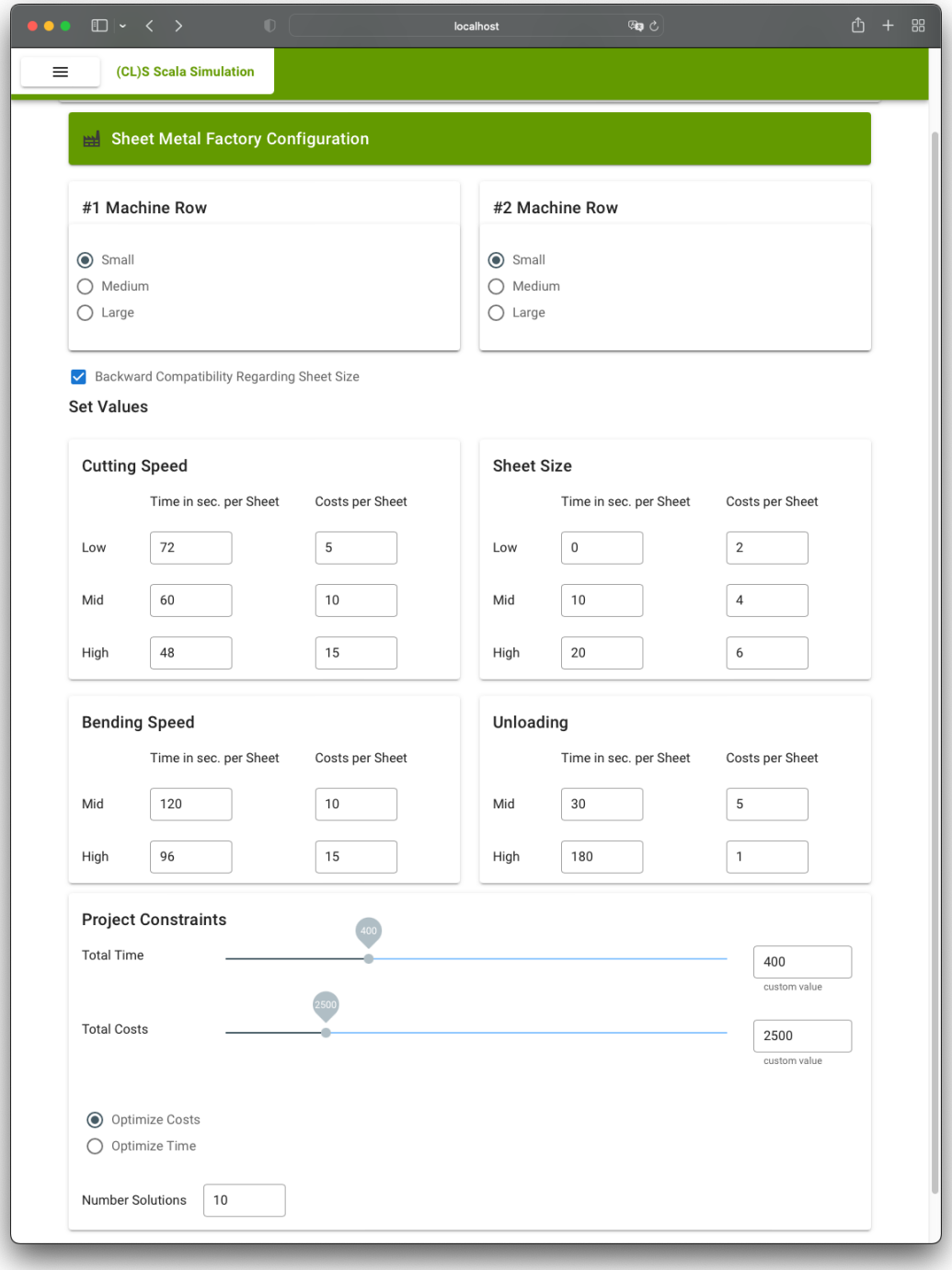


Abbildung 5.4: Screenshot des Frontends der entwickelten Technologie, das die Konfiguration der zu synthetisierenden Simulationsmodelle des ersten Anwendungsfalls zeigt. Im oberen Bereich kann die geforderte Blechgröße sowie die Abwärtskompatibilität dieser gesetzt werden. Im mittleren Bereich können die Gewichtungen der Variabilitätspunkte angepasst werden und im unteren Bereich die Randbedingungen und Auswahl des Optimierungskriteriums hinsichtlich der Kosten und Zeit gesetzt werden.

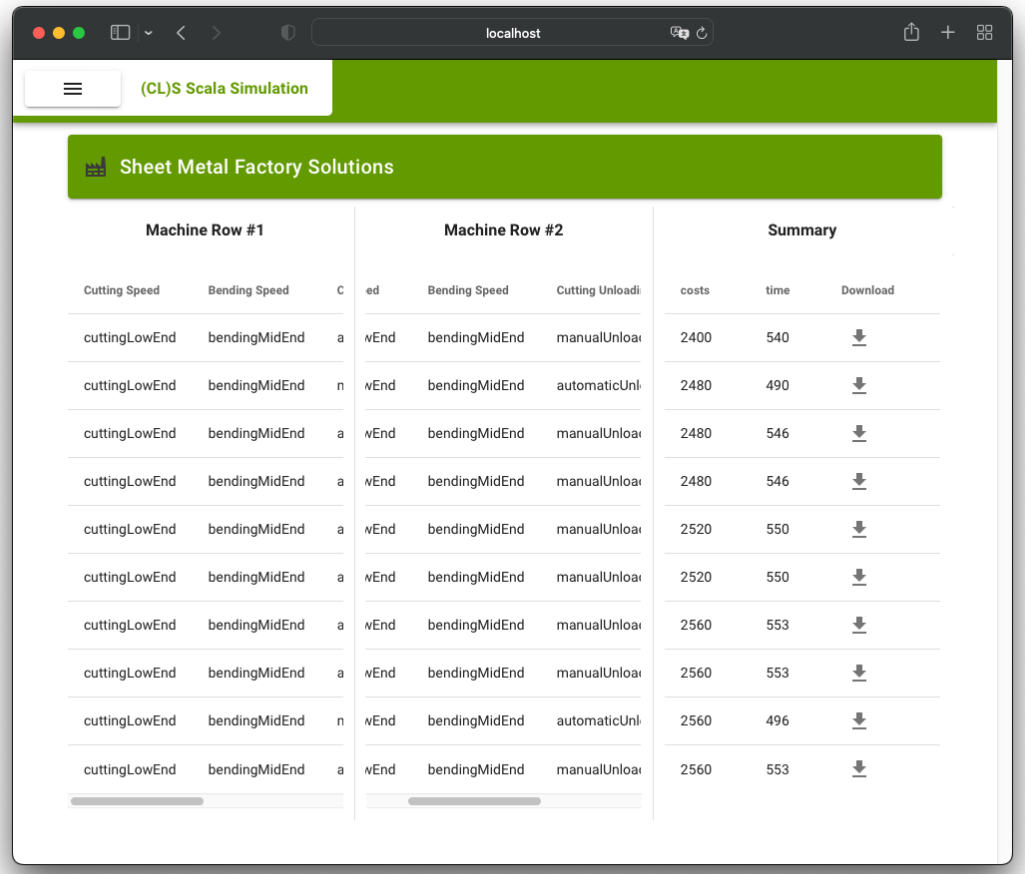


Abbildung 5.5: Screenshot des Frontends der entwickelten Technologie, das die synthetisierten Simulationsmodelle des ersten Anwendungsfalls auflistet

dass die Durchlaufzeiten, die durch die Simulation ermittelt werden, von den errechneten Werten abweichen können. Dies ist auf stochastische Effekte in der Simulation oder nicht berücksichtigte Zeiten, wie Transportzeiten, zurückzuführen. Weiterhin sei anzumerken, dass in den Experimenten hinsichtlich der Ressourcen optimale Bedingungen vorherrschen, indem beispielsweise zu jedem Zeitpunkt eine ausreichende Anzahl an Werker zur Verfügung steht. Eine Restriktion der Ressource beeinflusst möglicherweise die Bearbeitungszeit, genau dann, wenn etwa eine Maschine nicht entladen werden kann.

Die nachfolgenden Experimente werden auf einem lokalen Rechner ausgeführt, der mit einem 4-Kern Intel Core i5 8400H Prozessor und 32 Gigabyte Arbeitsspeicher ausgestattet ist. Als Betriebssystem ist Ubuntu in der Version 18.04.5 LTS installiert, auf dem der SMT-Solver Z3 in der Version 4.8.9 vorhanden ist. Die Simulationsläufe werden in AnyLogic 8 mit einem zufällig gewählten Startwert durchgeführt, sodass die Ergebnisse der Simulationsläufe aufgrund von stochastischen Effekten unterschiedlich sind. Angesichts dessen wird jede synthetisier-

te Fabrikkonfiguration in insgesamt fünf Durchläufen simuliert und der Durchschnitt der Ergebnisse gebildet.

5.4.1 Begrenzung der Durchlaufzeit und Optimierung hinsichtlich Kosten

Im ersten Experiment werden Fabrikkonfigurationen gefordert, die eine maximale Bearbeitungszeit von 400 Minuten bei gleichzeitig minimalen Kosten umfassen. Eine entsprechende Konfiguration auf der gesonderten Seite im Frontend resultiert in der JSON-Datei, die in Listing 5.16 dargestellt ist und über die REST-API an das Backend versendet wird. In der JSON-Datei werden in den Zeilen 2 bis 5 die Anzahl der zu verarbeitenden Bleche sowie die Blechgröße für die erste Maschinenreihe definiert. Analog werden diese für die zweite Maschinenreihe in den Zeilen 6 bis 9 festgelegt. Zur Demonstration einer höheren Anzahl an Varianten wird die Subtyping-Beziehung der Komponentensammlung hinzugefügt, sodass die Maschinen hinsichtlich der Blechgrößen abwärtskompatibel sind. Hierzu wird in der Zeile 10 der Wert des entsprechenden Schlüssels auf den Wahrheitswert *true* gesetzt. In den Zeilen 11 und 12 werden die numerischen Beschränkungen hinsichtlich der maximalen Bearbeitungszeit und Kosten festgelegt. Die Kosten sind in diesem Fall nicht begrenzt, daher wird der Wert 9999 angegeben, der rechnerisch durch keine Konfiguration erreicht werden kann. Die Zeile 13 umfasst das Optimierungskriterium, das entweder die Kosten (*costs*) oder die Durchlaufzeit (*time*) umfasst. Eine Optimierung beider Werte ist nicht zielführend, da die genannten Kriterien gegensätzliche Ziele verfolgen. Das Array mit dem Schlüssel *values* in der Zeile 14ff umfasst die Gewichtungen der einzelnen Komponenten. Diese entsprechen in beiden Experimenten den Werten aus der Tabelle 5.4 und sind daher in Listing 5.16 nur schematisch angedeutet.

Gegeben dem Syntheseziel $SheetMetalProduction \cap Small \cap Medium$ und der Komponentensammlung können mittels des Syntheseframeworks CLS 5184 Lösungen synthetisiert werden. Die Synthese der 5184 Lösungen benötigte mit dem CLS-Framework fünf Sekunden. Durch die Anwendung des Filterns mittels der SMT-Techniken verbleiben 325 Lösungen. Diese wurden innerhalb von 7 Minuten und 11 Sekunden gefiltert. Innerhalb dieser Zeit wurden auch die entsprechenden Simulationsmodelle durch das Syntheseframework produziert. Die fünf kostengünstigsten Fabrikkonfigurationen sind in der Tabelle 5.5 aufgelistet.

Die Zeilen der Tabelle 5.5 zeigen die synthetisierten Fabrikkonfigurationen mit den jeweiligen Kosten sowie der errechneten Bearbeitungszeit. Nachfolgend wird exemplarisch anhand der kostengünstigsten Konfiguration in diesem Experiment demonstriert, wie sich die Werte zusammensetzen. Hierzu wird die Berechnung der Gesamtdurchlaufzeit erläutert. In Entsprechung zur kostengünstigsten Konfiguration in der ersten Zeile wird ein kleines Blech zunächst von der Schneidemaschine der ersten Maschinenreihe mit einer niedrigen Geschwindigkeit von 72 Sekunden pro Blech verarbeitet und anschließend in 30 Sekunden automatisiert entladen. Im zweiten Schritt wird das kleine Blech von der ersten Biegemaschine mit einer mittleren Geschwindigkeit von 120 Sekunden pro Blech verarbeitet. Da zufolge der Gewichtungen aus

```
{
    "firstRow": {
      "amount": 40,
      "sheetSize": "small"
    },
    "secondRow": {
      "amount": 60,
      "sheetSize": "medium"
    },
    "useTaxonomy": true,
    "maximalTime": 600,
    "maximalCosts": 9999,
    "optimizationCriteria": "costs",
    "values": [
      {
        "name": "cuttingLowEnd",
        "costs": 5,
        "time": 72
      }
      ...
    ]
}
```

Listing 5.16: Konfiguration des ersten Experiments im JSON-Format. Diese sieht eine Produktion von 40 kleinen und 60 mittelgroßen Blechen, eine Begrenzung der Bearbeitungszeit auf 600 sowie die Kosten als Optimierungskriterium vor.

der Tabelle 5.4 die Bearbeitung eines kleinen Bleches die Bearbeitungsdauer nicht verzögert, ergibt sich eine rechnerische Verarbeitungsdauer von $72 + 30 + 0 + 120 + 0 = 222$ Sekunden pro Blech. Für die zu verarbeitenden 40 kleinen Bleche ergibt sich eine Bearbeitungsdauer von $40 \times 222 = 8880$ Sekunden oder 148 Minuten. Die Bearbeitungsdauer eines mittelgroßen Blechs durch die Schneide- und Biegemaschine der zweiten Maschinenreihe kann analog berechnet werden und beträgt $72 + 30 + 10 + 120 + 10 = 242$ Sekunden. Hierbei werden je Blech und Maschine zehn Sekunden aufgrund der mittelgroßen Größe des Blechs addiert. Damit ergibt sich eine Gesamtdauer für die Verarbeitung der 60 mittelgroßen Bleche von $60 \times 242 = 14520$ Sekunden oder 242 Minuten. Dies wiederum ergibt eine Gesamtbearbeitungszeit von $148 + 242 = 390$ Minuten. Die Gesamtbearbeitungszeit bezieht sich in diesem Fall nicht auf den Beendigungszeitpunkt der Produktion, sondern stellt die Summe der Bearbeitungszeiten beider Maschinenreihen dar. Die Kosten lassen sich analog berechnen.

Nachfolgend werden die vom CLS-Framework produzierten korrespondierenden Simulationsmodelle verwendet, um zu überprüfen, ob die errechnete Sortierung der Lösungen auch den tatsächlichen Ergebnissen der Simulation entsprechen. Das heißt, dass die Simulationsmodelle der fünf vielversprechendsten Fabrikkonfigurationen aus der Tabelle 5.5 ausgeführt

			#1 Schneide-maschine			#2 Schneide-maschine			#1 Biege-maschine		#2 Biege-maschine	
Kosten	Zeit (rechn.)	Zeit (sim.)	Geschw.	Blechgröße	Entladung	Geschw.	Blechgröße	Entladung	Geschw.	Blechgröße	Geschw.	Blechgröße
2400	540	572	N	K	A	N	M	M	M	K	M	M
2480	490	559	N	K	M	N	M	A	M	K	M	M
2480	546	579	N	M	A	N	M	M	M	K	M	M
2480	546	579	N	K	A	N	M	M	M	M	M	M
2520	550	581	N	K	A	N	M	M	M	K	M	G

Tabelle 5.5: Ergebnis der komponentenbasierten Synthese und des Filterns für das erste Experiment. Dieses umfasst eine Auswahl mit den fünf kostengünstigsten Fabrikkonfigurationen für die Verarbeitung von 40 kleinen und 60 mittelgroßen Blechen, wobei N, M und H für jeweils eine niedrige, mittlere und hohe Geschwindigkeit, A und M für automatisiertes und manuelles Entladen sowie K, M und G für kleine, mittelgroße sowie große Bleche stehen.

und anhand der Auswertung der Kennzahlen im Simulationsmodell überprüft wird, ob die vielversprechendste Lösung bessere Ergebnisse als die zweit-vielversprechendste Lösung liefert. Das Ergebnis dieser Simulation ist in der dritten Spalte der Tabelle 5.5 in Form der simulierten Durchlaufzeit vermerkt. Wie eingangs erwähnt, wird nicht erwartet, dass die Werte exakt übereinstimmen. Jedoch bestätigt die Simulation die berechnete Sortierung, sodass die vielversprechendste durch den SMT-Solver gefilterte Lösung auch in der Simulation der vielversprechendsten Lösung (im Vergleich zu den vier weiteren Konfigurationen) entspricht.

5.4.2 Begrenzung der Kosten und Optimierung der Durchlaufzeitzeit

Im zweiten Experiment dieses Anwendungsfalls stellen nicht die Gesamtkosten einer Fabrikkonfiguration das Optimierungskriterium dar, sondern die Bearbeitungszeit. Auch in diesem Experiment wird die Konfiguration des Experiments über eine JSON-Datei definiert, die in Listing 5.17 dargestellt ist. Jedoch wird in diesem Experiment die Subtyping-Beziehung nicht berücksichtigt, sodass ausschließlich die Fabrikkonfigurationen synthetisiert werden, die eine Parametrisierung der Maschinen mit den geforderten Blechgrößen umfassen. In diesem Experiment lautet der semantische Zieltyp somit $\mathit{SheetMetalProduction} \cap \mathit{Medium} \cap \mathit{Large}$.

Die Synthese mittels des Syntheseframeworks CLS findet für dieses Experiment 144 Lösungen innerhalb von fünf Sekunden. Nach dem Filtern mittels der SMT-Techniken verbleiben 122 Lösungen, wobei die fünf vielversprechendsten in der Tabelle 5.6 aufgelistet sind. Das Filtern mittels des SMT-Solvers und das Interpretieren der Lösungen durch das CLS-Framework benötigte 2 Minuten und 9 Sekunden. Auch in diesem Experiment werden die synthetisierten Simulationsmodelle simuliert und evaluiert.

```
{
    "firstRow": {
      "amount": 40,
      "sheetSize": "medium"
    },
    "secondRow": {
      "amount": 60,
      "sheetSize": "large"
    },
    "maximalTime": 9999,
    "maximalCosts": 4000,
    "optimizationCriteria": "time",
    "useTaxonomy": false,
    "values": [ ... ]
}
```

Listing 5.17: Konfiguration des zweiten Experiments im JSON-Format. Diese sieht eine Produktion von 40 mittleren und 60 großen Blechen, eine Begrenzung der Kosten auf 400 sowie die Bearbeitungszeit als Optimierungskriterium vor.

Während im ersten Experiment die Sortierung der Konfigurationen auch nach der Ausführung und Auswertung der Simulationsläufe übereinstimmte, ist dies im zweiten Experiment nicht der Fall. So erzielt die fünft-vielversprechendste Fabrikkonfiguration in der Simulation ein besseres Ergebnis als die vielversprechendste Konfiguration. Eine mögliche Erklärung hierfür findet sich in der Modellierung des Simulationsmodells. So werden im Simulationsmodell die Bleche auf Paletten, die maximal 50 Bleche umfassen, transportiert. Die Maschinen sind derart modelliert, dass diese die Paletten möglichst vollständig beladen, ehe diese zur nächsten Maschine oder in das Lagersystem gelangen. Hierbei benötigt in der vermeintlich vielversprechendsten Konfiguration die zweite Schneidemaschine eine längere Zeit, um die Palette zu beladen, da die Verarbeitungsgeschwindigkeit dieser geringer im Vergleich zur fünft-vielversprechendsten Konfiguration ist. Dadurch erhält die zweite Biegemaschine in der vielversprechendsten Konfiguration die Palette zu einem späteren Zeitpunkt, sodass die fünft-vielversprechendste Konfiguration einen Vorsprung hat, der die kürzere Gesamtbearbeitungszeit möglicherweise erklärt. Dennoch stellt die ermittelte Sortierung eine zufriedenstellende Lösung dar. So benötigt im Vergleich die rechnerisch ungünstigste Konfiguration 715 Minuten zur Verarbeitung sämtlicher Bleche.

5.5 Diskussion

Im vorliegenden Szenario erweist sich das Filtern mittels der SMT-Techniken als geeignet, da die relevanten Kennzahlen auf die einzelnen Features, genauer gesagt die korrespondierenden

			#1 Schneide-maschine			#2 Schneide-maschine			#1 Biege-maschine		#2 Biege-maschine	
Kosten	Zeit (rechn.)	Zeit (sim.)	Geschw.	Blechgröße	Entladung	Geschw.	Blechgröße	Entladung	Geschw.	Blechgröße	Geschw.	Blechgröße
3940	367	449	H	M	A	N	G	A	H	M	H	G
3840	371	459	N	M	A	M	G	A	H	M	H	G
3740	375	458	M	M	A	N	G	A	H	M	H	G
3940	375	454	N	M	A	H	G	A	M	M	H	G
3940	379	446	H	M	A	M	G	A	H	M	M	G

Tabelle 5.6: Ergebnis der komponentenbasierten Synthese und des Filterns für das zweite Experiment. Dieses umfasst eine Auswahl mit den fünf Fabrikkonfigurationen mit der kürzesten Durchlaufzeit für die Verarbeitung von 40 kleinen und 60 mittelgroßen Blechen, wobei N, M und H für jeweils eine niedrige, mittlere und hohe Geschwindigkeit, A und M für automatisiertes und manuelles Entladen sowie K, M und G für kleine, mittelgroße sowie große Bleche stellvertretend stehen.

Komponenten abgebildet werden können. Jedoch trifft dies nicht auf alle Anwendungsfälle zu, sodass dies im ersten Schritt bei einer geplanten Verwendung der SMT-Techniken überprüft werden sollte. Ferner sollte auch die Komponentensammlung derart gestaltet sein, dass eine sinnvolle Zuordnung von numerischen Werten zu den Komponenten möglich ist.

Der Aufbau der Komponentensammlung erfolgt in diesem Anwendungsfall händisch. Dies war aufgrund der geringen Anzahl an Maschinen, Maschinentypen und Konfigurationspunkten in einem relativ kurzen Zeitraum realisierbar. Bei einer steigenden Anzahl dieser Elemente skaliert dieses Vorgehen womöglich jedoch nicht. Insbesondere bei Änderungen am ursprünglichen Simulationsmodell müssen bei diesem Ansatz die möglicherweise angepassten XML-Fragmente in den Implementierungsdetails der Komponenten aktualisiert werden. Daher wird in den nachfolgenden Kapiteln eine Automatisierung der Erstellung der Komponenten sowie eine Alternative zur Verwendung der XML-Repräsentation eines Simulationsmodells untersucht.

Der Einsatz der SMT-Techniken kann insofern kritisch hinterfragt werden, als dass das Filtern von Wörtern einer Baumgrammatik kein vorgesehenes Einsatzgebiet dieser Techniken ist. Dies spiegelte sich in Übergangslösungen bei der Implementierung wider, bei der etwa das Skript mit der übersetzten Baumgrammatik und domänenspezifischen Randbedingungen um Negationen der zuvor berechneten Modelle erweitert wurde, um weitere Lösungen zu erhalten. Überdies erhöht der Einsatz der SMT-Techniken die Laufzeit der Generierung von Lösungen um ein Vielfaches. Ferner erhöht das notwendige Vorhandensein einer lokalen Installation eines SMT-Solvers auf dem ausführenden System die Einstiegshürde zur Verwendung des Ansatzes. Jedoch kann bei einem cloudbasierten Betrieb des Ansatzes, auch eine Instanz des

SMT-Solvers in der Cloud betrieben werden, sodass eine lokale Installation obsolet ist.

Abschließend sei zu sagen, dass dennoch mithilfe der SMT-Techniken aus einer großen Anzahl an synthetisierten Lösungen, besonders vielversprechende Lösung in Hinblick auf ausgewählte Optimierungskriterium ermittelt werden können. Weiterhin kann festgehalten werden, dass die SMT-Techniken prinzipiell geeignet sind, sofern gewisse Rahmenbedingungen gegeben sind. Überdies sollte die Anzahl zu filternder Lösungen derart hoch sein, dass diese den zusätzlichen Aufwand zur Einbindung der SMT-Techniken rechtfertigen.

5.6 Zusammenfassung und Ausblick

In diesem Kapitel wurde ein praxisnahes Simulationsmodell einer blechverarbeitenden Fabrik in eine Produktlinie zur Synthese von Simulationsmodellvarianten migriert. Hierbei bildeten unterschiedliche Maschinenkonfigurationen die Variabilitätspunkte. Der Aufbau und die Implementierung der Komponentensammlung erfolgte manuell. Weiterhin lag der Fokus in einer Untersuchung der Eignung von Techniken des SMT-Constraint-Solvings zur Filterung von synthetisierten Lösungen. Letztere erfolgte basierend auf numerischen Randbedingungen, die den logistischen Kennzahlen wie der Bearbeitungszeit und Kosten entsprachen. Hierzu wurden numerische Werte als Gewichtungen der Komponenten hinzugefügt, die wiederum zur benötigten Verarbeitungsdauer oder Kosten korrespondierten. Für die Umsetzung wurde ein Framework verwendet, dass eine vom CLS-Framework synthetisierte Baumgrammatik in ein Skript mit SMT-Formeln übersetzt. Gegeben dem Skript und weiteren SMT-Formeln, die domänenspezifische Randbedingungen wie eine Limitierung oder Optimierung der Gesamtbearbeitungszeit oder -kosten umfassten, konnten Wörter der Baumgrammatik ermittelt werden, die den geforderten Randbedingungen entsprechen. Anschließend wurden die Wörter mittels des CLS-Frameworks interpretiert und die entsprechenden Simulationsmodelle produziert. Letztere wurden anschließend in einer exemplarischen Simulationsstudie verwendet, um die vielversprechendste Konfiguration für ein gegebenes Auftragsportfolio zu ermitteln.

In zukünftigen Arbeiten sollte die Automatisierung der Erstellung einer Komponentensammlung fokussiert werden, wobei dies im Rahmen der nachfolgenden Kapitel bereits zum Teil erfolgt. Ferner sollten alternative Methoden zur Filterung von synthetisierten Lösungen erarbeitet werden, die ebenfalls eine Filterung basierend auf numerische Randbedingungen erlauben. Insbesondere sollte bei diesen alternativen Ansätzen die benötigte Rechenzeit der Filterung nicht die benötigte Zeitspanne zur Synthese der Lösungen weit übertreffen.

Kapitel 6

Automatisierte Komponentisierung und Synthese von Kontrollstrategien

In diesem Kapitel wird die komponentenbasierte Softwaresynthese zur Generierung von agentenbasierten Simulationsmodellen eingesetzt, die sich hinsichtlich der Logik und des Verhaltens unterscheiden. Die Logik wird durch die Kontrollflussgraphen der Agenten bestimmt, während das Verhalten durch Kontrollstrategien in Form von Funktionen, die in einer Programmiersprache (z. B. Java) implementiert sind, festgelegt wird. Diese stehen insofern in Beziehung, als eine Variation des Kontrollflussgraphen oftmals eine Anpassung der Kontrollstrategien erfordert. So zum Beispiel, wenn in einer Variante eines Kontrollflussgraphen Prozessbausteine entfallen, die in einer oder mehreren Kontrollstrategien referenziert werden. Ebenso können Prozessbausteine hinzukommen, die in den Kontrollstrategien noch nicht berücksichtigt werden. Daher erfolgt in diesem Kapitel eine Untersuchung der Synthese von Kontrollstrategien, die maßgeschneidert auf zuvor synthetisierte Kontrollflussgraphen sind. In diesem Zusammenhang wird die Zusammenführung der synthetisierten Kontrollflussgraphen und Kontrollstrategien in ein ausführbares Simulationsmodell untersucht. Überdies fokussiert dieses Kapitel eine Automatisierung der Erstellung einer Komponentensammlung zur Synthese der Kontrollflussgraphen. Hierbei basiert die Erstellung auf einem bereits existenten Simulationsmodell. Die Automatisierung stellt einen Beitrag zur Reduzierung des Zeitaufwands der Modellierung von Simulationsmodellen sowie der Einstiegshürde zur Verwendung der entwickelten Technologie dar.

Die Migration eines Simulationsmodells einschließlich der Kontrollstrategien erfolgt in diesem Kapitel in den nachfolgenden Schritten:

1. Automatisierte Erstellung der Komponentensammlung zur Synthese der Kontrollflussgraphen der einzelnen Agenten des zu migrierenden Simulationsmodells

2. Markierung der Variabilitätspunkte der Kontrollflussgraphen der Agenten durch die

anwendende Person in Form von Anpassungen an den Komponentensammlungen

3. Aufteilung der Kontrollstrategien in Komponenten und Festlegung von Bedingungen des Auftretens durch die anwendende Person
4. Automatisierte Erstellung der Komponentensammlung zur Synthese der Kontrollstrategien basierend auf vorheriger Aufteilung durch die Implementierung
5. Durchführung der komponentenbasierten Synthese der Kontrollflussgraphen
6. Durchführung der komponentenbasierten Synthese der Kontrollstrategien für jeden synthetisierten Kontrollflussgraphen
7. Synchronisation der synthetisierten Kontrollflussgraphen und Kontrollstrategien mittels der komponentenbasierten Synthese ausführbarer Simulationsmodelle

Die Abbildung 6.1 zeigt die resultierenden Schichten, die im Rahmen der Migration des bereits existenten Simulationsmodells (unterste Schicht) hin zu den synthetisierten Simulationsmodellen mit den variierten Kontrollflussgraphen und Kontrollstrategien (oberste Schicht) erzeugt werden. Die Schichten illustrieren hierbei die Aufteilung eines Simulationsmodells in die Bestandteile, die wiederum in Komponenten überführt werden, um anschließend zu neuen Kompositionen des Simulationsmodells mittels der komponentenbasierten Synthese zusammengesetzt zu werden. Ferner zeigt die Abbildung 6.1 die resultierenden Komponentensammlungen, die durch farbige Zylinder visualisiert sind und nachfolgend vorgestellt werden. Die erste Komponentensammlung (Zylinder 1) wird im Kontext der Synthese der Kontrollflussgraphen eingesetzt. Daher ist diese Komponentensammlung für jeden Agenten genau einmal vorhanden und beinhaltet Komponenten, welche die Produktion einzelner Prozessbausteine sowie Kompositionen dieser verantworten. Die Top-Level-Komponente dieser Komponentensammlung verantwortet die Produktion unterschiedlicher Varianten des Kontrollflussgraphen, sofern möglich. Die Komponentensammlung, die mit der Ziffer 2 beschriftet ist, wird zur Synthese der Kontrollstrategien verwendet. Da die Synthese der Kontrollstrategien von den synthetisierten Kontrollflussgraphen abhängt, erfolgt die Erstellung der zweiten Komponentensammlung nach der Synthese der Kontrollflussgraphen und erhält zudem eine Referenz auf die synthetisierten Kontrollflussgraphen. Die Komponenten verantworten die Produktion von Fragmenten sowie Kompositionen der jeweiligen Kontrollstrategie. Die Top-Level-Komponente dieser Komponentensammlung produziert alle möglichen Konstellationen der Kontrollstrategien für eine synthetisierte Variante eines Kontrollflussgraphen des aktuellen Agenten. Die Komponentensammlung, die mit der Ziffer 3 beschriftet ist, umfasst Komponenten, die Varianten der Agenten produzieren. Diese Komponenten werden basierend auf den zuvor synthetisierten Kontrollflussgraphen und Kontrollstrategien gebildet. Die Top-Level-Komponente dieser Komponentensammlung verantwortet die Produktion aller möglichen Varianten des Simulationsmodells.

Auf den nachfolgenden Seiten wird das Vorgehen detaillierter erläutert und anhand von zwei praxisnahen Anwendungsfällen demonstriert und evaluiert. Der erste Anwendungsfall umfasst

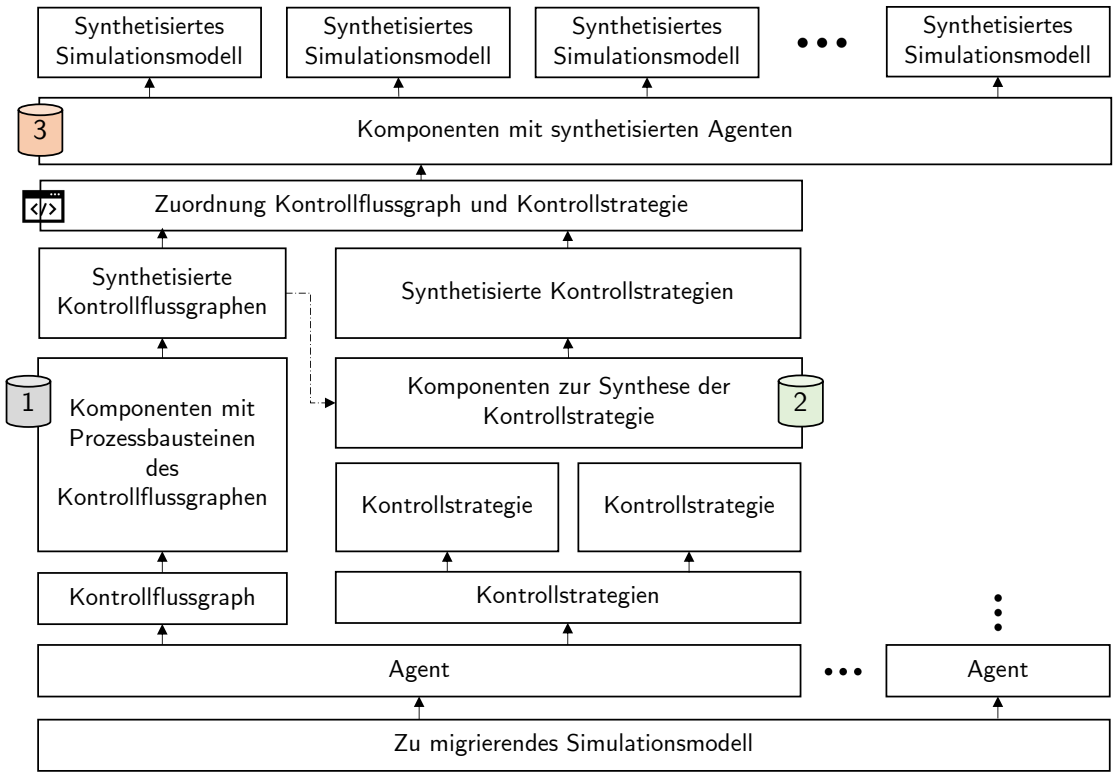


Abbildung 6.1: Übersicht über die unterschiedlichen Schichten ausgehend vom ursprünglichen Simulationsmodell hin zu den synthetisierten Varianten des Simulationsmodells. Die einzelnen Schichten sind dabei jeweils von der darunterliegenden Schicht abgeleitet. So stammen etwa die Agenten aus dem ursprünglichen Simulationsmodell und der Kontrollflussgraph und die Kontrollstrategien aus dem aktuellen Agenten. Die farbig markierten Zylinder kennzeichnen die Komponentensammlungen.

ein fiktives Produktionssystem, während im zweiten Anwendungsfall ein Simulationsmodell eines Bodenblocklagers in eine Produktlinie migriert wird.

6.1 Automatisierter Aufbau von Komponentensammlungen

In diesem Abschnitt wird der automatisierte Aufbau einer Komponentensammlung zur Synthese eines Kontrollflussgraphen thematisiert. Die Automatisierung reduziert den benötigten Zeitaufwand der Migration und ermöglicht diese ohne Kenntnisse in der Programmierung oder hinsichtlich des Syntheseframeworks Combinatory Logic Synthesizer. Im darauffolgenden Abschnitt wird ein teilautomatisierter Ansatz zum Aufbau einer Komponentensammlung zur Synthese von Kontrollstrategien vorgestellt.

6.1.1 Aufbau für die Synthese von Kontrollflussgraphen

Der automatisierte Aufbau einer Komponentensammlung zur Synthese von Kontrollflussgraphen basiert auf der Aufteilung und Typisierung der Komponenten, die in Kapitel 4.4.2 vorgestellt wurde. Das nachfolgende Verfahren beschreibt den automatisierten Aufbau für Simulationsmodelle der Simulationsumgebung AnyLogic 8. Als Basis dient die XML-Repräsentation des Simulationsmodells, die eingelesen und aus der mittels eines entwickelten Algorithmus die Agenten des Simulationsmodells ermittelt werden. Anschließend wird für jeden Agenten eine Komponentensammlung erstellt, sodass gewährleistet ist, dass jede Komponentensammlung ausschließlich die Komponenten enthält, die für den jeweiligen Agenten relevant sind. Dies wiederum resultiert in einer kürzeren Laufzeit der Synthese, insofern, als die Anzahl der Komponenten möglichst gering gehalten wird.

Im nächsten Schritt werden die Abhängigkeiten und logischen Verbindungen der Komponenten in der semantischen Schicht ergänzt. Beispielsweise müssen die Prozessbausteine mittels gerichteter Kanten miteinander verbunden werden, um einen validen Kontrollflussgraphen produzieren zu können. Dabei ist die Reihenfolge der Vorkommen der Prozessbausteine sowie die Anzahl der ein- und ausgehenden Kanten eines Prozessbausteins relevant. Diese Informationen werden durch die Typspezifikationen der Komponenten ausgedrückt. Eine Herausforderung bei dieser Vorgehensweise besteht darin, parallel laufende Verkettungen über einzelne semantische Typspezifikationen auszudrücken. Hierbei ist es insbesondere schwierig auszudrücken, mit welchem Prozessbaustein ein Pfad beginnt und mit welchem Baustein dieser endet. Nachfolgend wird eine Möglichkeit vorgestellt, die es ermöglicht, über das Typsystem die geforderten Informationen abzubilden, in dem der Kontrollflussgraph auf mehrere Teil-Kontrollflussgraphen aufgeteilt wird, die durch entsprechende Komponenten produziert werden.

Hierzu wird in einem ersten Schritt der vorliegende Kontrollflussgraph in eine Datenstruk-

tur eines Graphen überführt. Diese Überführung erfolgt automatisiert und nutzt die Bibliothek *graphs*[1]. Somit stellt die Datenstruktur eine interne Repräsentation des ursprünglichen Kontrollflussgraphen dar. Die Knoten in dieser Datenstruktur korrespondieren zu den Prozessbausteinen im Kontrollflussgraphen. Ferner verfügt jeder Knoten über eine Referenz auf ein Datenobjekt des entsprechenden Prozessbausteins. Die Kanten zwischen den Knoten im Graphen entsprechen den Verbindungen im Kontrollflussgraphen.

Die Überführung des Graphen in eine Komponentensammlung erfolgt schrittweise mittels mehrerer Läufe durch den Graphen. In jedem Lauf wird ausgehend von einem Knoten nach einem festgelegten Muster gesucht. Letzteres charakterisiert sich durch die Anzahl der vorwärts und rückwärts gerichteten Kanten eines Knotens. Für jedes Muster ist eine Übersetzung erarbeitet worden, das vorgibt, wie bei der Erkennung eines Musters, die Knoten in getypte Komponenten überführt werden. Für die Durchläufe wird ein beliebiger Startknoten ausgewählt, der sich dadurch charakterisiert, dass dieser über keine eingehende Kanten verfügt. Im Rahmen der Überführung in eine Komponentensammlung wird der Graph nach jedem erkannten Muster angepasst, sodass diese Auswirkungen auf die nachfolgenden Läufe hat. Die Tabelle 6.1 zeigt die Muster und die Übersetzung anhand eines beispielhaften Graphens. Der Graph in der Spalte *Graph vor Transformation* zeigt das zu erkennende Muster, während der Graph in der Spalte *Graph nach Transformation* den Graphen nach der Übersetzung und Anpassung zeigt. In der Spalte *Konstruierte Komponenten* sind die erzeugten Komponenten aufgelistet. Nachfolgend werden die einzelnen Muster erläutert.

Im ersten Lauf werden Knoten gesucht, die über mehrere ausgehende Kanten verfügen. Gleichzeitig müssen die Pfade, die von diesen Knoten ausgehen, mindestens einen gemeinsamen Folgeknoten besitzen. Im beispielhaften Graphen in der Tabelle 6.1 verfügt der Knoten *a* über Kanten zu den Knoten *b, c* und *d*. Die Pfade haben den gemeinsamen Folgeknoten *e*. Bei diesem Muster werden im Rahmen der Übersetzung sämtliche Pfade in Komponenten ausgelagert. Diese Komponenten verantworten die Produktion der Pfade, einschließlich der Prozessbausteine. Hierfür wird die Datenstruktur `SubProcess` in Verbindung mit der Komponente `SubProcessComponent` verwendet. Im Beispiel werden die Komponenten *sub1, sub2* und *sub3* konstruiert. Ferner werden Komponenten erzeugt, welche die auf dem Pfad liegenden Prozessbausteine produzieren. In diesem Fall werden die Komponenten *b, c* und *d* der Komponentensammlung hinzugefügt. Außerdem wird eine weitere Komponente erzeugt, die den aktuellen Knoten *a* produziert. Diese Komponente erwartet die zuvor erstellten Komponenten, welche die Pfade produzieren, als Argumente. Für das Beispiel wird daher eine gleichnamige Komponente *a* konstruiert. Nach der Erzeugung der Komponenten werden die überführten Knoten einschließlich der Kanten aus dem Graphen entfernt. Anschließend wird eine neue Kante zwischen dem aktuellen Knoten und dem gemeinsamen Folgeknoten der Pfade im Graph ergänzt.

Im zweiten Lauf wird erneut nach einem Muster gesucht, das Knoten mit mehreren ausgehenden Kanten umfasst. Jedoch erfordert dieses Muster nicht das Vorhandensein eines

[1] Bibliothek *graphs* zur Repräsentation von Graphen - https://flowtick.github.io/graphs/

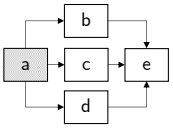
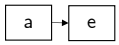
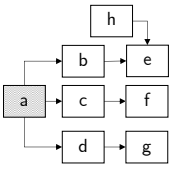
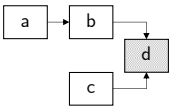
Lauf	Suche nach Knoten mit	Beispiel: Graph vor Transformation	Beispiel: Graph nach Transformation	Beispiel: Konstruierte Komponenten
1	• mehreren ausgehenden Kanten • gemeinsamem Nachfolger			$\Gamma = \Gamma \cap \{$ a : *Sub1* → *Sub2* → *Sub3* → *A*, sub1 : *B* → *Sub1* sub2 : *C* → *Sub2*, sub3 : *D* → *Sub3*, a : *A*, …, e : *E* $\}$
2	• mehreren ausgehenden Kanten			$\Gamma = \Gamma \cap \{$ a : *Sub1* → *Sub2* → *Sub3* → *A*, e : *Sub4* → *E*, sub1 : *B* → *E* → *Sub1*, sub2 : *C* → *F* → *Sub2*, sub3 : *D* → *G* → *Sub3*, sub4 : *H* → *Sub4*, a : *A*, …, h : *H* $\}$
3	• mehreren eingehenden Kanten			$\Gamma = \Gamma \cap \{$ a : *Sub1* → *Sub2* → *Sub3* → *D*, sub1 : *A* → *B* → *Sub1*, sub2 : *C* → *Sub2*, a : *A*, …, d : D $\}$

Tabelle 6.1: Transformation eines Kontrollflussgraphen in eine Komponentensammlung. Der Kontrollflussgraph wird in eine Graph-Datenstruktur überführt und anschließend in drei Durchläufen nach festgelegten Mustern untersucht. Ein Muster wird ausgehend von einem Knoten betrachtet. Sobald ein Muster identifiziert wurde, werden entsprechende Komponenten gebildet und der Komponentensammlung hinzugefügt. Die Tabelle zeigt die Muster anhand mehrerer beispielhaften Graphen und der resultierenden Komponentensammlung.

gemeinsamen Folgeknotens. Auch bei diesem Muster werden die Pfade in Komponenten ausgelagert. Zusätzlich werden bei der Auslagerung der Pfade die Knoten der Pfade durchlaufen und für jeden Knoten wird geprüft, ob dieser über zusätzliche eingehende Kanten verfügt. So erstreckt sich im Beispielgraphen ausgehend vom Knoten *a* ein Pfad über die Knoten *b* und *e*. Letzterer verfügt über eine eingehende Kante, die den Knoten *h* mit dem Knoten *e* verbindet. Diese Information wird bei der Generierung der Typspezifikation der Komponente, die den repräsentierten Prozessbaustein des Knoten *e* produziert, berücksichtigt, indem der eingehende Knoten *h* als Argument erwartet wird. Um genauer zu sein, wird eine Komponente erwartet, die den repräsentierten Prozessbaustein des Knotens *h* produziert. Im Beispiel wird dieser durch die Komponente *sub4* produziert. Die verbleibenden Pfade (*c, f* und *d, g*) werden analog in entsprechende Komponenten überführt. Wie auch beim ersten Muster, werden Komponenten erzeugt, welche die repräsentierten Prozessbausteine der Knoten, die auf den Pfaden liegen, produzieren. Ebenfalls wird eine Komponente erzeugt, die den aktuellen Knoten produziert und als Argumente jene Komponenten erwartet, welche die Pfade produzieren. Nach der Erkennung dieses Musters werden alle Pfade ausgehend vom aktuellen Knoten entfernt, sodass ausschließlich der aktuelle Knoten im Graph verbleibt.

Im dritten Lauf wird nach einem Muster gesucht, das Knoten mit mehreren eingehenden Kanten umfasst. Bei der Erkennung solch eines Knotens werden die eingehenden Pfade des Knotens rückwärts durchlaufen. Die Überführung dieser Pfade in Komponenten erfolgt analog zum vorherigen Fall. Das Beispiel in der Tabelle 6.1 zeigt dies für den Knoten *d*. Die Pfade werden durch die Komponenten *sub1* und *sub2* produziert. Die Komponente *d* erwartet als Argumente die Zieltypen der Komponenten, welche die Pfade produzieren. Wie auch beim zweiten Muster werden sämtliche Pfade aus dem Graphen entfernt und es verbleibt allein der aktuelle Knoten.

Nach dem dritten Lauf erfolgt ein weiterer, finaler Durchlauf des Graphen. In diesem Lauf werden alle Knoten, die noch nicht in Komponenten überführt sind, ermittelt und in Komponenten überführt. Bei den bisherigen Durchläufen wurden die überführten Knoten protokolliert und anhand dessen können die noch nicht überführten Knoten identifiziert werden. Nach dem finalen Lauf erfolgt die Konstruktion eines Syntheseziels basierend auf dem angepassten Graphen. Das Syntheseziel bezeichnet in diesem Zusammenhang die Typspezifikation der Top-Level-Komponente, welche den Kontrollflussgraphen des aktuellen Agenten produziert. Zur Konstruktion des Syntheseziels wird der angepasste Graph vom Start bis zum Ende durchlaufen und für jeden Knoten wird der Zieltyp der entsprechenden Komponente zur Argumentliste der Top-Level-Komponente hinzugefügt. An dieser Stelle sei anzumerken, dass durch die Anpassungen des Graphen im Rahmen des dritten Laufs ein Startknoten verbleibt. Ferner sei zu erwähnen, dass die Komponenten zum Teil Kompositionen von Prozessbausteinen produzieren und somit eine verschachtelte Struktur vorherrscht. Dadurch kann mittels der semantischen Typspezifikationen ein komplexer Kontrollflussgraph beschrieben werden.

An dieser Stelle sei auf ein implementierungstechnisches Detail hinzuweisen. So werden in der Implementierung die Pfade, die genau ein Element umfassen, nicht in Subprozesse

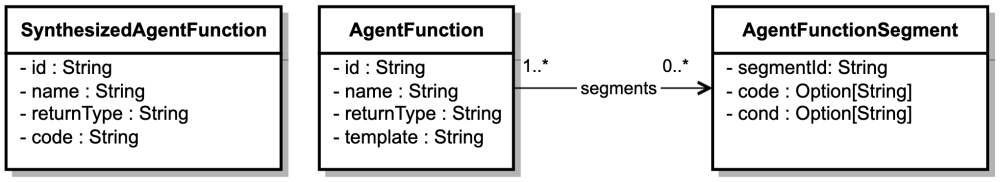


Abbildung 6.2: Datenstruktur zur Repräsentation von unvollständigen Funktionen, Funktionssegmenten sowie synthetisierten Funktionen

ausgelagert. Stattdessen wird das Element, das auf dem Pfad liegt, unmittelbar über den Zieltypen referenziert. Im obigen Beispiel ergibt sich für den ersten Lauf somit der semantische Typ $B \rightarrow C \rightarrow D \rightarrow A$ für die Komponente *a*.

6.1.2 Aufbau für die Synthese von Kontrollstrategien

Nachfolgend wird ein Vorgehen vorgestellt, bei dem teilautomatisiert Komponenten zur Synthese von Kontrollstrategien erzeugt werden. Die Generierung der Komponentensammlung erfolgt nicht voll automatisiert, da die Kontrollstrategien anwendungsfallspezifisch und häufig komplex sind. Zunächst werden die Datenstrukturen vorgestellt, die den Aufbau einer Komponentensammlung ermöglichen. Ferner wird im Folgenden der Begriff der Funktion verwendet, da im Nachfolgenden insbesondere Kontrollstrategien synthetisiert werden, die durch Funktionen in einer höheren Programmiersprache implementiert werden.

Die Abbildung 6.2 zeigt die Datenstrukturen zur Repräsentation von unvollständigen, vervollständigten Funktionen sowie Funktionsfragmenten. Die Klasse `AgentFunction` repräsentiert eine Funktion eines Agenten beziehungsweise Simulationsmodells. Hierzu beinhaltet diese Variablen zur Repräsentation der Kennung (`id`), des Namens (`name`), des Rückgabetyps (`returnType`) sowie des lückenhaften Quellcodes `template`. Ferner verfügt die Klasse über eine Liste `segments` mit Funktionsfragmenten, aus denen die Funktion zusammengesetzt werden kann. Letztere sind vom Datentyp `AgentFunctionSegment`, die durch eine gleichnamige Klasse repräsentiert werden. Letztere beinhaltet Variablen zur Repräsentation der Kennung (`segmentId`), des Quellcodes (`code`) und einer Bedingung `cond`, die festlegt, wann das Fragment zur Vervollständigung einer Funktion verwendet werden soll. Die Bedingung beinhaltet den Namen eines Prozessbausteins des Kontrollflussgraphen. Sofern dieser Prozessbaustein in einem synthetisierten Kontrollflussgraph vorhanden ist, soll das Fragment in der Kontrollstrategie verwendet werden. Die Klasse `SynthesizedAgentFunction` repräsentiert eine vervollständigte Funktion und beinhaltet wie die Klasse `AgentFunction` ebenfalls Variablen zur Repräsentation der Kennung, des Namens und des Rückgabetyps. Statt eines lückenhaften Quellcodes verfügt diese über die Variable `code`, die den vollständigen Quellcode der Funktion beinhaltet. Bei dieser Datenstruktur handelt es sich um den nativen Zieltypen der Komponenten, die eine vervollständigte Kontrollstrategie produzieren.

Nachfolgend werden die Komponenten vorgestellt, die zur Synthese der Kontrollstrategien

verwendet werden. Auch diese Komponenten können dynamisch instantiiert und für beliebige Kontrollstrategien eingesetzt werden. Hierfür sind drei Komponenten vorgesehen, die jeweils

1. Quellcodefragmente,
2. vervollständigte Kontrollstrategien sowie
3. Konfigurationen mit vervollständigten Kontrollstrategien

produzieren. In Listing 6.1 ist die Implementierung der ersten Komponente dargestellt. Die Komponente erwartet keine Argumente in der Typspezifikation (vgl. native Typspezifikation in Zeile 2). Eine Instanz der Komponente kann über den Aufruf des Konstruktors in der Zeile 1 erzeugt werden. Der Konstruktor erwartet ein Objekt des Typs `AgentFunctionSegment` sowie eine semantische Typspezifikation der Komponente. Letztere wird automatisiert von der Implementierung generiert. In der Zeile 2 wird innerhalb der Implementierungsdetails das übergebene Quellcodefragment als Produktionsergebnis zurückgegeben. Dadurch ergibt sich der native Zieltyp `AgentFunctionSegment` dieser Komponente.

```
1 class FunctionSegmentComponent(functionSegment: AgentFunctionSegment, val
   ↪  semanticType: Type) {
2   def apply(): AgentFunctionSegment = functionSegment
3 }
```

Listing 6.1: Implementierung einer Komponente zur dynamischen Instantiierung und Produktion eines Quellcode-Fragments einer Kontrollstrategie.

Listing 6.2 zeigt die Implementierung der zweiten Komponente. Der Konstruktor dieser Komponente in der Zeile 1 erwartet neben dem semantischen Typen ein Objekt des Datentyps `AgentFunction`, das die unvollständige Kontrollstrategie beinhaltet. Diese wird in den Implementierungsdetails verwendet, um die vervollständigte Kontrollstrategie zu produzieren. Die Variabilitätspunkte dieser Komponente bilden die Quellcodefragmente, die zur Vervollständigung verwendet werden, und deren Auswahl über die Typspezifikation gesteuert wird. Wie in Kapitel 4.4.2 erläutert, erwarten die Komponenten in ihrer Implementierung die Argumente in Form von Listen, die wiederum die Argumente beinhalten. Dementsprechend umfasst das Argument `args` der `apply`-Methode in Zeile 2 sämtliche Quellcodefragmente, welche die Komponente zur Produktion der vervollständigten Kontrollstrategie benötigt. Die Erstellung und die Typisierung dieser Listen erfolgt automatisiert. In den Implementierungsdetails der Komponente (vgl. Zeile 2 bis 6) wird die Liste mit den Quellcodefragmenten mittels der Scala-Funktion `foldLeft` durchlaufen. Letztere erhält einen Startwert (hier die lückenhafte Kontrollstrategie) und eine *Combining*-Operation, welche ein Element der Liste entnimmt und den Startwert modifiziert. In der darauffolgenden Iteration wird das nächste Element entnommen und die Änderung erfolgt anhand des zuvor geänderten Startwerts. In diesem Fall werden die Platzhalter im Quellcode der lückenhaften Kontrollstrategie durch entsprechende

```
1 class FunctionComponent(agentFunction: AgentFunction, val semanticType:
   ↪ Type) {
2  def apply(args: List[AgentFunctionSegment]): SynthesizedAgentFunction =
   ↪ {
3   val code = args.foldLeft(agentFunction.template)((codeTemplate,
   ↪ functionComponent) => codeTemplate.replace("$" +
   ↪ functionComponent.id, functionComponent.code))
4
5   SynthesizedAgentFunction(agentFunction.id, agentFunction.name,
   ↪ agentFunction.returnType, code)
6  }
7 }
```

Listing 6.2: Implementierung einer Komponente zur dynamischen Instantiierung und Produktion einer vervollständigten Kontrollstrategie.

```
1 class FunctionConfigurationComponent(val semanticType: Type) {
2  def apply(args: List[SynthesizedAgentFunction]):
   ↪ Seq[SynthesizedAgentFunction] = args
3 }
```

Listing 6.3: Implementierung einer Komponente zur dynamischen Instantiierung und Produktion sämtlicher Varianten einer Kontrollstrategie.

Quellcodefragmente ersetzt. Hierzu wird die Funktion `replace` der Programmiersprache Scala verwendet. Nach der Ersetzung sämtlicher Quellcodefragmente, wird in der Zeile 6 eine Instanz des Objekts des Datentyps `SynthesizedAgentFunction` erzeugt. Hierzu werden ausgewählte Attribute des `AgentFunction`-Objekts und der zuvor vervollständigte Quellcode als Argumente übergeben. Das instantiierte Objekt wird als Produktionsergebnis zurückgegeben. An dieser Stelle sei anzumerken, dass vorausgesetzt wird, dass sämtliche Platzhalter durch mindestens ein Quellcodefragment abgedeckt sind.

In Listing 6.3 ist die Implementierung der dritten Komponente *FunctionConfigurationCombinator* dargestellt. Diese bildet die Top-Level-Komponente und verantwortet die Produktion einer Liste mit vervollständigten Kontrollstrategien. Hierzu erwartet der Konstruktur der Komponente in der Zeile 1 die semantische Typspezifikation, die wiederum als Argumente die vervollständigten Kontrollstrategien erwartet. Letztere sind in einer Liste verschachtelt, die durch eine gesonderte Komponente produziert wird. Diese Liste mit den vervollständigten Kontrollstrategien bildet in der Zeile 3 das Produktionsergebnis der Komponente. Die zuvor vorgestellten Datenstrukturen und Komponenten ermöglichen den Aufbau von Komponentensammlunge zur Synthese beliebiger Kontrollstrategien.

Nachfolgend wird anhand eines Beispiels demonstriert wie eine Komponentensammlung

aufgebaut wird. Hierzu wird die XML-Repräsentation des Simulationsmodells verwendet, um die Kontrollstrategien zu ermitteln und in die zuvor vorgestellten Datenstrukturen zu überführen. Die XML-Repräsentation umfasst die Kontrollstrategienn seperiert nach den Agenten. Die Kontrollstrategien sind wiederum in XML-Fragmente unterteilt. Das Listing 6.4 zeigt ein XML-Fragment, das die beispielhafte Funktion *InitWorkingStations* (vgl. Name in der Zeile 6) repräsentiert. Dieses XML-Fragment beinhaltet Metainformationen hinsichtlich der Kontrollstrategie (vgl. Zeile 1 bis 6) als auch hinsichtlich der visuellen Repräsentation im Simulationsmodell (vgl. Zeile 7 bis 10). Das XML-Tag `Body` beinhaltet den Quellcode der Kontrollstrategie (vgl. Zeile 11 bis 13). Der enthaltene Quellcode beinhaltet lediglich die Anweisungen der Kontrollstrategie. Die Funktionssignatur ist kein Bestandteil, sondern wird von der Simulationsumgebung AnyLogic 8 zur Laufzeit ergänzt.

```
1  <Function AccessType="default" StaticFunction="false">
2    <ReturnModificator>VOID</ReturnModificator>
3    <ReturnType><![CDATA[double]]></ReturnType>
4    <Id>1627296210846</Id>
5    <PublicFlag>false</PublicFlag>
6    <Name><![CDATA[InitWorkingStations]]></Name>
7    <X>230</X><Y>210</Y>
8    <Label><X>10</X><Y>0</Y></Label>
9    <PresentationFlag>true</PresentationFlag>
10   <ShowLabel>true</ShowLabel>
11   <Body><![CDATA[this.WorkingStation1.initWorkingStation();
12         this.WorkingStation2.initWorkingStation();
13         this.WorkingStation3.initWorkingStation();]]></Body>
14 </Function>
```

Listing 6.4: Kontrollstrategie zur Initialisierung von Arbeitsstationen als XML-Fragment im proprietären Format der Simulationsumgebung AnyLogic 8.

Dieses XML-Fragment wird im Rahmen der Überführung in Komponenten zunächst in ein Objekt der Datenstruktur `AgentFunction` überführt. Auch hier wird die Bibliothek *Xtract* eingesetzt (vgl. Kapitel 4.4.2), um das XML-Fragment automatisiert in ein Datenobjekt zu überführen. An dieser Stelle wird eine initiale Aufteilung des Quellcodes in Fragmente vorgenommen, die den Quellcode in ein einzelnes Quellcodefragment auslagert und diesem die Kennung *code* zuweist. Im nächsten Schritt werden die erzeugten Datenobjekte als JSON-Objekte serialisiert und die REST-Schnittstelle zur Verfügung gestellt. Dies hat den Hintergrund, dass die Kontrollstrategien händisch durch die anwendende Person im webbasierten Frontend aufgeteilt werden sollen.

In Listing 6.5 ist das serialisierte JSON-Objekt mit der Kontrollstrategie *initWorkingStations* dargestellt. Die JSON-Serialisierung enthält eine Auflistung der Kontrollstrategien aller Agenten. In dem aktuellen Beispiel ist die Kontrollstrategie ein Bestandteil des Hauptagenten *Main* (vgl. Zeile 4). Die Serialisierung einer Kontrollstrategie entspricht dem Aufbau des entspre-

```
1  {
2    "agents": [
3     {
4      "name": "Main",
5      "functions": [
6       {
7        "id": "1627296210846",
8        "name": "initWorkingStations",
9        "metaInformations": { ... },
10       "accessType": "default",
11       "isStaticFunction": false,
12       "returnModificator": "VOID",
13       "returnType": "double",
14       "template": "$code",
15       "components": [
16        {
17         "id": "code",
18         "code": "this.workingStation1.initWorkingStation();\n
           ↪  this.workingStation2.initWorkingStation();\n
           ↪  this.workingStation3.initWorkingStation();"
19        }
20       ]
21      }
22     ]
23    }
24  ]
25 }
```

Listing 6.5: Serialisierung eines Agenten mit einer Funktion im JSON-Format.

chenden Datenobjekts `AgentFunction`. So finden sich etwa die Metainformationen einer Kontrollstrategie auch im Objekt mit der Kennung `metaInformation` in der Zeile 9 sowie in den Feldern `accessType`, `isStaticFunction`, `returnModificator` und `returnType` in der Zeilen 10 bis 13. Der Quellcode der Kontrollstrategie befindet sich im Feld `template` (vgl. Zeile 14) mit dem unvollständigen Quellcode sowie dem Array `components` (vgl. Zeile 15 bis 20) mit den Quellcodefragmenten, die zur Vervollständigung eingesetzt werden.

6.2 Markierung von Variabilitätspunkten mittels Anpassungen der Komponenten

In diesem Unterkapitel werden die möglichen Anpassungen an den Komponentensammlungen thematisiert. Diese erlauben die Markierung von Variabilitätspunkten und damit

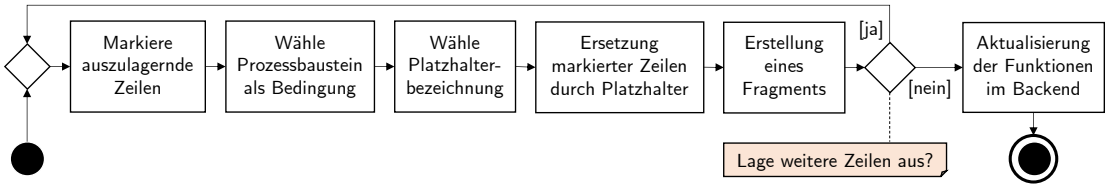


Abbildung 6.3: Ablauf der benutzergesteuerten Aufteilung und Überführung einer Kontrollstrategie in Komponenten. Der Ausgangspunkte ist eine bereits existente Kontrollstrategie. Das Vorgehen wird im webbasierten Frontend implementiert.

die Bildung von Varianten. Zunächst wird die Komponentensammlung zur Synthese der Kontrollflussgraphen angepasst, ehe die Komponentensammlungen zur Synthese der Kontrollstrategien folgen. Dies hat den Hintergrund, dass eine Aufteilung der Kontrollstrategien von den Anpassungen der Komponentensammlung der Kontrollflussgraphen abhängig ist.

Die Anpassung der Komponentensammlung zur Synthese von Kontrollflussgraphen erfolgt im Wesentlichen durch zwei Operationen:

- Duplizieren von Komponenten
- Anpassen der Typspezifikation

Das Duplizieren einer Komponente dient als Multiplikator der Lösungen, da hierdurch Alternativen einer Komponente geschaffen werden. Die Operation ist insbesondere zur Synthese von Prozessbausteinen mit unterschiedlicher Parametrisierung notwendig. Hierzu wird die Komponente dupliziert und die Parametrisierung des Duplikats angepasst. Im Rahmen der Synthese werden dann mindestens zwei Varianten synthetisiert, die jeweils eine Ausprägung dieses Prozessbausteins beinhalten. Eine Kombination der beiden Operationen ist dann sinnvoll, wenn Komponenten, die Kompositionen von Prozessbausteinen produzieren, variiert werden sollen. In diesem Fall wird die entsprechende Komponente dupliziert und anschließend folgt eine Anpassung der Typspezifikation des Duplikats. Letzteres ermöglicht eine Anpassung der Komposition der Prozessbausteine, in dem etwa die Argumentliste angepasst wird. Die beiden Operationen sind in der entwickelten Benutzeroberfläche des webbasierten Frontends durchführbar. Im Rahmen der Experimente dieses Kapitel werden die Anpassungen zur Markierung von Variabilitätspunkten demonstriert.

Wie bereits erwähnt, erfolgt die Aufteilung der Kontrollstrategien in Quellcodefragmente durch die Interaktion mit der anwendenden Person. Hierfür wurde ein Verfahren entwickelt, bei dem die anwendende Person die Komponenten nicht händisch programmiert, sondern die Webanwendung verwendet. Die Abbildung 6.3 fasst den Prozess der Aufteilung sowie der automatisierten Erstellung von Quellcodefragmenten zusammen. Dieser Prozess ist im webbasierten Frontend der entwickelten Technologie implementiert. Als Ausgangsbasis dient die JSON-Serialisierung einer Kontrollstrategie. Im ersten Schritt markiert die anwendende Person eine oder mehrere Zeilen des Quellcodes, die in ein Quellcodefragment ausgelagert

werden sollen. Nach der Markierung legt die anwendende Person einen Prozessbaustein fest, der im synthetisierten Kontrollflussgraphen vorhanden sein muss, damit die zuvor markierten Zeilen in die Kontrollstrategie eingesetzt werden. Dies stellt somit die Bedingung dar, die das Vorhandensein von Quellcodefragmenten in der Kontrollstrategie an die Existenz von Prozessbausteinen in einem Kontrollflussgraphen bindet. Anschließend wählt die anwendende Person eine Bezeichnung, die als Platzhalter für das Fragment dient. Letzterer ersetzt im Quellcode die markierten Zeilen, sodass der Quellcode lückenhaft ist. Im Anschluss daran wird ein Quellcodefragment erzeugt und in der JSON-Serialisierung ergänzt. Hierauf wird die anwendende Person gefragt, ob weitere Zeilen der Kontrollstrategie in Fragmente aufgeteilt werden sollen. Sofern dies zutrifft, wiederholt sich das Vorgehen. Andernfalls wird die JSON-Serialisierung über die REST-Schnittstelle an das Backend-System versendet. Die Instantiierung und Typisierung der Komponentensammlung zur Synthese der Kontrollstrategien basierend auf der JSON-Serialisierung erfolgt im Backend-System im Rahmen der Synthese der Simulationsmodelle.

```
1 this.workingStation1.initWorkingStation();
2 this.workingStation2.initWorkingStation();
3 this.workingStation3.initWorkingStation();
4 this.initStatistics();
```

Listing 6.6: Kontrollstrategie, die aus drei Aufrufen von Funktionen besteht, die abhängig von der Struktur des Kontrollflussgraphen sind.

Nachfolgend wird die Aufteilung anhand einer beispielhaften Kontrollstrategie demonstriert, deren Quellcode in Listing 6.6 dargestellt ist. Im ersten Schritt wird die Zeile 1 zur Auslagerung in ein Quellcodefragment markiert. Als Bedingung wird das Vorhandensein eines Prozessbausteins mit dem Bezeichnung *workingStation1* vorausgesetzt, da die markierte Quellcodezeile sich auf diesen bezieht. Als Platzhalter wird *InitWorkingStation1* gewählt. Ebenso werden die Zeilen 2 und 3 markiert und in Quellcodefragmente ausgelagert. Da die Zeile 4 unabhängig von der Struktur des Kontrollflussgraphen ist, wird diese nicht ausgelagert und verbleibt daher im lückenhaften Quellcode. Listing 6.7 zeigt den unvollständigen Quellcode der Kontrollstrategie.

```
1 $InitWorkingStation1
2 $InitWorkingStation2
3 $initWorkingStation3
4 this.initStatistics();
```

Listing 6.7: Kontrollstrategie mit Platzhaltern, die im Rahmen der Generierung durch Quellcode-Fragmente ersetzt werden.

In Listing 6.8 ist die JSON-Serialisierung des Quellcodefragments abgebildet, welche die erste Zeile des Quellcodes umfasst. Das Feld *id* korrespondiert zum Platzhalter und das Feld *code* beinhaltet den ausgelagerten Quellcode.

```
{
    "id": "initWorkingStation1",
    "code": "this.workingStation1.initWorkingStation()";
}
```

Listing 6.8: JSON-Serialisierung eines Agenten mit einer Kontrollstrategie initWorkingStations nach der Aufteilung in Fragmente.

Dies demonstriert, wie eine Funktion in Quellcodefragmente aufgeteilt und diese an das Vorhandensein von Prozessbausteinen im Kontrollflussgraphen gebunden werden können. Die Erläuterung der Instantiierung der Komponentensammlung sowie die Typisierung der Komponenten werden im nachfolgenden Abschnitt erläutert und für das Beispiel demonstriert.

6.3 Synchronisation und Zusammenführung der Synthesevorgänge

In diesem Unterkapitel wird der ganzheitliche Ablauf zur komponentenbasierten Synthese eines Simulationsmodells beschrieben. Der Ablauf verortet sich nach der vorhergehenden Anpassung durch die anwendende Person und besteht aus drei aufeinander aufbauenden Schritten:

1. Komponentenbasierte Synthese der Kontrollflussgraphen,
2. Komponentenbasierte Synthese der Kontrollstrategien und
3. Komponentenbasierte Synthese des Simulationsmodells.

Im ersten Schritt werden alle Agenten des zu migrierenden Simulationsmodells iteriert und es wird für jeden Agenten eine Komponentensammlung instantiiert. Die Typspezifikationen der Komponenten entsprechen den Anpassungen der anwendenden Person. Gemäß der Syntheseziele der Agenten erfolgt die komponentenbasierte Synthese der Kontrollflussgraphen, wobei letztere möglicherweise in der Synthese mehrerer Varianten resultiert. Hierbei sei zu betonen, dass sämtliche Lösungsvarianten in den nachfolgenden Schritten verarbeitet werden. Die synthetisierten Kontrollflussgraph werden im jeweiligen Agenten eingesetzt und ersetzen den ursprünglichen Kontrollflussgraphen. Die angepassten Agenten sowie die Baumgrammatik der Inhabitanten werden in einer Liste zwischengespeichert, auf die in den nachfolgenden Schritten zurückgegriffen wird.

Im zweiten Schritt folgt die Synthese der Kontrollstrategien, die auf der Aufteilung durch die anwendende Person basiert. Zunächst werden die JSON-Serialisierungen der Kontrollstrategien durchlaufen und für jede Kontrollstrategie wird ein Objekt der Datenstruktur `AgentFunction` instantiiert. Analog werden Instanzen des Datentyps `AgentFunctionSegment`

zur Repräsentation der Quellcodefragmente erstellt. Daraufhin folgt die Instantiierung von Komponente des Datentyps `FunctionSegmentComponent` zur Produktion der Quellcodefragmente. Der semantische Typ dieser Komponente wird automatisiert generiert, indem die Bezeichnung des Platzhalters als semantischer Typ verwendet wird. Nach der Erstellung der Komponenten zur Produktion der Quellcodefragmente werden die Komponenten instantiiert, welche die vervollständigte Kontrollstrategie produzieren. Hierzu wird für jeden synthetisierten Kontrollflussgraphen eine Komponente der Klasse `FunctionComponent` erstellt, welche die zugeschnittene Kontrollstrategie produziert. Dementsprechend wird der semantische Typ dieser Komponenten automatisiert erzeugt. Hierbei wird berücksichtigt, ob der geforderte Prozessbausteine eines Quellcodefragments in der synthetisierten Variante des Kontrollflussgraphen vorhanden ist. Sofern dies der Fall ist, wird der Zieltyp der Komponente, die das Quellcodefragment produziert, in die Argumentliste der Komponente aufgenommen. Andernfalls wird das Quellcodefragment übersprungen. Der Zieltyp dieser Komponente entspricht dem großgeschriebenen Namen der Kontrollstrategie. Danach wird eine Komponente erzeugt, die sämtliche Kontrollstrategien für einen Agenten produziert. Die Argumentliste dieser Komponente wird ebenfalls automatisiert erzeugt und enthält die Zieltypen der Komponenten, welche die vervollständigten Kontrollstrategien produzieren. Der semantische Zieltyp setzt sich aus einer Intersektion des Typs *FunctionConfiguration* und des Agentennamen als Typ zusammen, während der native Zieltyp eine Liste mit Objekten des Datentyps `Synthesized-AgentFunction` ist.

Die Konstruktion der Komponentensammlung wird anhand des Beispiels der Kontrollstrategie *initWorkingStations* demonstriert. Da diese in Form von drei Quellcodefragmente vorliegen, werden drei Komponenten des Datentyps `FunctionSegmentComponent` instantiiert. Die semantischen Zieltypen der Komponenten leiten sich anhand der gewählten Platzhalter der Quellcodefragmente ab, sodass sich die Typen *InitWorkingStation1*, *InitWorkingStation2* und *InitWorkingStation3* ergeben. Der native Zieltyp aller Komponenten lautet `AgentFunctionSegment`. Da die Instantiierung der Top-Level-Komponente das Vorhandensein eines synthetisierten Kontrollflussgraphen bedingt, wird nachfolgend angenommen, dass dieser existiert und auch die Prozessbausteine *WorkingStation1* und *WorkingStation2* enthält. Gemäß der zuvor festgelegten Bedingungen der Quellcodefragmente sollen daher ausschließlich die Komponenten mit den Zieltypen *InitWorkingStation1* und *InitWorkingStation2* bei der Synthese der Kontrollstrategie berücksichtigt werden, sodass sich die Argumentliste der Top-Level-Komponente aus den genannten Zieltypen zusammensetzt. Der Zieltyp der Top-Level-Komponente lautet *InitWorkingStations*, das dem großgeschriebenen Namen der Kontrollstrategie entspricht. Die Tabelle 6.2 zeigt die resultierende Komponentensammlung für das Beispiel, wobei die ersten drei Zeilen den Komponenten zur Produktion der Quellcodefragmente, die vierte Zeile der Komponente zur Produktion der vervollständigten Kontrollstrategie und die letzte Zeile der Top-Level-Komponente zur Produktion sämtlicher Kontrollstrategien des Agenten entsprechen.

Im dritten Schritt erfolgt das Einsetzen der synthetisierten Kontrollstrategien in die entsprechenden Varianten der Agenten. Anschließend werden Letztere in Komponenten einer

Komponente	Nativer Typ	Semantischer Typ
FunctionSegmentComp.	AgentFunctionSegment	*InitWorkingStation1*
FunctionSegmentComp.	AgentFunctionSegment	*InitWorkingStation2*
FunctionSegmentComp.	AgentFunctionSegment	*InitWorkingStation3*
FunctionComp.	AgentFunctionSegment → AgentFunctionSegment → AgentFunction	*InitWorkingStation1* → *InitWorkingStation2* → *InitWorkingStations*
FunctionConfigurationComp.	AgentFunction → List[Synthesized-AgentFunction]	*InitWorkingStation* → *FunctionConfiguration* ∩ *Main*

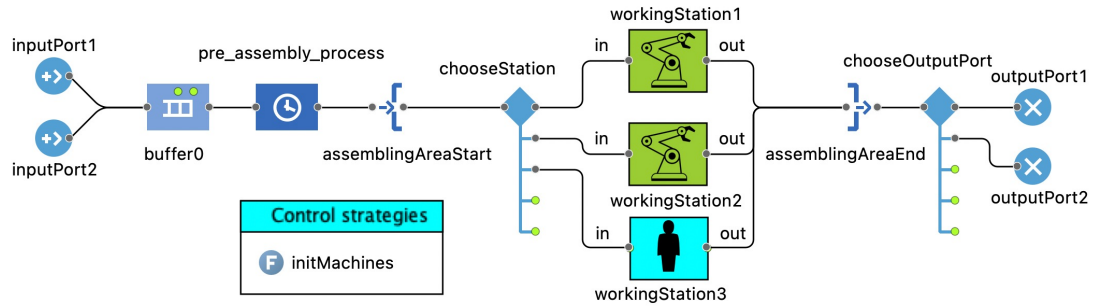
Tabelle 6.2: Exemplarische Komponentensammlung zur Synthese einer Kontrollstrategie.

weiteren Komponentensammlung überführt, die zur Synthese des ausführbaren Simulationsmodells verwendet wird. Die Top-Level-Komponente dieser Komponentensammlung verantwortet die Produktion aller möglichen Konstellationen der synthetisierten Agenten. Dementsprechend erwartet die Komponente in ihrer Argumentliste n Argumente, wobei n der Anzahl der Agenten im Simulationsmodell entspricht. In den Implementierungsdetails der Top-Level-Komponente werden die einzelnen Agenten in einer Liste konsolidiert und können mittels eines implementierten Übersetzers in das proprietäre Format der Simulationsumgebung AnyLogic 8 überführt.

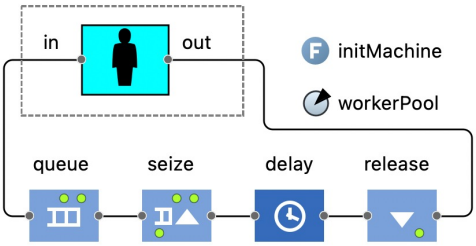
6.4 Durchführung von Experimenten: Fiktives Produktionssystem

Im ersten Experiment wird ein Simulationsmodell in eine Produktlinie migriert, das ein einfaches, fiktives Produktionssystem repräsentiert. Das Simulationsmodell stellt eine Erweiterung eines Anwendungsbeispiels aus [127] dar. Die Modellierung erfolgte im Rahmen dieser Dissertation und wurde in der Simulationsumgebung AnyLogic 8 als ein agentenbasiertes Simulationsmodell vorgenommen. Das modellierte Produktionssystem ermöglicht die Verarbeitung einer variablen Anzahl an Aufträgen, die eine optionale Vorverarbeitung und einen Fertigungsprozess, der durch eine von maximal fünf Arbeitsstationen durchgeführt wird, durchlaufen.

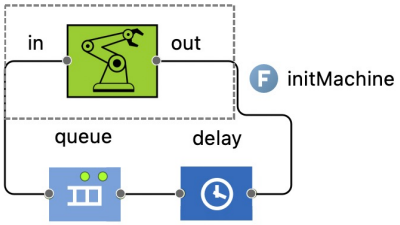
Die Abbildung 6.4 zeigt die Kontrollflussgraphen der Agenten des Simulationsmodells. Der Hauptagent umfasst die ganzheitliche Logik des Produktionssystems. Der Kontrollflussgraph dieses Agenten ist in der Abbildung 6.4a dargestellt. Die Prozessbausteine *inputPort1* und *inputPort2* stellen die initialen Prozessbausteine dar, welche Quellen zur Erzeugung von Aufträgen im System darstellen. Nach der Erzeugung durchlaufen die Aufträge einen Puffer *buffer0* sowie eine Vorverarbeitung *preAssemblyProcess*, die 30 bis 45 Sekunden benötigt. Anschließend betreten die Aufträge einen eingeschränkten Bereich, deren Anfang durch den



(a) Ausschnitt aus dem Simulationsmodell des Hauptagenten im Simulationsmodell des einfachen Produktionssystems



(b) Agent, der eine manuelle Maschine repräsentiert



(c) Agent, der eine automatisierte Maschine repräsentiert

Abbildung 6.4: Agenten im Simulationsmodell des einfachen Produktionssystem

Prozessbaustein *assemblingAreaStart* und deren Ende durch *assemblingAreaEnd* markiert ist. Eingeschränkte Bereiche ermöglichen in AnyLogic 8 in einem festgelegten Bereich die Anzahl der darin befindlichen Agenten (z. B. die Aufträge) zu limitieren. Der Entscheidungsblock *chooseMachine* verantwortet die Auswahl einer verfügbaren Arbeitsstation. Hierbei stehen drei Arbeitsstationen zur Verfügung, die durch separate Agenten modelliert sind. Nach der Verarbeitung eines Auftrags durch eine Arbeitsstation und dem Austritt aus dem eingeschränkten Bereich folgt ein weiterer Entscheidungsblock *chooseOutputPort*, der einen Auftrag auf eine der verfügbaren Senke *outputPort1* oder *outputPort2* leitet. Diese Senken markieren die erfolgreiche Bearbeitung des Auftrages und vernichten den Auftrag. Ferner umfasst der Hauptagent die schon bekannte Kontrollstrategie *InitWorkingStations*, welche die Arbeitsstationen initialisiert. Weiterhin enthält der Hauptagent einen sogenannten *ResourcePool*, der eine gegebene Anzahl an Werker umfasst.

Wie zuvor erwähnt, werden die Aufträge durch Arbeitsstationen verarbeitet. Im Produktionssystem können zwei Arten von Arbeitsstationen unterschieden werden. Erstere wird durch den Agenten, der in der Abbildung 6.4b dargestellt ist, modelliert und repräsentiert eine *manuelle* Arbeitsstation, die einen Werker benötigt. Im Kontrollflussgraphen dieses Agenten wird der zu bearbeitende Auftrag zunächst in einen Puffer *queue* geschoben, ehe ein Werker durch den Prozessbaustein *seize* angefordert wird und die Bearbeitung durch den Prozessbaustein *delay* erfolgt. Die Bearbeitungszeit beträgt zwischen 50 und 60 Sekunden. Danach wird der Werker durch den Prozessbaustein *release* freigegeben. Die zweite Art der Arbeitsstation ist durch

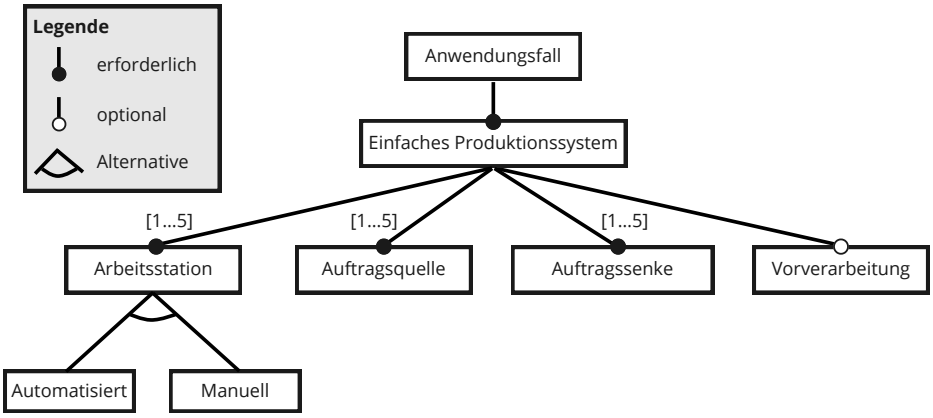


Abbildung 6.5: Feature-Diagramm, das die Variabilitätspunkte des einfachen Produktionssystems zeigt

den Agenten, der in der Abbildung 6.4c dargestellt ist, modelliert und repräsentiert eine *automatisierte* Arbeitsstation, die keinen Werker benötigt. Daher besteht der Kontrollflussgraph dieses Agenten lediglich aus einem Puffer und einem Prozessbaustein zur Bearbeitung. Die Bearbeitungszeit dieser Arbeitsstation umfasst 30 bis 35 Sekunden. Beide Arbeitsstationen teilen das Vorhandensein einer Kontrollstrategie *InitWorkingStation*, die von der gleichnamigen Kontrollstrategie des Hauptagenten aufgerufen wird.

In diesem Experiment bilden die durchschnittliche Bearbeitungszeit eines Auftrages sowie die Auslastung die zu untersuchenden logistischen Kennzahlen. Die komponentenbasierte Synthese wird hierbei zur Suche einer Variante des Hauptagenten, welche die Kennzahlen bestmöglich erfüllt, verwendet. Das Feature-Diagramm in der Abbildung 6.5 zeigt die zu untersuchenden Variabilitätspunkte. Zunächst soll die Anzahl der Maschinen zwischen eins und fünf variiert werden, wobei jede mögliche Konstellation der Arbeitsstationen berücksichtigt wird. Dadurch ergeben sich insgesamt $2^1 + 2^2 + 2^3 + 2^4 + 2^5 = 62$ Varianten. Ferner soll die Anzahl der Auftragsquellen und -senken zwischen eins und fünf variiert werden. Der letzte Variabilitätspunkt betrifft die Vorverarbeitung eines Auftrages durch den Prozessbaustein *preAssemblyProcess*. Dieser Schritt soll im Kontrollflussgraphen des Hauptagenten wahlweise entfallen, sodass der Prozessbaustein *buffer* direkt mit dem Prozessbaustein *assemblingAreaStart* verbunden ist. Damit ergeben sich rechnerisch $62 \times 5 \times 5 \times 2 = 3100$ unterschiedliche Varianten. Nachfolgend werden die Komponentensammlungen vorgestellt, die aus der automatisierten Erstellung resultieren und im Rahmen der komponentenbasierten Synthese eingesetzt werden.

6.4.1 Aufbau der Komponentensammlung

Im ersten Schritt wird das Simulationsmodell eingelesen und gemäß dem Vorgehen aus dem Kapitel 6.1 teilautomatisiert in Komponenten überführt. Das Resultat bilden drei Kompo-

nentensammlungen, die jeweils zur Synthese eines Agenten verwendet werden. Jedoch wird nachfolgend ausschließlich die Komponentensammlung zur Synthese des Hauptagenten vorgestellt, da zum einen die übrigen Komponentensammlungen analog aufgebaut sind und zum anderen in diesem Experiment nicht variiert werden. Die Tabelle 6.3 listet die Komponenten, unterteilt in vier Gruppen, zur Synthese des Hauptagenten auf.

Die erste Gruppe umfasst die Komponente *buffer0*, welche die Produktion des gleichnamigen Prozessbausteins sowie der vorgelagerten Komposition von Prozessbausteinen verantwortet. Diese Komponente ist insofern besonders, als der produzierte Prozessbaustein *buffer0* über zwei eingehende Kanten im Kontrollflussgraphen verfügt. Daher umfasst die Argumentliste dieser Komponente über zwei Argumente, wobei jedes Argument zu einer Komponente korrespondiert, die einen entsprechenden Pfad produziert. Der native Typ dieser Komponente lautet `ComposedEmbeddedObject`.

Die zweite Gruppe umfasst die Komponenten, die einzelne Prozessbausteine produzieren. Hierfür werden Komponenten des Typs `EmbeddedObjectComponent` verwendet und der native Typ dieser Komponenten lautet `EmbeddedObject`.

Die dritte Gruppe umfasst Komponenten zur Produktion von Entscheidungsknoten und der zugehörigen Pfade. Im Gegensatz zur ersten Gruppe erfolgt bei diesen Komponenten die Parametrisierung des Entscheidungsknotens teilautomatisiert, basierend auf der Anzahl der Pfade. Die Komponenten sind Instanzen der Datenstruktur `SelectOutputComponent` und `SelectOutput` stellt den nativen Typen dar.

Die vierte Gruppe beinhaltet die Top-Level-Komponente, welche die Produktion des Kontrollflussgraphen verantwortet. Hierbei wird der vordefinierte Komponententyp `AgentComponent` verwendet. Dementsprechend umfasst die Argumentliste die Zieltypen der Komponenten, welche die einzelnen oder zusammengesetzten Prozessbausteine des Kontrollflussgraphen produzieren. Auch hier wird die Argumentliste in der Implementierung auf eine einzelne Liste reduziert, sodass die Top-Level-Komponente zur Laufzeit lediglich ein Argument erwartet. Der native Zieltyp dieser Komponente ist `Agent`.

6.4.2 Anpassungen an der Komponentensammlung

In diesem Abschnitt werden die Anpassungen an den Typspezifikationen der Komponenten beschrieben, die zur geforderten Synthese von Lösungsvarianten führt, die den geforderten Variabilitätspunkte entsprechen. Das Feature-Diagramm mit den Variabilitätspunkten dient als Grundlage und gibt in diesem Experiment die Reihenfolge der Umsetzung vor. Die Durchführung der Anpassungen verläuft für jeden Variabilitätspunkt analog. Zunächst werden die Komponenten, die in Zusammenhang mit dem aktuellen Variabilitätspunkt stehen, ermittelt und dann erfolgen die Modifikationen. Dies setzt jedoch voraus, dass die anwendende Person ein Grundverständnis bezüglich der Aufteilung des Simulationsmodells in die Komponenten verfügt. Die Anpassungen werden im webbasierten Frontend vorgenommen.

Komponentenname	Semantischer Typ
buffer0	InputPort1 ∩ Source → InputPort2 ∩ Source → Buffer0 ∩ Queue
inputPort1	InputPort1 ∩ Source
outputPort1	OutputPort1 ∩ Sink
preAssemblyProcess	PreAssemblyProcess ∩ Delay
assemblingAreaStart	AssemblingAreaStart ∩ RestrictedAreaStart
assemblingAreaEnd	AssemblingAreaEnd ∩ RestrictedAreaEnd
workingStation1	WorkingStation1 ∩ WorkingStation
workingStation2	WorkingStation2 ∩ WorkingStation
inputPort2	InputPort2 ∩ Source
outputPort2	OutputPort2 ∩ Sink
workerResourcePool	WorkerResourcePool ∩ ResourcePool
workingStation3	WorkingStation3 ∩ ManualWorkingStation
chooseMachine	WorkingStation3 ∩ ManualWorkingStation → WorkingStation2 ∩ WorkingStation → WorkingStation1 ∩ WorkingStation → ChooseMachine ∩ SelectOutput5
chooseOutputPort	OutputPort2 ∩ Sink → OutputPort1 ∩ Sink → ChooseOutputPort ∩ SelectOutput5
mainAgent	Buffer0 ∩ Queue → PreAssemblyProcess ∩ Delay → AssemblingAreaStart ∩ RestrictedAreaStart → ChooseMachine ∩ SelectOutput5 → AssemblingAreaEnd ∩ RestrictedAreaEnd → ChooseOutputPort ∩ SelectOutput5 → Main ∩ Agent

Tabelle 6.3: Komponentensammlung zur Synthese des Kontrollflussgraphen des Hauptagenten des Simulationsmodells des einfachen Produktionssystems.

6.4.2.1 Wahlweise Auslassung der Vorverarbeitung

Im ersten Schritt werden die Anpassungen beschrieben, die eine wahlweise Auslassung der Vorverarbeitung durch den Prozessbaustein *preAssemblyProcess* realisieren. Daher werden die Komponenten in der Komponentensammlung des Hauptagenten (vgl. Tabelle 6.3) ermittelt, die mit diesem Prozessbaustein zusammenhängen. Das Ergebnis dieser Suche umfasst die gleichnamige Komponente, die den Prozessbaustein produziert, sowie die Top-Level-Komponente. Letztere erwartet in ihrer Argumentliste die Komponente, die den Prozessbaustein *preAssemblyProcess* produziert. Da bei diesem Variabilitätspunkt das Vorhandensein des Prozessbausteins im Kontrollflussgraphen variiert werden soll und die Top-Level-Komponente die Komponente unmittelbar referenziert, wird die Top-Level-Komponente derart angepasst, dass zusätzlich Varianten synthetisiert werden, die den Prozessbaustein nicht enthalten.

Die Anpassung kann auf unterschiedlichen Arten erfolgen. Eine Möglichkeit stellt das Duplizieren der Top-Level-Komponente und das Auslassen des entsprechenden Arguments in der Typspezifikation der duplizierten Top-Level-Komponente dar. Dies führt zwar zur geforderten Variabilität, resultiert jedoch darin, dass in nachfolgenden Anpassungen stets beide Varianten der Top-Level-Komponente berücksichtigt werden müssen. Dadurch erhöht sich der Aufwand der nachkommenden Anpassungen und führt zu einer höheren Anzahl an Komponenten, die wiederum zu unübersichtlichen Komponentensammlungen führt. Stattdessen wird im Folgenden in der Argumentliste der Top-Level-Komponente nicht mehr der Zieltyp der Komponente, die den Prozessbaustein *preAssemblyProcess* produziert, erwartet, sondern der gemeinsame Zieltyp von zwei neuen Komponente, die den Prozessbaustein in einem Subprozess kapseln. Die erste Komponente verantwortet die Produktion eines Pfades mit dem Prozessbaustein, während die zweite Komponente einen leeren Pfad produziert und damit für eine Auslassung des Prozessbausteins sorgt. Mittels dieses Vorgehens existiert in der Komponentensammlung weiterhin eine Top-Level-Komponente.

Komponentenname	Semantischer Typ
sub_a	PreAssemblyProcess ∩ Delay → Sub1 ∩ SubProcess
sub_b	Sub1 ∩ SubProcess
mainAgent	Buffer0 ∩ Queue → Sub1 ∩ SubProcess → AssemblingAreaStart ∩ RestrictedAreaStart → ChooseMachine ∩ SelectOutput5 → AssemblingAreaEnd ∩ RestrictedAreaEnd → ChooseOutputPort ∩ SelectOutput5

Tabelle 6.4: Anpassungen an der Komponentensammlung zur Realisierung des wahlweisen Auslassens der Vorverarbeitung. Die Tabelle zeigt die zusätzlichen Komponenten sub1_a und sub_b sowie die geänderte Typspezifikation der Top-Level-Komponente mainAgent.

Die Tabelle 6.4 zeigt die Anpassungen gemäß der vorherigen Beschreibung. In dieser und den nachkommenden Tabellen werden ausschließlich die Komponenten aufgelistet, die modifi-

ziert oder ergänzt wurden. Jedoch sind die übrigen Komponenten weiterhin ein Bestandteil der Komponentensammlung. So sind die Komponenten sub_a und sub_b hinzugekommen, welche die Produktion beziehungsweise die Auslassung des Prozessbausteins zur Vorverarbeitung verantworten. Beide Komponenten verfügen über denselben semantischen Zieltyp Sub1 ∩ SubProcess und nativen Zieltyp `SubProcess`. Der gemeinsame Zieltyp ermöglicht, dass die Komponenten nach Außen nicht zu unterscheiden sind und zur geforderten Variantenbildung führen. Der Zieltyp der Subprozesse ersetzt das zweite Argument in der Typspezifikation der Top-Level-Komponente. Die Anpassung führt bei der Ausführung der komponentenbasierten Synthese zu zwei Lösungsvarianten.

6.4.2.2 Anzahl der Auftragsquellen und -senken

Im zweiten Schritt werden Anpassungen durchgeführt, die in einer Variation der Anzahl der Auftragsquellen und -senken resultiert. Zunächst erfolgt die Identifizierung der relevanten Komponenten. So werden die Auftragsquellen von den Komponenten *inputPort1* und *inputPort2* produziert, während die Komponenten *outputPort1* und *outputPort2* die Auftragssenken produzieren. Die Anzahl dieser Prozessbausteine soll laut des Feature-Diagramms zwischen eins und fünf variieren. Zur Realisierung dieses Variabilitätspunkts werden Duplikate der Prozessbausteine erzeugt, welche die einzelnen Auftragsquellen und -senken produzieren und sich hinsichtlich ihrer Parametrisierung unterscheiden. Anschließend werden die Komponenten, welche die Produktion der Auftragsquellen und -senken im Kontrollflussgraphen verantworten, dupliziert und angepasst, sodass jede geforderte Anzahl durch eine Komponente produziert wird.

Komponentenname	Semantischer Typ
inputPort3	InputPort3 ∩ Source
inputPort4	InputPort4 ∩ Source
inputPort5	InputPort5 ∩ Source
outputPort3	OutputPort3 ∩ Sink
outputPort4	OutputPort4 ∩ Sink
outputPort5	OutputPort5 ∩ Sink
buffer0$_a$	InputPort1 ∩ Source → Buffer0 ∩ Queue
buffer0$_b$	InputPort1 ∩ Source → InputPort2 ∩ Source → InputPort3 ∩ Source → Buffer0 ∩ Queue
buffer0$_c$	InputPort1 ∩ Source → InputPort2 ∩ Source → InputPort3 ∩ Source → InputPort4 ∩ Source → Buffer0 ∩ Queue

buffer0$_d$	InputPort1 ∩ Source → InputPort2 ∩ Source → InputPort3 ∩ Source → InputPort4 ∩ Source → InputPort5 ∩ Source → Buffer0 ∩ Queue
chooseOutputPort$_a$	OutputPort1 ∩ Sink → ChooseOutputPort ∩ SelectOutput5
chooseOutputPort$_b$	OutputPort1 ∩ Sink → OutputPort2 ∩ Sink → OutputPort3 ∩ Sink → ChooseOutputPort ∩ SelectOutput5
chooseOutputPort$_c$	OutputPort1 ∩ Sink → OutputPort2 ∩ Sink → OutputPort3 ∩ Sink → OutputPort4 ∩ Sink → ChooseOutputPort ∩ SelectOutput5
chooseOutputPort$_d$	OutputPort1 ∩ Sink → OutputPort2 ∩ Sink → OutputPort3 ∩ Sink → OutputPort4 ∩ Sink → OutputPort5 ∩ Sink → ChooseOutputPort ∩ SelectOutput5

Tabelle 6.5: Anpassungen an der Komponentensammlung zur Realisierung der unterschiedlichen Anzahlen an Auftragsquellen und -senken. Die erste Gruppe beinhaltet die Komponenten, die zusätzlichen Auftragsquellen und -senken produziert. Die zweite Gruppe umfasst die duplizierten Komponenten, die den Prozessbaustein *buffer0* sowie die Auftragsquellen produzieren. Die dritte Gruppe enthält die duplizierten Komponenten, die den Prozessbaustein *chooseOutputPort* und die Auftragssenken produzieren.

Die Tabelle 6.5 zeigt die vorgenommenen Anpassungen an der Komponentensammlung. Im ersten Schritt werden die Komponenten *inputPort3, inputPort4, inputPort5, outputPort3, outputPort4* und *outputPort5* hinzugefügt, welche die Produktion der zusätzlichen Auftragsquellen und -senken verantworten. Diese stellen Duplikate der Komponenten *inputPort1* und *outputPort1* dar, die hinsichtlich der Parametrisierung des Prozessbausteins sowie des Namens und semantischen Typs der Komponente angepasst wurden. Anschließend werden die Komponenten ermittelt, welche die Auftragsquellen und -senken im Kontrollflussgraphen produzieren. Im Fall der Auftragsquellen ist dies die Komponente *buffer0*, die in ihrer Argumentliste die Zieltypen der Komponenten, welche die Auftragsquellen produzieren, erwartet. Nachfolgend soll die Komponente *buffer0* unterschiedliche Konfigurationen produzieren, die eine bis fünf Auftragsquellen umfassen soll. Da die Ausgangskonfiguration, die zwei Auftragsquellen vorsieht, bereits durch die Komponente *buffer0* produziert wird, werden vier Duplikate dieser Komponente erzeugt, welche die verbleibenden Konfigurationen abdecken. Jedes Duplikat wird anschließend hinsichtlich der Typspezifikation angepasst, sodass diese in der Produktion einer entsprechenden Anzahl an Auftragsquellen resultiert. Auch in diesem Fall wird der semantische Zieltyp der Duplikate nicht verändert, um eine Bildung von Varianten zu erreichen. Die Komponenten buffer0$_a$ bis buffer0$_d$ stellen die duplizierten und angepassten Komponenten zur Produktion des Prozessbausteins *buffer0* sowie den unterschiedlichen Anzahlen der Auftragsquellen dar. Dabei ist den Argumentlisten des semantischen Typs zu entnehmen, dass jede Komponente eine unterschiedliche Anzahl an Argumenten umfasst, die der Anzahl der zu produzierenden Auftragsquellen entspricht. Die Anpassung wird analog für

die Bildung der Varianten hinsichtlich der Auftragssenken durchgeführt. Hierbei verantwortet die Komponente *chooseOutputPort* die Produktion der Auftragssenken und wird aufgrund dessen dupliziert und angepasst. Die Komponenten chooseOutputPort$_a$ bis chooseOutputPort$_d$ stellen die angepassten Duplikate dar. Die beschriebenen Anpassungen an der Komponentensammlung führen zur Bildung von 25 weiteren Varianten bei der Durchführung der komponentenbasierten Synthese.

6.4.2.3 Anzahl der Arbeitsstationen

Im dritten Schritt wird die Anzahl der Arbeitsstationen im Kontrollflussgraphen des Hauptagenten variiert. Gemäß dem Feature-Diagramm aus der Abbildung 6.5 soll die Anzahl zwischen eins und fünf und die Art der Arbeitsstationen (manuell oder autonom) variieren.

Hierfür wird zunächst die freie Wahl einer Arbeitstation realisiert, in dem die Komponenten zur Produktion der Arbeitstationen ermittelt und hinsichtlich der Typspezifikation angepasst werden. Im ursprünglichen Kontrollflussgraphen sind drei Arbeitstationen vorhanden, wovon zwei die Aufträge autonom verarbeiten. Angesichts dessen sind in der automatisiert erstellten Komponentensammlung drei Komponenten vorhanden, die jeweils eine Arbeitstation produzieren. Nachfolgend soll in der Komponentensammlung jeweils eine Komponente für genau eine Art einer Arbeitsstation vorhanden sein. Die Prozessbausteine *workingStation1* und *workingStation2* können daher zusammengefasst werden. Dies ist insbesondere möglich, da diese über eine identische Konfiguration im Simulationsmodell verfügen. Daher wird eine der Komponenten aus der Komponentensammlung entfernt und die verbleibende Komponente umbenannt. So wird die Komponente *workingStation1* zu *automatedWorkingStation* umbenannt und die Komponente *workingStation3* wird nachfolgend als *manualWorkingStation* bezeichnet. In diesem Zusammenhang werden auch die semantischen Typen dieser Komponenten angepasst, sodass diese kongruent zum Komponentennamen sind. Ferner werden die semantischen Typen um einen gemeinsamen Zieltypen *WorkingStation* mittels einer Intersektion erweitert. Damit kann eine Komponente zur Produktion einer Arbeitstation gefordert werden, ohne die Art der Arbeitstation explizit zu spezifizieren.

Nach der Anpassung der Typspezifikationen dieser Komponenten werden die semantischen Typen aller Komponenten überarbeitet, die in ihrer Typspezifikation die zuvor angepassten Typen erwarten. Diese müssen derart angepasst werden, sodass sie den modifizierten semantischen Typen entsprechen. Dies trifft auf die Komponente *chooseMachine* zu, welche die Zieltypen der Komponenten zur Produktion der Arbeitsstationen als Argumente in ihrem semantischen Typen erwartet. Damit wird der semantische Typ wie folgt angepasst:

$$\text{WorkingStation} \cap \text{Manual} \rightarrow \text{WorkingStation} \cap \text{Manual} \rightarrow \text{WorkingStation} \cap \text{Automated} \rightarrow \text{ChooseMachine} \cap \text{SelectOutput}$$

Diese Typspezifikation entspricht der ursprünglichen Konfiguration des Kontrollflussgraphen

des Hauptagenten. Im nächsten Schritt erfolgt eine Unterspezifizierung der Argumente, sodass nicht explizit spezifiziert wird, welche Art einer Arbeitsstation als Argument erwartet wird. Damit wird spezifiziert, dass eine beliebige Arbeitsstation gefordert wird und zur Syntheselaufzeit werden diese automatisiert durch entsprechende Arbeitstationen ersetzt. Nach der Anpassung der Typspezifikation der Komponente *chooseMachine*, wird diese analog zu den vorherigen Anpassungen dupliziert. Danach wird jedes Duplikat hinsichtlich des semantischen Typs angepasst, sodass die geforderte Anzahl an Arbeitsstationen produziert wird.

Komponentenname	Semantischer Typ
manualWorkingStation	WorkingStation ∩ Manual
automatedWorkingStation	WorkingStation ∩ Automated
chooseMachine_a	WorkingStation → ChooseMachine ∩ SelectOutput5
chooseMachine_b	WorkingStation → WorkingStation → ChooseMachine ∩ SelectOutput5
chooseMachine_c	WorkingStation → WorkingStation → WorkingStation → ChooseMachine ∩ SelectOutput5
chooseMachine_d	WorkingStation → WorkingStation → WorkingStation → WorkingStation → ChooseMachine ∩ SelectOutput5
chooseMachine_e	WorkingStation → WorkingStation → WorkingStation → WorkingStation → WorkingStation → ChooseMachine ∩ SelectOutput5

Tabelle 6.6: Anpassungen an der Komponentensammlung zur Realisierung der Generalisierung der Maschinen sowie zur Variantenbildung hinsichtlich der Anzahl der Maschinen.

Die Tabelle 6.6 zeigt die resultierende Komponentensammlung. Die ersten beiden Zeilen beinhalten die Komponenten zur Produktion der Prozessbausteine der Arbeitstationen. Die verbleibenden Zeilen zeigen die duplizierten und angepassten Komponenten zur Produktion des Entscheidungsknoten *chooseMachine* inklusive der Arbeitsstationen. Mittels der durchgeführten Anpassung werden bei der Ausführung der komponentenbasierten Synthese weitere 62 Varianten des Kontrollflussgraphen des Hauptagenten synthetisiert.

6.4.3 Komponentenbasierte Synthese, Simulation und Auswertung

Im dritten Schritt folgt die Zusammensetzung der Kontrollflussgraphen und Kontrollstrategien in ausführbare Simulationsmodelle. Da die Kontrollstrategie *InitWorkingStations* bereits in Kapitel 6.1.2 als beispielhaft in eine Produktlinie migriert wurde, wird diese im Nachfolgenden nicht erneut vorgestellt. Gemäß dem Feature-Diagramm aus der Abbildung 6.5 ergeben sich 3100 Varianten des Kontrollflussgraphen des Hauptagenten. Da für jede Variante des Agenten in diesem Experiment genau eine Kontrollstrategie *InitWorkingStations* synthetisiert wird, erhöht sich die Anzahl der Varianten nicht. Ebenso werden keine Varianten

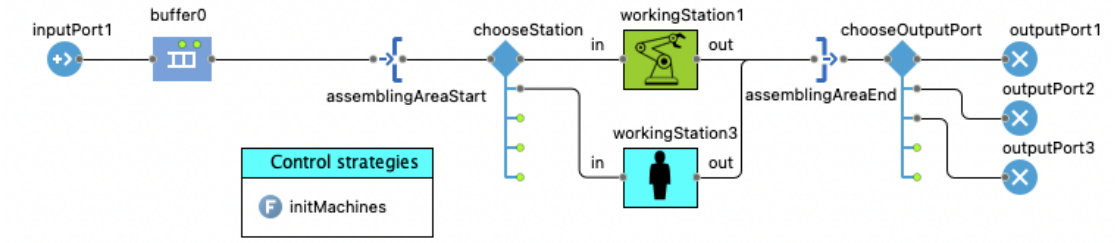
Komponentenname	Semantischer Typ
mainAgent_1	Main ∩ Agent
mainAgent_2	Main ∩ Agent
⋮	⋮
mainAgent_{3100}	Main ∩ Agent
manualWorkingStationAgent	ManualWorkingStation ∩ Agent
automatedWorkingStationAgent	AutomatedWorkingStation ∩ Agent
simulationModel	Main ∩ Agent → ManualWorkingStation ∩ Agent → AutomatedWorkingStation ∩ Agent → SimulationModel

Tabelle 6.7: Automatisiert erzeugte Komponentensammlung mit Komponenten, welche die Produktion zuvor synthetisierter Agenten verantworten. Die Top-Level-Komponente *simulationModel* produziert eine Komposition der Agenten in Form eines ausführbaren Simulationsmodells.

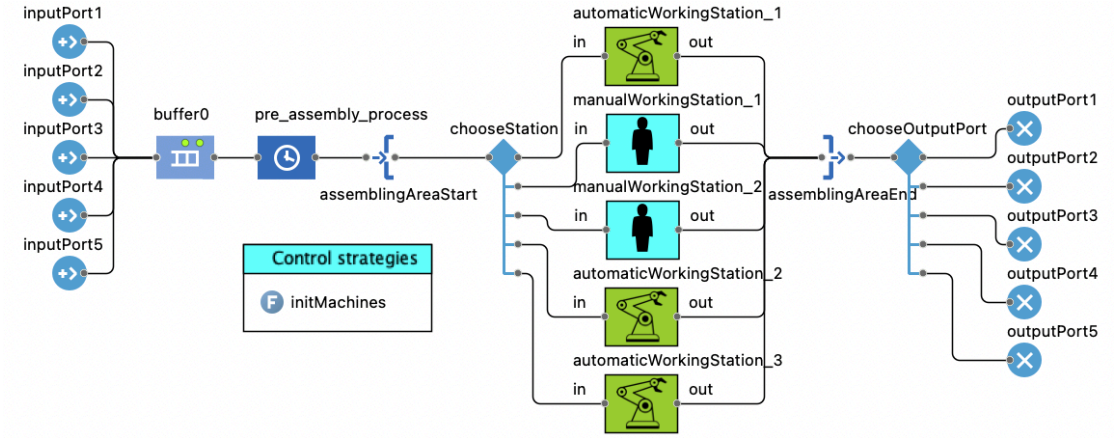
der Kontrollflussgraphen der anderen Agenten des Simulationsmodells synthetisiert. Somit umfasst die Komponentensammlung zur Synthese der ausführbaren Simulationsmodelle $3100 + 2 + 1$ Komponenten, wobei 3100 für die Produktion einer Zusammensetzung aus einem Kontrollflussgraphen und einer Kontrollstrategie verantwortlich sind. Die verbleibenden Komponenten verantworten die Produktion der Agenten zur Repräsentation einer Arbeitsstation sowie zur Komposition der Agenten und anschließender Übersetzung in ein proprietäres Format einer Simulationsumgebung.

Die resultierende Komponentensammlung ist in der Tabelle 6.7 dargestellt. Die Komponenten *manualWorkingStationAgent* und *AutomatedWorkingStationAgent* sind für die Produktion der Agenten zuständig, welche die manuelle (vgl. Abbildung 6.4b) und autonome Arbeitsstation (vgl. Abbildung 6.4c) repräsentieren. Beide Komponenten sind in der Komponentensammlung genau einmal vorhanden, da jeweils eine Variante synthetisiert wurde. Ferner entsprechen diese Varianten der ursprünglichen Modellierung. Die Komponenten $\textit{mainAgent}_1$ bis $\textit{mainAgent}_{3100}$ entsprechen den Varianten des Hauptagenten. Die Top-Level-Komponente *simulationModel* produziert das ausführbare Simulationsmodell und erwartet in der Argumentliste ihrer Typspezifikation die Komponenten, die einen Agenten produzieren.

Im Folgenden wird die Durchführung der Simulation basierend auf den synthetisierten Simulationsmodellen beschrieben. Die Auswertung der Simulationsläufe ermöglicht den Vergleich einzelner Varianten mittels logistischer Kennzahlen, die zur Simulationslaufzeit ermittelt werden. Zur Ausführung der Simulationsmodelle werden die Simulationsmodelle in die Simulationsumgebung AnyLogic 8 geladen und die Simulation gestartet. Das exemplarische Simulationsmodell schreibt nach der Beendigung eines Simulationslaufs die Kennzahlen automatisiert in eine Datenbank. Diese Datenbank enthält Einträge mit Informationen über die aktuelle Konfiguration (Anzahl der Auftragsquellen, -senken, und Maschinen sowie Maschinentypen) sowie den im Rahmen der Simulation ermittelten Kennzahlen.



(a) Erste Variante des Hauptagenten, die eine Auftragsquelle, drei Auftragssenken, eine manuelle Maschine sowie eine autonome Maschine umfasst.



(b) Zweite Variante des Hauptagenten, die fünf Auftragsquellen und -senken, eine Vorverarbeitung, zwei manuelle Maschinen sowie drei autonome Maschinen umfasst.

Abbildung 6.6: Synthetisierte Kontrollflussgraphen, die aus der Migration des Simulationsmodells des einfachen Produktionssystems resultieren.

Nachfolgend werden zwei exemplarische Varianten des Simulationsmodells betrachtet. Die Abbildung 6.6 zeigt die Kontrollflussgraphen der Hauptagenten der synthetisierten Simulationsmodelle, die sich hinsichtlich ihrer Struktur unterscheiden. An dieser Stelle sei anzumerken, dass die Visualisierung der Verbindungen zwischen den Prozessbausteinen für die Verschriftlichung der vorliegenden Dissertation händisch angepasst werden, um die Verbindungen deutlicher sichtbar zu machen. In den synthetisierten Simulationsmodellen überschneiden sich die Verbindungen zum Teil. Letzteres beeinflusst jedoch nicht die Funktionsfähigkeit des Simulationsmodells, sondern wirkt sich lediglich auf Übersichtlichkeit aus. Daher sollte in zukünftigen Forschungsarbeiten das Layout der visuellen Darstellung eines Simulationsmodells ebenfalls automatisiert angepasst werden. Das erste synthetisierte Simulationsmodell (vgl. Abbildung 6.6a) beinhaltet eine Auftragsquelle, drei Auftragssenken, eine manuelle sowie zwei autonome Arbeitsstationen. Die zweite Variante (vgl. Abbildung 6.6b) verfügt über jeweils fünf Auftragsquellen und -senken, eine Vorverarbeitung, zwei manuelle sowie drei autonome Arbeitsstationen. Beide Varianten des Produktionssystems werden mit einem Auftragsportfolio mit 100 Aufträgen, die in einem Abstand von zwei Sekunden generiert werden, simuliert.

Anzahl Auftrags-quellen	Anzahl Maschinen	Anzahl Auftrags-senken	Vorverarbeitung vorhanden	Durchschnittl. Durchlaufzeit	Durchschnittl. Maschinen-auslastung
2	3	2	ja	16 Sek.	77 %
1	2	3	nein	6 Sek	82 %
5	5	5	ja	35 Sek.	55 %

Tabelle 6.8: Ergebnisse der Simulationsläufe ausgewählter Varianten im Anwendungsfall des fiktiven Produktionssystems.

Die Tabelle 6.8 zeigt die Ergebnisse der Ausführung des ursprünglichen Simulationsmodells sowie von zwei ausgewählten Varianten. Die Durchführung der Simulation zeigt, dass im ursprünglichen Simulationsmodell die durchschnittliche Durchlaufzeit 26 Sekunden beträgt. In der ersten Variante beträgt die durchschnittliche Durchlaufzeit 6 Sekunden pro Auftrag, während die zweite Variante im Schnitt 35 Sekunden pro Auftrag benötigt. Die kürzere Durchlaufzeit bei den Varianten mit weniger Auftragsquellen ist darauf zurückzuführen, dass die Erzeugung der Aufträge in diesen Konfigurationen in größeren Abständen erfolgt. Dadurch stauen sich die Aufträge nicht im Puffer *buffer0*. Andererseits wird eine größere Zeitspanne zur Abarbeitung sämtlicher Aufträge benötigt. Während die Bearbeitung der 100 Aufträge in der ursprünglichen Variante 117 Sekunden benötigt, sind es in der ersten Varianten 182 Sekunden und in der zweiten Variante 109 Sekunden. Die Auslastung der Maschinen beträgt in der ursprünglichen Variante des Simulationsmodells 77 %, in der ersten Variante 82 % und in der zweiten Variante 55 %. Die geringe Auslastung in der zweiten Variante ist auf die geringe Auslastung der Arbeitsstation *automaticWorkingStation_2* zurückzuführen, die lediglich 10 % beträgt. Dies wiederum ist auf die Terminplanung in diesem Beispiel zurückzuführen, welche die Aufträge im Entscheidungsknoten *chooseStation* auf die nächste freie Arbeitsstation gemessen an der vertikalen Anordnung weiterleitet. Damit zeigt sich in diesem einfachen Beispiel, dass je nach Kennziffer eine andere Variante zu wählen ist. An dieser Stelle sei erneut darauf hinzuweisen, dass dieses Experiment keineswegs den Anspruch einer Studie zur Untersuchung logistischer Systeme anstrebt, sondern die Einsatzfähigkeit des Vorgehens demonstrieren möchte.

6.5 Durchführung von Experimenten: Bodenblocklager

Im zweiten Experiment dieses Anwendungsfalls wird ein Simulationsmodell eines Bodenblocklagersystems in eine Produktlinie migriert. Ein Bodenblocklager stellt eine Ausprägung eines Lagersystems dar, das sich unter anderem dadurch auszeichnet, dass es keine Infrastruktur voraussetzt. In solchen Lagersystemen werden die Güter auf Paletten auf dem Boden platziert, wobei diese übereinander gestapelt werden können. Die räumliche Verteilung der Güter ist frei wählbar. Durch den Einsatz von (fahrerlosen) Transportsystemen werden die

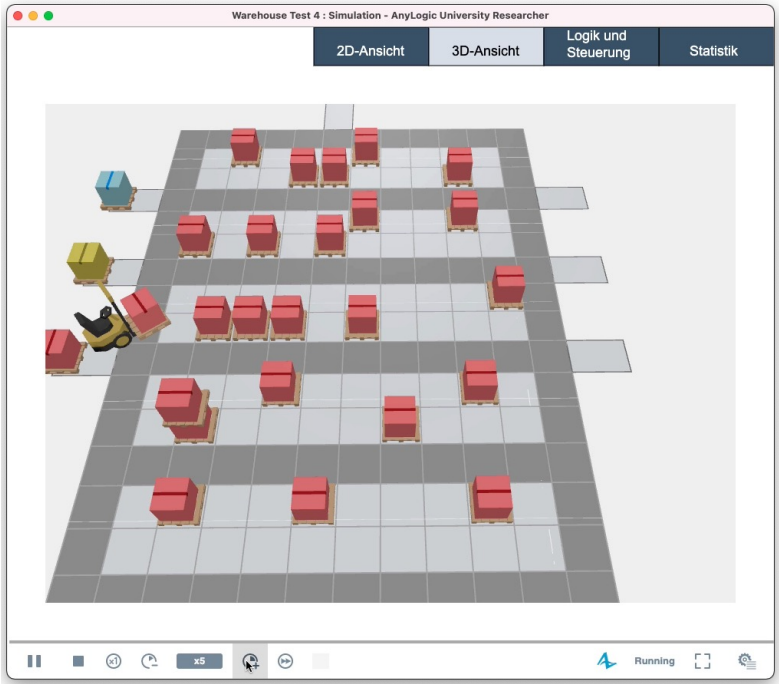


Abbildung 6.7: 3D-Visualisierung des Simulationsmodells, das ein Bodenblocklagersystem repräsentiert.

Güter im Lager ein-, um- und ausgelagert. In der Abbildung 6.7 ist die 3D-Visualisierung eines Simulationsmodells eines Bodenblocklagersystems dargestellt. In dieser Visualisierung stellen Gabelstapler das Transportsystem dar und das Bodenblocklager ist in Quadrate aufgeteilt, wobei die hellgrauen Blöcke die Stellflächen, die dunkelgrauen Blöcke die Transportwege und die äußeren, hellgrauen Blöcke die Wareneingänge und -ausgänge markieren.

Bodenblocklagersysteme stellen im Kontext der Logistik eine gänge Variante eines Lagersystems dar und sind dementsprechend auch Gegenstand aktueller Forschungsarbeiten. Dieses Experiment wurde im Rahmen einer Zusammenarbeit mit Jakob Pfrommer innerhalb des Graduiertenkollegs 2193 durchgeführt. Das Forschungsprojekt von Jakob Pfrommer im Kontext des GRKs 2193 umfasst die wissenschaftliche Untersuchung von zentralen Herausforderungen und Lösungsansätzen in Bezug auf autonom organisierte Bodenblocklager. In [92] geben Pfrommer und Meyer in einer systematischen Literaturrecherche einen Überblick über relevante Teilprobleme in diesem Forschungsgebiet. Auch in der Forschung zu Bodenblocklagersystemen stellt sich die Simulation als ein gängiges Werkzeug dar. So verwenden Derhami et al. in [30] ein Simulationsmodell, um ein optimales Layout für ein Bodenblocklager zu ermitteln. Vieira et al. nutzen in [121] die Simulation zur Evaluation unterschiedlicher Einlagerungsstrategien in einem Simulationsmodell. Auch Jakob Pfrommer nutzt im Rahmen seiner Forschungsarbeit die Simulation. So entstand das nachfolgend verwendete Simulationsmodell im Kontext seiner Forschungsarbeit. Durch die Zusammenarbeit konnte im Rahmen dieses Dissertationsprojekts die Migration anhand eines praxisnahen Simulationsmodells

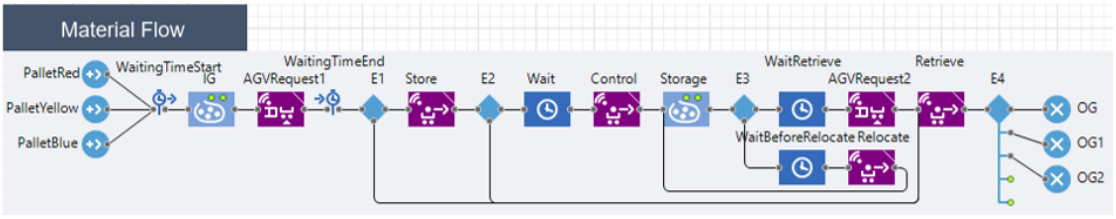


Abbildung 6.8: Kontrollflussgraphen des Hauptagenten im Simulationsmodell, das die Logik eines Bodenblocklagersystems umfasst.

demonstriert werden. Gleichzeitig konnte im Kontext der Untersuchung von Fragestellungen in Zusammenhang mit Bodenblocklagersystemen eine größere Anzahl an Konfigurationen untersucht werden, deren händische Modellierung einen hohen Zeitaufwand verursacht hätte.

Nachfolgend wird die Logik des Simulationsmodells zur Repräsentation von Bodenblocklagern vorgestellt. Auch dieses Simulationsmodell wurde in der Simulationsumgebung AnyLogic 8 entwickelt. Die Abbildung 6.8 zeigt den Kontrollflussgraphen des Hauptagenten. Die Prozessbausteine *PalletRed, PalletYellow* und *PalletBlue* repräsentieren die Wareneingänge im Bodenblocklager, welche die Paletten mit den einzulagernden Gütern erzeugen. Jeder Wareneingang oder genauer gesagt der korrespondierende Prozessbaustein erzeugt unterschiedliche Paletten, die sich hinsichtlich der Farbe der Palette unterscheiden. So erzeugt der Prozessbaustein *PalletRed* rote, *PalletYellow* gelbe und *PalletBlue* blaue Paletten. Nach der Erzeugung einer Palette wird mittels des Prozessbausteins *AGVRequest1* ein Transportmittel zum Transport der Palette angefordert. Im Entscheidungsknoten *E1* wird anschließend geprüft, ob die Palette eingelagert (rechter Pfad ausgehend von *E1*) oder sofort ausgelagert werden soll (unterer Pfad ausgehend von *E1*). Sofern eine Palette eingelagert wird, kann im weiteren Verlauf eine Umlagerung dieser erfolgen. Ob eine Umlagerung erfolgt, wird im Entscheidungsknoten *E3* bestimmt. Der untere Pfad bildet das Umlagern im Kontrollflussgraphen ab und der rechte Pfad die Auslagerung einer Palette aus dem Bodenblocklager. Die auszulagernde Palette wird an eine der Warenausgänge (Prozessbausteine *OG, OG1* oder *OG2*) transportiert. An diesen Warenausgängen wird die Palette aus dem Bodenblocklager entfernt. Hierbei ist es für die Warenausgänge unerheblich, welche Farbe eine Palette hat.

In der Modellierung des Simulationsmodells sind weitere Agenten enthalten sowie eine Reihe von Kontrollstrategien. Letztere erweitern insbesondere die Logik der Prozessbausteine und bilden verschiedene Prozesse des Bodenblocklagers wie Einlagerungsstrategien ab. Hierbei werden folgende Einlagerungsstrategien unterstützt: Closest Open Location (COL), ABC-Einlagerung (Aufteilung des Lagers in umsatzbasierte Zonen) oder eine zufallsbasierte Einlagerung. Das grundlegende Layout des Bodenblocklagers, das sich durch die Anzahl und Positionierung der Stellflächen und Transportwege charakterisiert, wird durch ein Skript dynamisch generiert. Dadurch wird eine Variation hinsichtlich der Anzahl der inneren Blöcke unabhängig von der komponentenbasierten Synthese realisiert. Die äußeren Blöcke für die Wareneingänge und -ausgänge werden nachfolgend mittels der Synthese hinzugefügt.

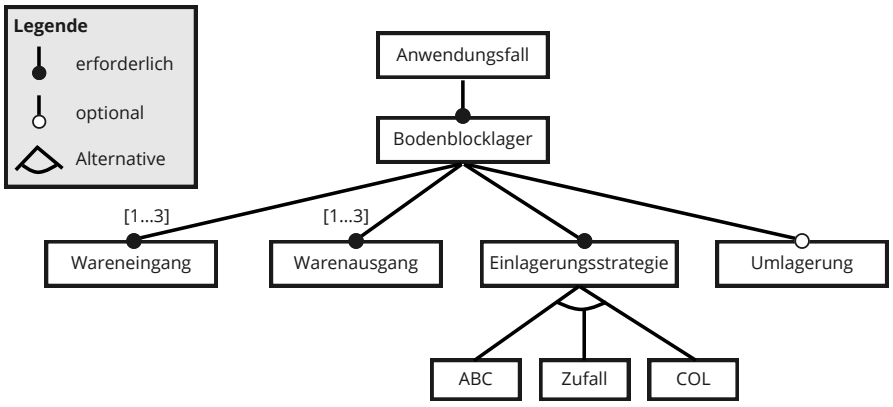


Abbildung 6.9: Feature-Diagramm, das die Variabilitätspunkte des Bodenblocklagersystems zeigt.

Im Rahmen dieses Experiments werden ausgewählte Variabilitätspunkte untersucht, die im Feature-Diagramm in der Abbildung 6.9 zusammengefasst sind. Jedoch stellen diese eine Auswahl der möglichen Variabilitätspunkte in einem Bodenblocklager dar. Durch die Realisierung einer beschränkten Auswahl an Variabilitätspunkte konnte eine Migration des Simulationsmodells innerhalb kürzester Zeit vorgenommen werden. Nachfolgend werden die einzelnen Variabilitätspunkte vorgestellt. Der erste Variabilitätspunkt bezieht sich auf die Anzahl der Wareneingänge im Bodenblocklager, die zwischen eins und drei variiert werden soll. Ebenso trifft dies auf den zweiten Variabilitätspunkt zu, der eine Variation der Anzahl der Warenausgänge vorsieht. Der dritte Variabilitätspunkt bezieht sich auf die Auswahl einer Einlagerungsstrategie. Dieser Variabilitätspunkt beeinflusst die Parametrisierung des Simulationsmodells, die durch eine Kontrollstrategie vorgenommen wird. Der vierte Variabilitätspunkt bezieht sich auf das Vorhandensein der Umlagerung im Bodenblocklager, die wahlweise vorhanden sein soll. Insgesamt ergeben sich somit in diesem Experiment $3 \times 3 \times 3 \times 2 = 54$ unterschiedliche Varianten.

6.5.1 Aufbau der Komponentensammlung

Auch in diesem Experiment wird im ersten Schritt das bereits existierende Simulationsmodell automatisiert in eine Komponentensammlung zur Synthese der Kontrollflussgraphen überführt. Ebenfalls wird in diesem Experiment ausschließlich der Kontrollflussgraph des Hauptagenten variiert, sodass nachfolgend lediglich die Komponentensammlung dieses Agenten vorgestellt wird. Die Tabelle 6.10 listet die Komponenten dieser Komponentensammlung auf.

Wie auch im ersten Experiment können die Komponenten gruppiert werden. Die Gruppierungen sind in der Tabelle durch horizontale Trennlinien separiert. Die erste Gruppe umfasst Komponenten *sub10* bis *sub14,* die Kompositionen von Prozessbausteinen produzieren.

Komponentenname	Semantischer Typ
sub10	Retrieve ∩ MoveByTransporter → E4 ∩ SelectOutput5 → Sub10 ∩ SubProcess
sub11	Store ∩ MoveByTransporter → E2 ∩ SelectOutput → Sub11 ∩ SubProcess
sub12	Wait ∩ Delay → Control ∩ MoveByTransporter → Storage ∩ Wait → E3 ∩ SelectOutput → Sub12 ∩ SubProcess
sub13	WaitBeforeRelocate ∩ Delay → Retrieve ∩ MoveByTransporter → Storage ∩ Wait → Sub13 ∩ SubProcess
sub14	WaitRetrieve ∩ Delay → AGVRequest2 ∩ SeizeTransporter → Retrieve ∩ MoveByTransporter → Sub14 ∩ SubProcess
waitingTimeStart	PalletYellow ∩ Source → PalletRed ∩ Source → PalletBlue ∩ Source → WaitingTimeStart ∩ TimeMeasureStart
transporterFleet	TransporterFleet ∩ TransporterFleet
transporterControl	TransporterControl ∩ TransporterControl
palletRed	PalletRed ∩ Source
og	OG ∩ Sink
store	Store ∩ MoveByTransporter
retrieve	Retrieve ∩ MoveByTransporter
storage	Storage ∩ Wait
palletYellow	PalletYellow ∩ Source
palletBlue	PalletBlue ∩ Source
relocate	Relocate ∩ MoveByTransporter
waitRetrieve	WaitRetrieve ∩ Delay
waitBeforeRelocate	WaitBeforeRelocate ∩ Delay
control	Control ∩ MoveByTransporter
agvRequest1	AGVRequest1 ∩ SeizeTransporter
agvRequest2	AGVRequest2 ∩ SeizeTransporter
ig	IG ∩ Wait
waitingTimeEnd	WaitingTimeEnd ∩ TimeMeasureEnd
wait	Wait ∩ Delay
og1	OG1 ∩ Sink
og2	OG2 ∩ Sink
e1	Sub11 ∩ SubProcess → Sub10 ∩ SubProcess → E1 ∩ SelectOutput
e2	Sub12 ∩ SubProcess → Retrieve ∩ MoveByTransporter → E2 ∩ SelectOutput
e3	Sub14 ∩ SubProcess → Sub13 ∩ SubProcess → E3 ∩ SelectOutput
e4	OG ∩ Sink → OG1 ∩ Sink → OG2 ∩ Sink → E4 ∩ SelectOutput5
mainAgent	WaitingTimeStart ∩ TimeMeasureStart → IG ∩ Wait → AGVRequest1 ∩ SeizeTransporter → WaitingTimeEnd ∩ TimeMeasureEnd → E1 ∩ SelectOutput → MainAgent ∩ Agent

Tabelle 6.9: Komponentensammlung zur Synthese von Kontrollflussgraphen des Hauptagenten des Bodenblocklager-Simulationsmodells.

Die zweite Gruppe umfasst die Komponente *waitingTimeStart*, die den gleichnamigen Prozessbaustein sowie die vorgelagerten Prozessbausteine der Wareneingänge produziert. Die dritte Gruppe umfasst die Komponenten *transporterFleet* bis *og2*, die jeweils die Produktion eines einzelnen Prozessbausteins verantworten. Die vierte Gruppe umfasst Komponenten, die Entscheidungsknoten sowie die entsprechenden Subprozesse produzieren. Die fünfte Gruppe umfasst die Top-Level-Komponente, die den Kontrollflussgraphen des Hauptagenten produziert.

Die Typisierung dieser Komponentensammlung erfolgte automatisiert, basierend auf dem Vorgehen aus dem Kapitel 4.4.2. Jedoch weist die Typisierung dieser Komponentensammlung zwei Besonderheiten auf. Zum einen umfasst die Argumentliste der Top-Level-Komponente nur einen Teil der Typen, welche die Prozessbausteine gemäß der Reihenfolge im ursprünglichen Kontrollflussgraphen produzieren. Ferner bildet *e1* das letzte Argument der Typspezifikation der Top-Level-Komponente, auch wenn der vom Argument repräsentierte Prozessbaustein nicht das letzte Element im Kontrollflussgraphen darstellt. Dies könnte zur Annahme führen, dass nicht alle erforderlichen Prozessbausteine synthetisiert werden. Jedoch trifft dies nicht zu, da die fehlenden Prozessbausteine durch die Komponente *e1* abgedeckt sind. Diese sind auf unterschiedliche Komponenten verteilt, die im Rahmen der komponentenbasierten Synthese produziert werden. So erwartet die Komponente *e1* zwei Argumente des Typs Sub10 ∩ SubProcess und Sub11 ∩ SubProcess. Letztere sind für die Produktion der ausgehenden Pfade im Kontrollflussgraphen verantwortlich. So produziert die Komponente mit dem semantischen Typen Sub10 ∩ SubProcess den unteren Pfad ausgehend vom Prozessbaustein *E1*. Dementsprechend produziert das andere Argument den rechten Pfad. Bei der Betrachtung der Typspezifikation dieser Komponenten, welche die Pfade produzieren, fällt auf, dass diese jeweils eine Komponente erwarten, die einen weiteren Entscheidungsknoten produziert. Die entsprechenden Komponenten, die diese Prozessbausteine produzieren, erwarten wiederum ebenfalls Komponenten, die Subprozesse produzieren. Die Komponente *sub10* erwartet ein Argument, das in der Produktion des Prozessbausteins *E4* resultiert. Die Komponente, die den Prozessbaustein *E4* produziert, erwartet ebenfalls Komponenten, die Subprozesse produzieren. Dies setzt sich so lange fort, bis Komponenten erreicht werden, die nicht weiter verschachtelt sind. In der Komponentensammlung trifft dies auf die Komponente *e4* zu.

Die zweite Eigenheit betrifft die Schleife im Kontrollflussgraphen, die vom Prozessbaustein *E3* ausgeht und das Umlagern von Paletten repräsentiert. Die ausgehenden Pfade des Prozessbausteins *E3* werden durch die Komponenten *sub13* und *sub14* produziert. Hierbei verantwortet die Komponente *sub13* die Produktion der Umlagerung. In der Typspezifikation dieser Komponente ist in der Argumentliste ein Argument vorhanden, das in der Produktion des Prozessbausteins *Storage* resultiert. Gleichzeitig wird der Prozessbaustein bereits durch die Komponente *e2* produziert, sodass dieser Baustein doppelt produziert wird. In den Implementierungsdetails der Komponente, die den Kontrollflussgraphen zusammensetzt, ist hierzu eine Logik implementiert, die Duplikate eines Prozessbausteins ermittelt und herausfiltert.

Komponentenname	Semantischer Typ
waitingTimeStart$_a$	PalletYellow ∩ Source → WaitingTimeStart ∩ TimeMeasureStart
waitingTimeStart$_b$	PalletYellow ∩ Source → PalletRed ∩ Source → WaitingTimeStart ∩ TimeMeasureStart
e4$_a$	OG ∩ Sink → E4 ∩ SelectOutput5
e4$_b$	OG ∩ Sink → OG1 ∩ Sink → E4 ∩ SelectOutput5

Tabelle 6.10: Anpassungen an der Komponentensammlung zur Realisierung der Variation der Anzahl der Wareneinein- und ausgänge.

6.5.2 Anpassungen an der Komponentensammlung

Nachfolgend werden die notwendigen Anpassungen beschrieben, welche die geforderte Variabilität gemäß dem Feature-Diagramm aus der Abbildung 6.9 realisieren.

6.5.2.1 Anzahl der Ein- und Ausgänge

Im ersten Schritt wird die Anzahl der Wareneingänge und -ausgänge variiert. Auch in diesem Experiment werden zunächst die Komponenten ermittelt, die im Kontext der entsprechenden Prozessbausteine stehen. Hierbei ist festzustellen, dass die Prozessbausteine *PalletRed, PalletBlue, PalletYellow* zur Repräsentation der Wareneingänge sowie die Prozessbausteine *OG, OG1, OG2* zur Repräsentation der Warenausgänge durch gleichnamige Komponenten produziert werden. Die Komponente *waitingTimeStart* verantwortet die Produktion der Wareneingänge im Kontrollflussgraphen, während die Komponente *e4* für die Produktion der Warenausgänge verantwortlich ist. Zur Erreichung der geforderten Varianten werden beide Komponenten dupliziert und die Typspezifikation derart angepasst, sodass jeweils eine Komponente zur Produktion der entsprechenden Anzahl der Wareneingänge und -ausgänge vorhanden ist. Der Zieltyp der jeweiligen Komponenten bleibt unverändert. Ferner erfolgt in der Implementierung eine automatisierte Anpassung der Parametrisierung des Entscheidungsknoten *E4* basierend auf der Anzahl der Warenausgänge. Sofern etwa eine Lösungsvariante mit einem einzelnen Warenausgang synthetisiert wird, wird in der Parametrisierung die Wahrscheinlichkeit für die Auswahl des Warenausgangs auf 100 % gesetzt. Die Tabelle 6.10 zeigt die Komponenten, die der Komponentensammlung hinzugefügt werden.

Die Komponente *waitingTimeStart$_a$* produziert eine Konfiguration mit einem Wareneingang, während die Komponente *waitingTimeStart$_b$* eine Konfiguration mit zwei Wareneingängen produziert. Die Komponente *waitingTimeStart* aus der ursprünglichen Komponentensammlung ist weiterhin vorhanden und produziert die Ausgangskonfiguration mit drei Wareneingängen. Ebenso gilt dies für die Komponenten, welche die Produktion des Entscheidungsknoten *e4* verantworten. Die durchgeführten Anpassungen resultieren in der Synthese von neun unterschiedlichen Varianten.

6.5.2.2 Wahlweise Auslassung der Umlagerung

Im zweiten Schritt wird das wahlweise Auslassen der Umlagerung realisiert. Auch hier werden zunächst die entsprechenden Prozessbausteine und Komponenten ermittelt. In diesem Fall handelt es sich um die Prozessbausteine, die auf einem Pfad liegen, der vom Entscheidungsknoten *E3* ausgeht. Dieser Pfad erstreckt sich über die Prozessbausteine *WaitBeforeRelocate*, *Relocate* und *Storage*. In der Komponentensammlung produziert die Komponente *sub13* diese Abfolge. Der Zieltyp dieser Komponente ist ein Bestandteil der Argumentliste der Typspezifikation der Komponente *e3*. Dementsprechend kann die Produktion dieses Pfades über das Vorhandensein des Zieltyps in der Argumentliste dieser Komponente beeinflusst werden. Um das wahlweise Auslassen dieses Pfades zu realisieren, wird nachfolgend die Komponente *e3* dupliziert. Daraufhin wird im Duplikat der Zieltyp der Komponente, die den Pfad zur Umlagerung produziert, aus der Typspezifikation entfernt. Die duplizierte Komponente, die als $e3_b$ benannt wird, erhält somit folgenden semantischen Typen:

$$\text{Sub14} \cap \text{SubProcess} \rightarrow \text{E3} \cap \text{SelectOutput}$$

Die hinzugefügte Komponente $e3_b$ produziert damit ausschließlich den Pfad, der das Auslagern modelliert. Ebenso wie in der vorherigen Anpassung wird auch in diesem Fall der Entscheidungsknoten hinsichtlich der Parametrisierung angepasst.

6.5.2.3 Wahl der Einlagerungsstrategie

Im dritten Schritt wird eine Variation der Einlagerungsstrategie realisiert. Diese kann gemäß dem Feature-Diagramm aus der Abbildung 6.9 drei Ausprägungen annehmen. Im Simulationsmodell wird die Einlagerungsstrategie mittels eines Parameters im Hauptagenten festgelegt. Dieser Parameter kann während der Modellierung in der Simulationsumgebung oder in der Startroutine des Simulationsmodells festgesetzt werden. Letztere stellt eine Kontrollstrategie dar, die zu einem festgelegten Zeitpunkt ausgeführt wird. Daher wird der Parameter in der Startroutine gesetzt und die komponentenbasierte Synthese von Kontrollstrategien wird zur Realisierung dieses Variabilitätspunkts genutzt.

Zunächst sei zu erwähnen, dass die Einlagerungsstrategie durch den Parameter `IncomingGoods` festgelegt wird. Der Datentyp dieses Parameters ist `String`. Die Simulationsumgebung AnyLogic 8 stellt die Funktion `setParameter(name, value, callOnChange)` zur Verfügung, welche die Zuweisung eines Werts `value` für einen Parameter mit dem Namen `name` erlaubt. Über das Funktionsargument `callOnChange` kann angegeben werden, ob eine festgelegte Logik ausgeführt werden soll. Beispielsweise kann die zufallsbasierte Einlagerung über den Aufruf von `setParameter(incomingGoods, "Random", true)` in der Startroutine festgelegt werden.

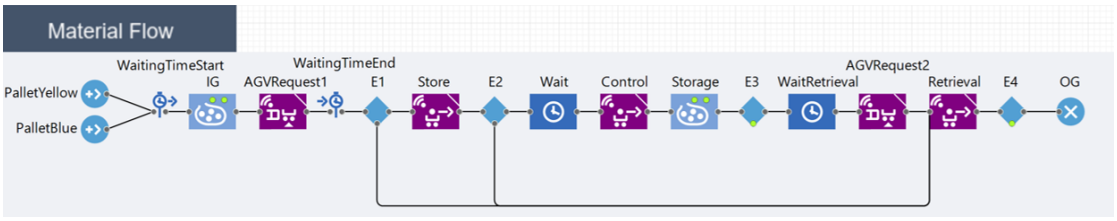
Nachfolgend wird eine Überführung des Quellcodes der Startroutine in Komponenten de-

monstriert. In diesem Fall existiert der entsprechende Quellcode nicht, sodass auch die Implementierung dieser Startroutine mittels der komponentenbasierten Synthese erfolgt. Hierzu wird für jede Einlagerungsstrategie eine separate Komponente erstellt, die jeweils die Festlegung eines Wertes des Parameters `IncomingGoods` verantwortet. Die semantischen Zieltypen der Komponenten werden übereinstimmend gewählt. Ferner wird den Komponenten kein Prozessbaustein als Bedingung zugewiesen, da die Einlagerungsstrategie in jeder Variante vorhanden sein muss. Die Tabelle 6.11 listet die resultierenden Komponenten auf. Die ersten drei Zeilen zeigen die Komponente, welche die Produktion einer Parametrisierung zur Festlegung einer Einlagerungsstrategie verantworten, während die letzte Zeile die Top-Level-Komponente zeigt. Letztere verantwortet die Produktion der Startroutine und erwartet eine Parametrisierung zur Festsetzung der Einlagerungsstrategie als Argument. Dadurch können für jeden synthetisierten Kontrollflussgraphen insgesamt drei Startroutinen synthetisiert werden.

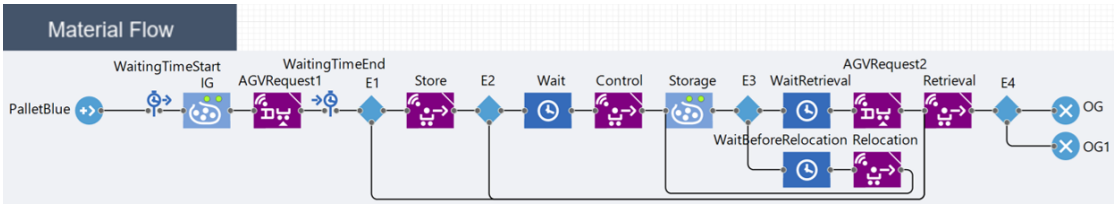
Komponente	Semantischer Typ	Quellcode
FunctionSegmentComp.	OrderStrategy	`this.setParameter("IncomingGoods", "OrderStrategies.ABCDistribution", true);`
FunctionSegmentComp.	OrderStrategy	`this.setParameter("IncomingGoods", "OrderStrategies.ABCOrderList", true);`
FunctionSegmentComp.	OrderStrategy	`this.setParameter("IncomingGoods", "OrderStrategies.RandomOrderList", true);`
FunctionConfiguration-Comp.	OrderStrategy → FunctionConfiguration ∩ Main	`$code`

Tabelle 6.11: Komponentensammlung zur Synthese einer Kontrollstrategie, welche die Strategie zum Einlagern von Gütern im Bodenblocklager festlegt.

Überdies werden im Rahmen dieses Experiments weitere Kontrollstrategien synthetisiert. Dabei handelt es sich um Kontrollstrategien, welche die Steuerung des Bestelleingangs sowie die initiale Befüllung des Lagers verantworten. Die anzupassenden Kontrollstrategien wurden folgendermaßen identifiziert: zunächst wurde das Simulationsmodell synthetisiert und in der Simulationsumgebung kompiliert. Sofern zur Kompilierzeit Fehler auftraten, konnten diese durch Betrachtung der Fehlermeldungen auf die verursachende Kontrollstrategie zurückverfolgt werden. Diese wurden anschließend ebenfalls in Produktlinien migriert und die Simulationsmodelle wurden erneut synthetisiert. Sofern weitere Fehler auftraten, wiederholte sich das Vorgehen. An dieser Stelle sei darauf hinzuweisen, dass bei zusätzlichen Prozessbausteinen, die im ursprünglichen Simulationsmodell nicht vorhanden waren, der Kompilierer zwar keine Fehler ausgibt, jedoch diese Kontrollstrategien nicht dem geforderten Verhalten entsprechen, da nicht alle relevanten Prozessbausteine berücksichtigt werden.



(a) Erste Variante des Hauptagenten, die zwei Wareneingänge, einen Warenausgang und keine Umlagerung umfasst



(b) Zweite Variante des Hauptagenten, die einen Wareneingang, zwei Warenausgänge und eine Umlagerung umfasst

Abbildung 6.10: Synthetisierte Kontrollflussgraphen, die aus der Migration des Simulationsmodells des Bodenblocklagersystems resultieren (Quelle: Kallat et al. [59]).

6.5.3 Komponentenbasierte Synthese der Simulationsmodelle

Bei der Ausführung der Synthese mit den durchgeführten Anpassungen an der Komponentensammlung können 54 Varianten synthetisiert werden. Die komponentenbasierte Synthese sämtlicher Varianten benötigt im Durchschnitt 3 Sekunden. Die Abbildung 6.10 zeigt zwei exemplarische synthetisierte Kontrollflussgraphen. Die erste Lösung beinhaltet zwei Wareneingänge, einen Warenausgang sowie keine Umlagerung der Paletten. Die zweite Lösung beinhaltet einen einzelnen Wareneingang, zwei Warenausgänge und ermöglicht eine Umlagerung der Paletten.

Auch in diesem Experiment steht die Auswertung in Bezug auf logistische Kennzahlen nicht im Vordergrund, sondern die Demonstration zur Fähigkeit der komponentenbasierten Synthese von strukturell verschiedenen Simulationsmodellen. In der gemeinsamen Veröffentlichung in [59], die aus diesem Anwendungsfall resultiert, findet sich eine Auswertung der Ergebnisse dieses Experiments. Die Kernaussage der Auswertung umfasst, dass in diesem Experiment die Begrenzung auf ein fahrerloses Transportmittel den Flaschenhals darstellte.

6.6 Diskussion

Die Migration von komplexen Simulationsmodellen, wie dem des Bodenblocklagers, die sich durch eine große Anzahl an Prozessbausteinen und unterschiedlichen Pfaden im Kontrollflussgraphen charakterisieren, zeigt, dass die resultierenden Komponentensammlungen aufgrund einer hohen Anzahl an Komponenten unübersichtlich werden können. Dieser Um-

stand erschwert die Ermittlung und Anpassung der Komponenten zur Realisierung geforderter Variabilitätspunkte. Ferner kann kritisch hinterfragt werden, ob und in welchem Zeitraum eine anwendende Person das Grundverständnis für die komponentenbasierte Synthese und das Typsystem erlangen kann, das für die Verwendung der Technologie erforderlich ist. Um Letzteres abzuschwächen, vereinfacht das entwickelte Frontend bereits die Anpassung der Typspezifikationen, sodass etwa das Hinzufügen eines Arguments einer Komponente grafisch erfolgen kann. Allerdings sollte in zukünftiger Forschung untersucht werden, inwiefern eine Abstraktionsschicht, wie in Form einer Standardisierung wie CMSD, eine Vereinfachung der Verwendung darstellt. Die Abstraktionsschicht könnte die enthaltenen Prozessbausteine sowie die Verbindungen in einem Kontrollflussgraphen beschreiben. So könnten die Verbindungen beispielsweise um Metainformationen erweitert werden, die angeben, ob eine Verbindung zwingend erforderlich ist oder wahlweise entfallen kann. Dadurch könnten die Variabilitätspunkte durch die anwendende Person beschrieben werden, ohne diese auf der Ebene der Komponenten, sondern auf Basis der Prozessbausteine zu beschreiben. Im Hintergrund könnte eine Umwandlung dieser Abstraktionsschicht in Komponenten für die Synthese automatisiert erfolgen.

Außerdem zeigt die Migration und die anschließende Ausführung der synthetisierten Simulationsmodelle in diesem Kapitel, dass das manuelle Starten der Ausführung der Simulationsmodelle bei einer hohen Anzahl an Varianten zeitaufwendig ist. Angesichts dessen wird eine Automatisierung des Startens und der Auswertung eines Simulationsmodells benötigt. Eine initiale Idee bestand in der Nutzung cloudbasierter Simulationsumgebungen. So wird auch in [126] aufgeführt, dass die Bereitstellung von Simulationsmodellen in der Cloud einen an Bedeutung gewinnenden Trend darstellt. Deshalb wurde der Einsatz der AnyLogic Cloud, einer cloudbasierten Plattform zur Ausführung von Simulationsmodellen der Simulationsumgebung AnyLogic 8, untersucht. Jedoch konnte dieses Vorgehen nicht realisiert werden, da in der aktuellen Version der Simulationsumgebung AnyLogic 8 keine Schnittstelle vorhanden ist, um synthetisierte Simulationsmodelle über die Kommandozeile in die AnyLogic Cloud zu laden. Eine alternative Lösung, die seitens des Kundendienstes von AnyLogic vorgeschlagen wurde, besteht darin, die synthetisierten Simulationsmodelle über die Kommandozeile automatisiert in eine ausführbare Java Archive (JAR) Datei zu überführen und diese erzeugte Datei über die Kommandozeile zu starten. Diese Lösung würde zwar nicht die cloudbasierte Plattform und deren Vorteile nutzen, jedoch zumindest das Starten, die Ausführung und Auswertung eines Simulationsmodells automatisieren.

Ferner kann die Art der Generierung diskutiert werden, die als extern klassifiziert ist und dementsprechend Simulationsmodelldateien auf dem lokalen System erzeugt. Dieses Vorgehen kann keine Zukunftssicherheit der Implementierung gewährleisten, da jede Aktualisierung einer Simulationsumgebung eine potenzielle Änderung des proprietären Formats nach sich ziehen kann. Andererseits können in vielen Fällen die Änderungen in relativ kurzer Zeit in den Implementierungsdetails angepasst werden. Dennoch sollte in Zukunft untersucht werden, ob die automatisierte Migration nicht intern erfolgen sollte oder die Simulationsmodelle in der Form von Standardisierungen generiert werden. Diese Standardisierungen könnten

dann mittels einer weiteren Implementierung in die proprietären Formate der Simulationsumgebungen exportiert werden. Letzteres Vorgehen erlaubt, dass bei einer Änderung eines proprietären Formats einer Simulationsumgebung lediglich eine Änderung des Übersetzers notwendig ist.

Die Synthese der Kontrollstrategien resultiert zum einen in maßgeschneiderten Kontrollstrategien, welche die Funktionsfähigkeit eines Simulationsmodells gewährleisten. Zum anderen kann damit gewährleistet werden, dass die synthetisierten Kontrollstrategien ausschließlich die Quellcodefragmente enthalten, die für die jeweilige Variante des Simulationsmodells relevant sind. Dies wiederum resultiert in einer geringeren Komplexität der Kontrollstrategie, da insbesondere `if/else`-Statements zur Separierung unterschiedlicher Fälle entfallen. Eine Herausforderung bei der Synthese der Kontrollstrategien besteht in der Erkennung von Fehlern, die auf eine fehlerhafte Aufteilung einer Kontrollstrategie in Komponenten zurückzuführen sind. Hierbei können einfache Überprüfungen im webbasierten Frontend Abhilfe schaffen, die beispielsweise prüfen, ob für alle Platzhalter in einer lückenhaften Kontrollstrategie entsprechende Komponenten vorhanden sind. Ferner sei an dieser Stelle zu erwähnen, dass die Kontrollstrategien häufig bereits bei der Implementierung derart implementiert werden können, dass sie kompatibel mit Strukturvarianten eines Simulationsmodells sind und somit keiner Anpassung bedürfen.

6.7 Zusammenfassung und Ausblick

In diesem Anwendungsfall wurde die Technologie zur Migration von agentenbasierten Simulationsmodellen in Produktlinien derart erweitert, sodass auch die Kontrollstrategien eines Simulationsmodells migriert werden können. Ferner wurden die initialen Kosten der Migration durch eine automatisierte Überführung eines Teils eines Simulationsmodells in getypte Komponenten reduziert. Auch wurde gezeigt, wie eine anwendende Person durch die Anpassung der Typspezifikationen der Komponenten die geforderten Variabilitätspunkte eines Simulationsmodells definieren kann, ohne die Implementierung der Komponentensammlung anzupassen. Das Vorgehen zur Migration und die automatisierte Erstellung der Komponentensammlung wurden im Rahmen von zwei Experimenten evaluiert. Im ersten Experiment wurde ein Simulationsmodell eines fiktiven Produktionssystems in eine Produktlinie migriert, deren Varianten sich hinsichtlich der Anzahl und Typen der vorkommenden Maschinen, dem Vorhandensein eines Puffers sowie der Summe der Auftragsquellen und -senken unterscheiden. Außerdem wurde im Rahmen des Experiments eine Kontrollstrategie synthetisiert, die die entsprechende Anzahl der Maschinen initialisiert. Das zweite Experiment umfasste ein Simulationsmodell eines Bodenblocklagers. Dabei wurde die Anzahl der Wareneingänge und -ausgänge sowie das Vorhandensein einer Funktionalität zur Umlagerung variiert. Ferner wurde mittels der Synthese einer Kontrollstrategie die Variation einer Einlagerungsstrategie realisiert.

Kapitel 7

Einbettung der komponentenbasierten Synthese in Simulationsmodelle

In diesem Kapitel wird ein Vorgehen zur graduellen Migration eines Simulationsbaukastens in eine Produktlinie zur komponentenbasierten Synthese von Simulationsmodellen vorgestellt. Nachfolgend bezeichnet ein Simulationsbaukasten eine Sammlung von zusammenhängenden Teil-Simulationsmodellen (Bausteine), welche in parametrisierter und zusammengesetzter Form die Modellierung einer Vielzahl unterschiedlicher Simulationsmodelle realisieren. Im Kontext der graduellen Migration werden ausgewählte Bausteine in Komponenten überführt. Die Parametrisierung und Komposition der Bausteine erfolgt mittels der komponentenbasierten Synthese. Eine weitere Besonderheit in diesem Kapitel betrifft die Art der Generierung der Simulationsmodelle. Diese erfolgt nämlich intern, welche gänzlich ohne die Generierung von lokalen Dateien auskommt und vollständig in der Simulationsumgebung eingebunden ist. Dies resultiert in einer Verwendung der komponentenbasierten Synthese, ohne dass diese für die anwendende Person sichtbar ist. Aus einer technischen Sichtweise betrachtet werden somit keine XML-Fragmente synthetisiert, sondern Datenobjekte in der Laufzeitumgebung des Simulationsmodells.

Als Anwendungsfall wird in diesem Kapitel erneut das blechverarbeitende Fabriksystem aus dem Kapitel 5 betrachtet. Während in Kapitel 5 hauptsächlich die Filterung von Lösungen untersucht wurde, fokussiert dieses Kapitel die graduelle Migration eines komplexen Simulationsbaukastens. Ferner werden in diesem Kapitel keine Kontrollflussgraphen, wie in Kapitel 6, synthetisiert, sondern Kompositionen der Teil-Simulationsmodelle.

Die nachfolgenden Ergebnisse entstanden in einer Zusammenarbeit mit den Firmen TRUMPF Werkzeugmaschinen SE & Co. KG und ITK Engineering GmbH. Im Rahmen einer Machbarkeitsstudie wurde der Simulationsbaukasten graduell in eine Produktlinie migriert. Die Ergebnisse dieser Machbarkeitsstudie wurden zudem in einem gemeinsamen Konferenzbeitrag (vgl. [37]) vorgestellt. Angesichts dessen weisen die Inhalte des Konferenzbeitrages und dieses Kapitels

stellenweise Übereinstimmigkeiten auf.

7.1 Beschreibung des Anwendungsfalls

Wie bereits erwähnt, wird auch in diesem Kapitel ein blechverarbeitendes Fabriksystem als Anwendungsfall betrachtet. Das Fabriksystem beinhaltet eine Vielzahl unterschiedlicher Maschinentypen, wobei im Rahmen dieses Kapitels jeweils genau eine Instanz der folgenden Maschinentypen betrachtet wird:

- 2D-Laserschneidemaschinen
- Biegemaschinen
- Stanz-Lasermaschinen

2D-Laserschneidemaschinen ermöglichen das automatisierte Schneiden von Blechen mittels der namensgebenden Lasertechnologie. Das automatisierte Biegen einer Vielzahl von Blechteilen wird durch Biegemaschinen im Fabriksystem ermöglicht. Stanz-Lasermaschinen stellen universale Maschinen dar, die das kombinierte Stanzen und Schneiden von Blechteilen erlauben. Neben den Maschinen ist im betrachteten Fabriksystem ein voll automatisiertes Lagersystem vorhanden, welches an die Maschinen angebunden ist und das Rohmaterial sowie die bearbeiteten Materialien beinhaltet. Das Lagersystem besteht aus einer variablen Anzahl an Lagertürmen, welche bis zu 24 Regale enthalten. Die Laserschneide- und Biegemaschinen sind jeweils an zwei Türmen und die Stanz-Lasermaschine an genau einen Turm des Lagersystems angeschlossen. Innerhalb des Lagersystems befindet sich ein Regalbediengerät (RBG), das den Transport von Materialien verantwortet. Dementsprechend bewegt sich das RBG über die gesamte Länge des Lagersystems. Im Fabriksystem wird ein Auftragsportfolio abgearbeitet, das die Verarbeitung von Blechteilen durch unterschiedliche Maschinen beschreibt. Die Abbildung 7.1 zeigt den schematischen Aufbau eines beispielhaften Systems, welches drei unterschiedliche Maschinentypen und ein Lagersystem mit 16 Türmen zeigt.

Das Feature-Diagramm in der Abbildung 7.2 zeigt die Variabilitätspunkte in diesem Anwendungsfall. Der erste Variabilitätspunkt bezieht sich auf die Anzahl der Lagertürme, die zwischen 4 und 144 variieren kann. Die verbleibenden Variabilitätspunkte sehen die Variation der Anzahl der Maschinen vor, wobei jeder Maschinentyp mindestens zweimal und höchstens fünfmal auftreten darf. An dieser Stelle sei zu erwähnen, dass nicht jede Kombination der Anzahl an Lagertürmen und Maschinen eine valide Konfiguration darstellt. So müssen ausreichend Lagertürme vorhanden sein, um eine entsprechende Anzahl an Maschinen mit Rohmaterial zu versorgen und das verarbeitete Material zu lagern. Neben den Variabilitätspunkten, die durch die anwendende Person beschrieben werden, beinhaltet der Anwendungsfall weitere Variabilitätspunkte, die sich durch die Variation der Struktur des Simulationsmodells ergeben. Kontrollstrategien müssen so etwa zur Initialisierung des Simulationsmodells oder

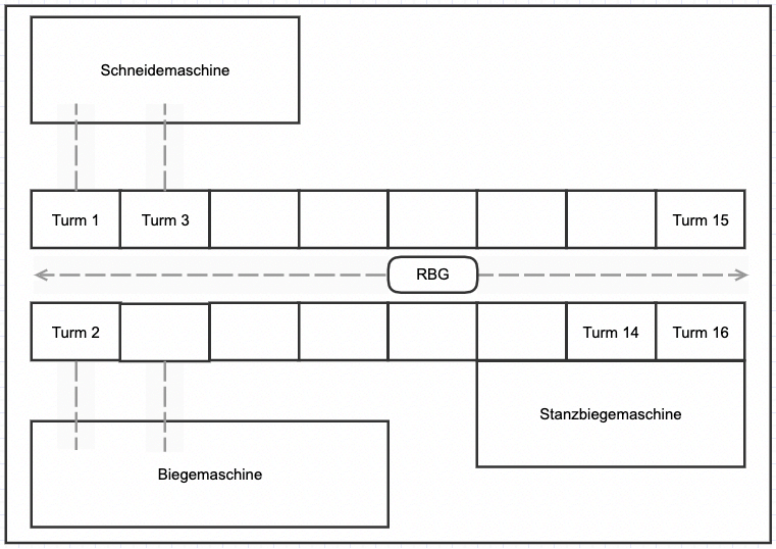


Abbildung 7.1: Schematischer Aufbau eines beispielhaften Shop-Floor-Systems mit drei Maschinen (2D-Laserschneide-, Biege- und Stanz-Lasermaschine), die an ein Lagersystem mit 16 Lagertürmen angeschlossen sind. Das Lagersystem verfügt über ein automatisiertes Regalbediengerät (RBG), welches die Be- und Entladung der Maschinen sowie Umlagerungen innerhalb des Lagersystems vornimmt.

gezeichnete Pfade zur Fortbewegung der Werker an die jeweilige Strukturvariante angepasst werden. Letzteres erfolgt ebenfalls im Rahmen der komponentenbasierten Synthese.

Der zu migrierende Simulationsbaukasten kann als ein *Master*-Simulationsmodell angesehen werden, das alle möglichen Konfigurationen eines Systems abdeckt. Der Simulationsbaukasten besteht aus einer Reihe von Teil-Simulationsmodellen, die in Form von Agenten in der Simulationsumgebung AnyLogic 8 modelliert sind. Die Bausteine repräsentieren unter anderem unterschiedliche Maschinen, Automatisierungseinheiten oder Lagersysteme. Diese Bausteine beinhalten die entsprechende Logik als auch die korrespondierenden 2D- und 3D-Visualisierungen. Ansonsten sind auch die Bausteine ein Teil des Baukastens, die zur Erfassung von Statistiken oder Auftragsverteilung eingesetzt werden.

Der gewöhnliche händische Arbeitsablauf zur Modellierung einer Konfiguration eines Fabriksystems mit dem Simulationsbaukasten setzt sich aus folgenden Schritten zusammen:

1. Erstellung eines initialen, leeren Simulationsmodells
2. Auswahl entsprechender Bausteine des Simulationsbaukastens
3. Platzierung und Parametrisierung im leeren Simulationsmodell
4. Verbindung der einzelnen Bausteine
5. Hinzufügung der Visualisierung (z. B. Pfade für Werker oder Transportsysteme)

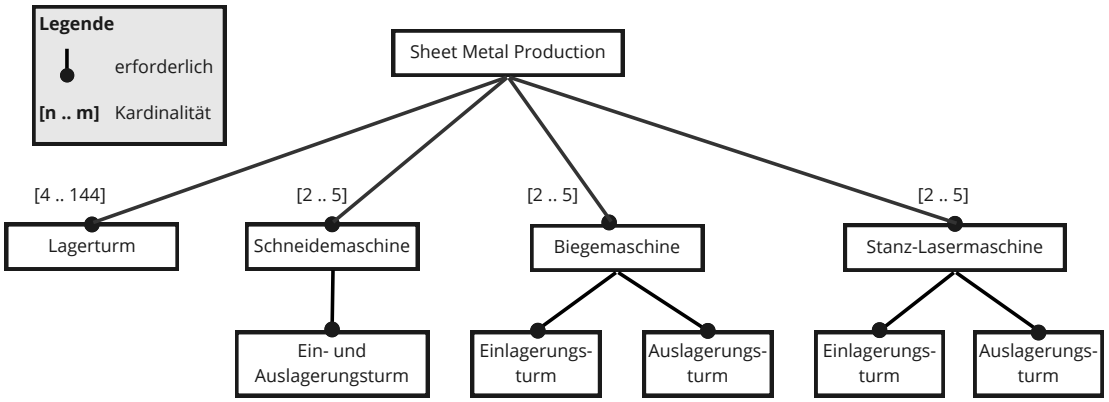


Abbildung 7.2: Feature-Diagramm, das die Variabilitätspunkte des Shop-Floor-Systems des dritten Anwendungsfalls illustriert. Diese setzen sich aus unterschiedlichen Anzahlen von Maschinen (zwei bis maximal fünf) und Lagertürmen (vier bis maximal 144) sowie der Zuordnung von Maschinen zu Lagertürmen zusammen.

Sofern eine weitere, ähnliche Konfiguration benötigt wird, wird bei diesem Arbeitsablauf das Simulationsmodell kopiert und die entsprechenden Änderungen werden händisch vollzogen. Der Einsatz des Simulationsbaukastens ermöglicht somit einen erheblichen Zeitgewinn in der Modellierung der entsprechenden Simulationsmodelle, da die Funktionalitäten und Logiken, die durch die Bausteine umgesetzt werden, nicht für jedes Simulationsmodell erneut modelliert werden müssen. Um die benötigte Dauer zur Modellierung der Simulationsmodelle, insbesondere von Varianten, weiter zu reduzieren, wird in diesem Kapitel eine Automatisierung der Komposition und Parametrisierung der Bausteine untersucht. Weiterhin soll die automatisierte Synthese unterschiedlicher Varianten für anwendende Personen ohne Kenntnisse in der Simulationsmodellierung und der komponentenbasierten Synthese ermöglicht werden. Hierzu soll die anwendende Person, das gewünschte Shop-Floor-System durch die Angabe von Wertebereichen für die einzelnen Variabilitätspunkte konfigurieren können. Um dieses Vorhaben zu realisieren, wird ein Teil des Simulationsbaukastens in eine Produktlinie migriert und die komponentenbasierte Synthese in der Simulationsumgebung eingebettet.

7.2 Verwandte Arbeiten

In der Literatur existieren einige Ansätze, welche die Generierung eines Simulationsmodells über eine Konfiguration des Zielsystems ermöglichen. Häufig werden generische Simulationsmodelle verwendet, die mehrere Varianten eines Systems inhärent abbilden. Wincheringer et al. stellen in [129] ein generisches Simulationsmodell vor, das eine dynamische Konfiguration eines Hochregallagers ermöglicht. Die Autorenschaft verwendet die Simulationsumgebung Siemens Plant Simulation, die über zur Verfügung gestellte Schnittstellen das dynamische Hinzufügen und Entfernen von Bausteinen im Simulationsmodell erlaubt. Dieser Aspekt stellt in diesem Kapitel eine Herausforderung dar, da diese Schnittstellen in der aktuellen Version

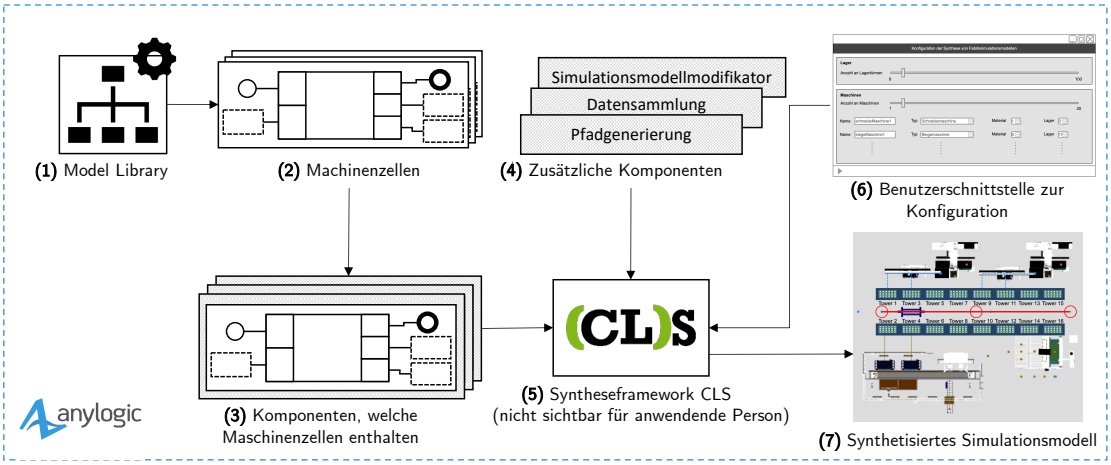


Abbildung 7.3: Architektur und Ablauf der komponentenbasierten Synthese einer Familie von Simulationsmodellen eines Shop-Floor-Systems. Der Ansatz ist in der Simulationsumgebung AnyLogic 8 eingebettet (blauer, gestrichelter Kasten). Dieses beinhaltet die Bausteine der Model Library (1), welche zur Modellierung von Maschinenzellen (2) eingesetzt werden. Diese Maschinenzellen werden in Komponenten (3) verpackt und mit weiteren Komponenten (4) als Eingabe für das Synthese-Framework CLS verwendet. Gegeben der Konfiguration eines Shop-Floor-Systems (6), wird das CLS-Framework zur Synthese eines entsprechenden Simulationsmodells verwendet (7) (Quelle: [37]).

der Simulationsumgebung AnyLogic 8 nicht vorhanden sind. Kassen et al. haben in [62] ein generisches Simulationsmodell in der Simulationsumgebung AnyLogic 8 entwickelt, das Produktionssysteme abbildet. In ihrem Ansatz erfolgt eine datengetriebene Konfiguration über eine REST-Schnittstelle, die an ein Enterprise-Ressource-Planning-System angeschlossen ist. Khemiri et al. ermitteln in [63] eine Reihe von Forschungsarbeiten bezüglich generischer Simulationsmodelle von Halbleiter-Produktionssystemen. In ihrer Arbeit stellen die wissenschaftlich Publizierenden ein generisches Simulationsmodell vor. Dieses Simulationsmodell berücksichtigt aktuelle Trends im Bereich der Halbleiter-Produktion und ermöglicht die Untersuchung einer Vielzahl von Problemen in diesen Systemen. Auch hier wurde die Simulationsumgebung AnyLogic 8 zur Modellierung des Simulationsmodells verwendet. Weiterhin sei an dieser Stelle erneut die Arbeit von Neyrinck et al. [87] zu erwähnen, die ebenfalls den Fokus auf eine bedienungsfreundliche Interaktion zur Generierung von Simulationsmodellen legt.

7.3 Lösungsansatz

In diesem Kapitel wird der Lösungsansatz skizziert, der zum einen den Simulationsbaukasten in eine Produktlinie migriert und zum anderen, die interne Generierung eines Simulationsmodells mittels der komponentenbasierten Softwaresynthese in AnyLogic 8 ermöglicht. Die Abbildung 7.3 zeigt die Architektur des Lösungsansatzes sowie den allgemeinen Ablauf.

Der Lösungsansatz ist dabei vollständig in der Simulationsumgebung AnyLogic 8 eingebettet, das durch den gestrichelten, blauen Rahmen dargestellt ist. Das heißt, dass sowohl die Konfiguration als auch die komponentenbasierte Synthese der Simulationsmodelle gänzlich intern erfolgen. Die Basis in diesem Lösungsansatz bildet der Simulationsbaukasten, der vom Unternehmen TRUMPF Werkzeugmaschinen zur Verfügung gestellt wurde. Basierend auf dem Simulationsbaukasten wird eine Abstraktionsschicht (2) eingeführt, die Maschinenzellen beinhaltet. Diese Maschinenzellen repräsentieren Maschinen inklusive ihrer Automatisierungseinheiten und stellen somit vordefinierte Konfigurationen dar. Die Abstraktionsschicht dient zur Reduktion der Komplexität dieses Anwendungsfalls, welche die Erlangung von ersten Ergebnissen innerhalb der Machbarkeitsstudie in relativ kurzer Zeit ermöglicht. Im nächsten Schritt folgt eine Überführung der Maschinenzellen in Komponenten (3), welche im Rahmen der komponentenbasierten Synthese die Produktion der Maschinenzellen im Simulationsmodell verantworten. Ferner sind weitere Komponenten (4) implementiert, die etwa Referenzen aus dem Simulationsmodell extrahieren. Diese Referenzen werden von weiteren Komponenten zur Modifikation des Simulationsmodells oder zum Zeichnen von Pfaden (z. B. für die Werker) verwendet. Sämtliche Komponenten bilden die Eingabe für das CLS-Framework (5), das die Synthese des Simulationsmodells verantwortet, dabei aber für die anwendende Person nicht sichtbar ist. Letztere interagiert ausschließlich mit der grafischen Benutzeroberfläche (6), die eine Konfiguration durch die Festlegung der Anzahl der einzelnen Maschinentypen sowie der Lagertürme ermöglicht. Anhand dieser Konfiguration erfolgt die Synthese, die das entsprechende Simulationsmodell (7) produziert.

7.3.1 Abstraktion des zu migrierenden Simulationsmodells

Die Abstraktionsschicht setzt sich aus zwei Bestandteilen zusammen, und zwar den Maschinenzellen sowie einem initial leeren Simulationsmodell. Die Maschinenzellen werden als Agenten im Simulationsmodell modelliert und repräsentieren vordefinierte Konfigurationen einer Maschine inklusive ihrer Automatisierungseinheiten. So beinhaltet eine beispielhafte Konfiguration eine 2D-Laserschneidemaschine mit einer Automatisierungseinheit zum Be- und Entladen, die rechts von der Maschine angeordnet ist. In einer anderen potenziellen Konfiguration könnte die Be- und Entladung durch einen Werker erfolgen. Im Rahmen dieses Anwendungsfalls werden drei Maschinenzellen modelliert, die jeweils eine Instanz der Maschinen zum 2D-Laserschneiden, Biegen und dem kombinierten Stanzen und Schneiden repräsentieren. Die Modellierung erfolgt durch die Verwendung der entsprechenden Bausteine des Simulationsbaukastens, die je nach Konfiguration parametrisiert werden.

Die Abbildung 7.4 zeigt den schematischen Aufbau einer Maschinenzelle, die durch einen Agenten modelliert wird. Das zentrale Element dieses Agenten bildet der mittig platzierte Baustein, die aus dem Simulationsbaukasten entstammt und eine parametrisierte Maschine repräsentiert. Die Parametrisierung wird innerhalb des Agenten vorgenommen und bezieht sich beispielsweise auf die Positionierung der Be- und Entladung oder die Schneidegeschwindigkeit. Der Maschinenbaustein verfügt über Anschlüsse, die durch Rechtecke am linken und

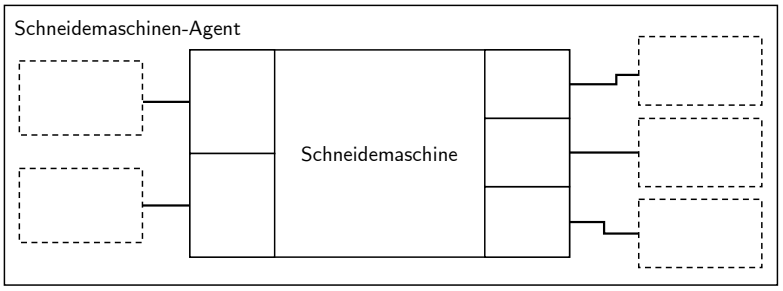


Abbildung 7.4: Schematische Darstellung eines Agenten, der eine Maschinenzelle einer 2D-Laserschneidemaschine repräsentiert. Der „innere“ Kasten stellt den eingesetzten Baukasten der ausgewählten 2D-Lasermaschine dar. Innerhalb dieses Bausteins befinden sich jeweils am linken und rechten Rand weitere Kästen, welche die Anschlüsse des Bausteins darstellen. Die gestrichelten Kästen stellen Kompositionen von Bausteinen dar, die eine Logik implementieren, die die Anschlüsse des Maschinenbausteins mit den entsprechenden Eingaben versorgen.

rechten Rand symbolisiert werden. Die Anschlüsse der linken Seite stellen die Eingaben einer Maschine dar, wie Aufträge oder Rohmaterial. Letzteres wird beispielsweise von einer Automatisierungseinheit zur Verfügung gestellt. Im Rahmen der Modellierung einer Maschinenzelle muss dementsprechend ebenfalls die Logik der Automatisierungseinheit modelliert werden. Hierfür können parametrisierte Bausteine des Simulationsbaukastens verwendet werden. Die Anschlüsse auf der rechten Seite der Maschine stellen die Ausgaben dar. Diese Ausgaben werden von weiteren Logiken und Bausteinen verarbeitet, die etwa das verarbeitete Material einlagern oder den Verschnitt entsorgen. Diese Logiken sind in der Abbildung 7.4 durch die gestrichelten Kästen angedeutet. Die Kanten zwischen den Bausteinen und Logiken stellen die Verbindung zwischen diesen her.

Der zweite Bestandteil der Abstraktionsschicht, das initial leere Simulationsmodell, stellt das Simulationsmodell dar, das die Maschinenzellen beinhaltet und schlussendlich ausgeführt wird. Dieses Simulationsmodell, nachfolgend als *Vorlage* bezeichnet, wird mittels der komponentenbasierten Synthese dynamisch mit Maschinenzellen befüllt. Das heißt, dass es als Basis für eine Vielzahl unterschiedlicher Varianten dient. Das Simulationsmodell besteht aus *statischen* und *dynamischen* Elementen. Erstere umfassen Bausteine, die in jeder Variante vorkommen und etwa zur Erfassung von Kennzahlen oder zum Laden des Auftragsportfolios dienen. Ausgewählte statische Elemente müssen nach der Synthese und der Befüllung des initialen Simulationsmodells erneut initialisiert werden. So müssen etwa die synthetisierten Maschinenzellen in der Maschinenstatistik zur Erfassung registriert werden. Die dynamischen Elemente hingegen bezeichnen die Bausteine, die nicht in allen Variante vorkommen. Dies betrifft insbesondere die Maschinenzellen, die sich in ihrer Anzahl der Vorkommen je nach Variante unterscheiden.

Die dynamische Befüllung des initialen Simulationsmodells mit Maschinenzellen wird mittels sogenannter Populationen realisiert. Letztere stellen in der Simulationsumgebung AnyLogic 8 eine Datenstruktur dar, die ähnlich zu Listen in gängigen Programmiersprachen ist.

Jede Population kann ausschließlich mit einem festgelegten Agententyp befüllt werden. Zur Laufzeit der Simulation kann der Zugriff auf die Populationen erfolgen, um die enthaltenen Agenten abzurufen oder weitere Agenten hinzufügen. Im initialen Simulationsmodell sind drei Populationen enthalten, die jeweils Agenten eines Maschinenzellentyps enthalten. Die Populationen sind zu Beginn leer und werden nach der Synthese mit den synthetisierten Maschinenzellen befüllt. An dieser Stelle sei zu erwähnen, dass versuchsweise die Instantiierung einer entsprechenden Anzahl an Maschinenzellen-Agenten und das unmittelbare Hinzufügen im Hauptagenten ausgetestet wurden. Jedoch erwies sich dieser Ansatz als erfolglos, da in der Startroutine der Simulationsumgebung AnyLogic 8 das Simulationsmodell bereits initialisiert wurde und somit keine Änderungen der Struktur möglich sind.

7.3.2 Aufbau der Komponentensammlung

Nachfolgend wird die Komponentensammlung vorgestellt, welche die Synthese der Simulationsmodelle dieses Anwendungsfalls ermöglicht. In diesem Kapitel werden nicht die vordefinierten Komponenten aus dem Kapitel 4.4.2 verwendet, da keine Kontrollflussgraphen, sondern eine Menge an Agenten, synthetisiert werden soll. Daher wurde ein Datenmodell entwickelt, das auf den aktuellen Anwendungsfall zugeschnitten und in der Abbildung 7.5 dargestellt ist. Die oberen Klassen der Abbildung werden von der Simulationsumgebung AnyLogic 8 implementiert und sind daher in der Programmiersprache Java programmiert, während die unteren Klassen ein Teil der Scala-Implementierung sind. Dadurch, dass die Simulation und die Scala-Implementierung in der Java-Virtual-Machine betrieben werden, kann das Datenmodell auch implementierungsübergreifend verwendet werden.

Die Klasse `InputData` beinhaltet die Konfiguration eines zu synthetisierenden Shop-Floor-Systems. Bei der Instantiierung dieser Klasse muss die Anzahl der gewünschten Lagertürme angegeben werden, die in der Variable `numberOfTowers` gespeichert ist. Überdies verfügt die Konfiguration über eine Liste `machines`, die Objekte des Datentyps `MachineEntry` enthält. Letztere repräsentiert eine einzelne Maschinenkonfiguration und umfasst den Maschinennamen sowie -typen. Weiterhin enthält die Konfiguration die zugeordneten Lagertürme, an denen die Maschine angeschlossen werden. Die Zuordnung erfolgt hierbei über die Nummerierung der Lagertürme. Im Falle einer Maschine, die an einen Lagerturm angeschlossen ist, enthalten beide Variablen denselben Wert.

Die Datenstruktur `Context` dient im Rahmen der Synthese zur Bündelung von Referenzen, die aus unterschiedlichen Komponenten des ursprünglichen Simulationsmodells extrahiert werden. So enthält diese eine Referenz auf die Werker (`worker`) und deren Positionierung im Simulationsmodell (`workerHome`). Ferner sind Referenzen auf die Bausteine des Lagersystems (`storage`) und der Initialisierung (`configSetup`) vorhanden. Die Datentypen dieser Referenzen stammen aus den Java-Bibliotheken der Simulationsumgebung AnyLogic 8. Diese können hier verwendet werden, da die Scala-Implementierung diese als externe Bibliotheken eingebunden hat. Somit kann auf sämtliche Methoden und Variablen dieser Datentypen

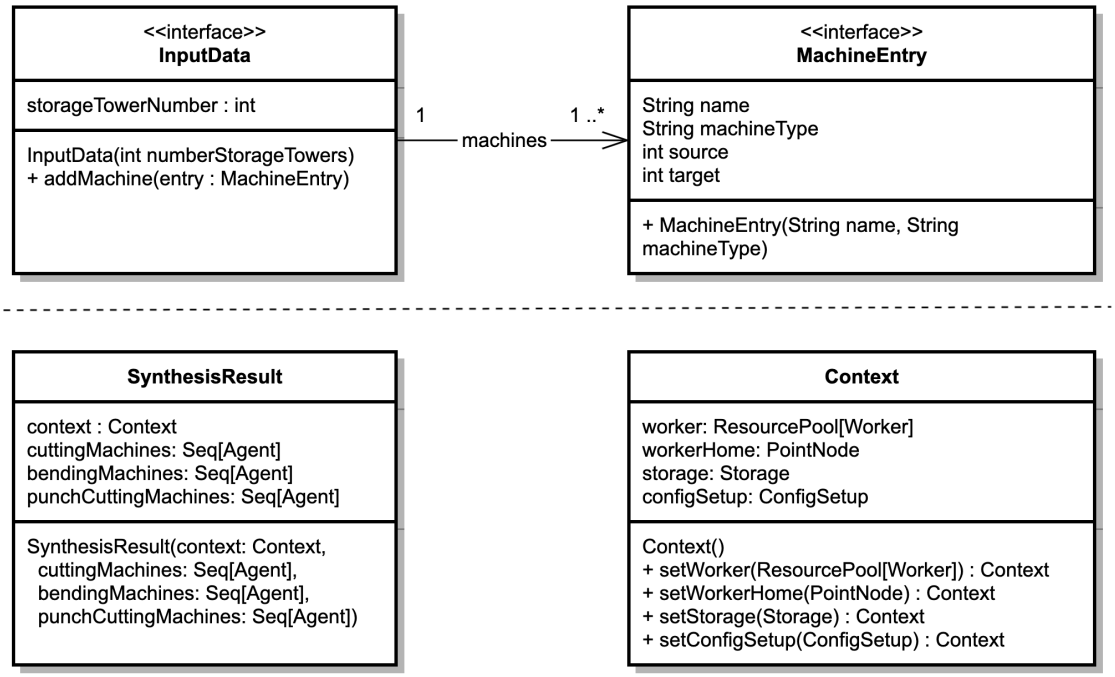


Abbildung 7.5: UML-Klassendiagramme der Datenstrukturen zum Austausch der synthetisierten Bestandteile (Klasse `SynthesisResult`) und der Daten (Klasse `Context`) zwischen der Technologie und dem Simulationsmodell.

zugegriffen werden.

Die Datenstruktur `SynthesisResult` enthält eine Instanz des Datentyps `Context` sowie drei Listen, welche die synthetisierten Maschinenzellen-Agenten beinhalten. Diese Datenstruktur wird von der Top-Level-Komponente der Komponentensammlung synthetisiert und in der Startroutine des Simulationsmodells verwendet, um die synthetisierten Agenten den Populationen hinzuzufügen. Auch dieses Objekt wird implementierungsübergreifend verwendet.

Nachfolgend wird der Aufbau der Komponentensammlung zur Synthese der entsprechenden Simulationsmodelle erläutert. Die Tabelle 7.1 zeigt die Komponenten, mittels denen die Simulationsmodellvarianten gemäß dem Feature-Diagramm aus der Abbildung 7.2 synthetisiert werden können. Im Gegensatz zu den bisherigen Anwendungsfällen sind bei der Darstellung der Komponentensammlung auch die nativen Typen aufgelistet. Dies hat den Hintergrund, dass sich zum einen diese bei den einzelnen Komponenten unterscheiden und zum anderen einen Teil der domänenspezifischen Informationen abbilden. Die Top-Level-Komponente bildet *simulationModelComponent*, die als erstes Argument ein Objekt des Datentyps `Context` erwartet. Diese wird durch die Komponente *completedContextComponent* produziert, die ebenfalls mehrere Argumente erwartet. Das erste Argument wird durch die Komponente *emptyContextComponent* produziert und stellt einen initial leeren Kontext dar. Dies ist auch dem semantischen Zieltyp der Komponente *emptyContextComponent* zu entnehmen, der dies durch eine Intersektion der Typen *Context* und *IsEmpty* ausdrückt. Die verbleibenden

Argumente verantworten die Produktion der Bestandteile des `Context`-Objekts, das durch die Komponenten *completedStorageComponent, configSetupComponent, workerHomeComponent, workerComponent* sowie *viewAreaComponent* realisiert wird.

Die Komponente *completedStorageComponent* produziert ein Lagersystem mit der gewünschten Anzahl an Lagertürmen. Hierfür benötigt die Komponente ebenfalls mehrere Argumente. Das erste Argument erwartet ein Objekt des Datentyps `InputData`, der durch die Komponente *inputDataComponent* produziert wird, und unter anderem die Anzahl der zu synthetisierenden Lagertürme enthält. Das zweite Argument stellt ein leeres Lagersystem dar, das durch die Komponente *emptyStorageComponent* produziert wird. Das dritte Argument ist für die Produktion der Visualisierung verantwortlich und wird durch die Komponente *storagePresentationComponent* realisiert. Die zuvor genannten Argumente ermöglichen die Produktion eines Lagersystems in den Implementierungsdetails der Komponente *completedStorageComponent*.

Die Komponente *configSetupComponent* produziert ein Objekt des Datentyps `ConfigSetup`. Dabei handelt es sich um einen Baustein des Simulationsbaukastens, der für das Laden von Aufträgen aus dem Auftragsportfolio verantwortlich ist. Die Komponenten *workerComponent* und *workerHomeComponent* verantworten die Produktion der Ressource der Werker sowie die Positionierung dieser im Simulationsmodell. Mit den vorgestellten Komponenten kann die Produktion des `Context`-Objekts und infolgedessen des ersten Arguments der Top-Level-Komponente erfolgen.

Die verbleibenden Argumente der Top-Level-Komponenten erwarten jeweils Sequenzen von Agenten, welche die einzelnen Maschinenzellen beinhalten. So umfasst das zweite Argument die Agenten, die Maschinenzellen einer 2D-Laserschneidemaschine repräsentieren. Diese Agenten werden in den Implementierungsdetails der Komponente *cuttingMachinesComponent* produziert. Hierfür benötigt die Komponente die Referenz auf das Simulationsmodell, die im `Context`-Objekt enthalten ist, sowie die Anzahl der zu synthetisierenden Laserschneidemaschinen, die wiederum Teil der Konfiguration im `InputData`-Objekt ist. Somit wird die Anzahl der zu synthetisierenden Maschinenzellen nicht über das Typsystem beeinflusst. Dies hat den Grund, dass in diesem Anwendungsfall die einzelnen Maschinenzellen jedes Maschinentyps identisch aufgebaut sind und sich als Folge dessen kein Mehrwert bei einer Steuerung der Anzahl mittels der Typspezifikationen ergeben würde. Mit den aufgezählten Argumenten kann die entsprechende Anzahl der Maschinenzellen produziert und in Form einer Liste zurückgegeben werden. Hierfür wird die Datenstruktur `Seq` der Programmiersprache Scala gewählt. Die Agenten der Maschinenzellen der Biegemaschinen und Stanz-Lasermaschinen werden durch die Komponenten *bendingMachinesComponent* und *punchCuttingMachinesComponent* produziert. Auch diese Komponenten erwarten Objekte des Datentyps `Context` und `InputData` als Argumente. Damit kann die Argumentliste der Top-Level-Komponente vollständig bedient werden. Diese bündelt die synthetisierten Agenten und das `Context`-Objekt in einer neuen Instanz des Datentyps `SynthesisResult`, die das Ergebnis der Synthese bildet. Letzteres wird in der Startroutine des Simulationsmodells verwendet, um die synthetisierten Agenten

Komponentenname	Nativer Typ	Semantischer Typ
simulationModel-Component	Context → Seq[Agent] → Seq[Agent] → Seq[Agent] → SynthesisResult	*Context* ∩ *IsComplete* → *CuttingMachines* → *BendingMachines* → *PunchCuttingMachines* → *TopLevelComponent*
completedContext-Component	Context → BasicStorageWrapper → ConfigSetupWrapper → PointNode → WorkerWrapper → Context	*Context* ∩ *IsEmpty* → *BasicStorage* ∩ *HasStorageTowers* → *ConfigSetup* → *WorkerHome* → *Worker* → *Context* ∩ *IsComplete*
emptyContext-Component	Context	*Context* ∩ *IsEmpty*
emptyStorage-Component	BasicStorageWrapper	*Storage* ∩ *IsEmpty*
storagePresentation-Component	StoragePresentation	*StoragePresentation*
completedStorage-Component	InputData → StorageWrapper → StoragePresentation → StorageWrapper	*InputData* → *Storage* ∩ *IsEmpty* → *StoragePresentation* → *Storage* ∩ *HasStorageTowers*
workerHome-Component	PointNode	*WorkerHome*
workerComponent	WorkerWrapper	*Worker*
configSetupComponent	ConfigSetupWrapper	*ConfigSetup*
cuttingMachine-Component	Context → InputData → Seq[Agent]	*Context* ∩ *IsComplete* → *InputData* → *CuttingMachines*
bendingMachine-Component	Context → InputData → Seq[Agent]	*Context* ∩ *IsComplete* → *InputData* → *BendingMachines*
punchCutting-MachineComponent	Context → InputData → Seq[Agent]	*Context* ∩ *IsComplete* → *InputData* → *PunchCuttingMachines*
inputDataComponent	InputData	*InputData*

Tabelle 7.1: Komponenten mit ihren Typspezifikationen zur komponentenbasierten Synthese der Simulationsmodelle eines Shop-Floor-Systems.

und angepassten Bausteine in das Simulationsmodell einzufügen. Bevor auf diesen Schritt eingegangen wird, werden zunächst die Implementierungen der vorgestellten Komponenten näher vorgestellt.

In diesem Abschnitt werden die Implementierungen der Komponenten vorgestellt. Zunächst wird auf eine Besonderheit eingegangen, die bei der Betrachtung der nativen Typen ausgewählter Komponenten auffällt. So produziert die Komponente *configSetupComponent* ein Objekt des Datentyps `ConfigSetupWrapper` und keine Instanz der Klasse `ConfigSetup`. Bei `ConfigSetupWrapper` handelt es sich um eine Datenstruktur, die als Umhüllung für die Datenstruktur `ConfigSetup` der AnyLogic 8-Bibliothek dient. In der Implementierung existieren mehrere umhüllende Datenstrukturen, die durch das Suffix *Wrapper* gekennzeichnet sind. Diese Umhüllungen werden benötigt, da eine unmittelbare Produktion ausgewählter Datenstrukturen der AnyLogic 8-Bibliothek in der verwendeten Version des CLS-Frameworks nicht möglich war, sodass Instanzen dieser Datenstrukturen nicht als Produktionsergebnis einer Komponente zurückgegeben werden können. In Listing 7.1 ist die Implementierung der umhüllenden Datenstruktur des Bausteins `ConfigSetup` dargestellt. Mittels des Konstruktor (vgl. Zeile 1) wird der zu umhüllende Baustein übergeben und die Funktion `getAgent` (vgl. Zeile 2) gibt diesen zurück.

```
1 case class ConfigSetupWrapper(var configSetup: ConfigSetup) {
2   def getAgent: ConfigSetup = configSetup
3 }
```

Listing 7.1: Implementierung einer Datenstruktur zur Umhüllung des Prozessbausteins `BasicStorage`.

Im nächsten Schritt wird auf die Instantiierung der Komponentensammlung (Repository) eingegangen. Diese erfolgt durch den Aufruf des Konstruktors der Komponentensammlung. Letztere ist in Listing 7.2 gezeigt.

```
1 class Repository(rootAgent: Agent,
2                  inputData: InputData
3                  bbFactory: IBuildingBlockFactory) {
4   // ... Komponenten ...
5 }
```

Listing 7.2: Konstruktor der Komponentensammlung, der eine Referenz auf den Hauptagenten im Template-Simulationsmodell (`rootAgent`), eine Implementierung der Maschinenzellen-Fabrik (`bbFactory`) sowie eine Synthesekonfiguration (`inputData`) erwartet. Diese werden in den Implementierungsdetails der Komponenten verwendet.

Der Konstruktor der Komponentensammlungerwartet drei Argumente. Das erste Argument `rootAgent` stellt eine Referenz auf den Hauptagenten des initialen Simulationsmodells dar, das nachfolgend innerhalb der Implementierungen der Komponenten verwendet wird, um

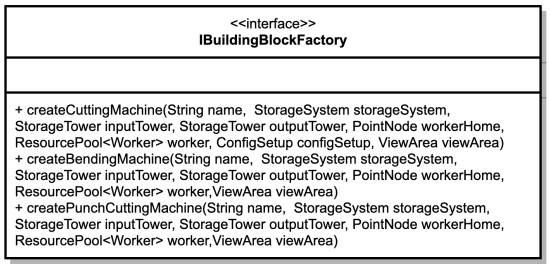


Abbildung 7.6: UML-Klassendiagramm der Schnittstelle, die eine Erzeugung von Maschinenzellen-Agenten in der Implementierung der Technologie ermöglicht. Diese umfasst eine Fabrik (Entwurfsmuster), bei der die Methoden *create[...]Machine* synthetisierte Bestandteile einer Maschinenzelle als Argumente erwarten und Instanzen der Agenten erzeugen.

Elemente des Simulationsmodells zu extrahieren. Das zweite Argument `inputData` umfasst die aktuelle Konfiguration des zu synthetisierenden Shop-Floor-Systems. Das dritte Argument `bbFactory` beinhaltet die Implementierung einer Schnittstelle, die eine Instanziierung der Maschinenzellen-Agenten erlaubt. Letztere ist in der Scala-Implementierung als ein Java-Interface implementiert, während die tatsächliche Implementierung im Simulationsmodell vorhanden ist. Das UML-Klassendiagramm des Java-Interfaces ist in der Abbildung 7.6 dargestellt. Wie dem Namen der Schnittstelle zu entnehmen ist, wird hier das Entwurfsmuster der Fabrikmethode verwendet, sodass jede der Methoden für die Erzeugung einer konkreten Maschinenzelle verantwortlich ist. Dieses Vorgehen begründet sich darin, dass die Komponentensammlung zwar über eine Referenz auf das initiale Simulationsmodell verfügt, aber keine Referenzen auf die einzelnen Agenten, welche die Maschinenzellen repräsentieren. Über die Referenz des initialen Simulationsmodells können die Agenten ebenso nicht erreicht werden. Das initiale Simulationsmodell hingegen verfügt über die notwendigen Referenzen auf diese Agenten und kann daher die Schnittstelle implementieren. Sofern weitere Maschinentypen und entsprechende Maschinenzellen ergänzt werden, muss lediglich die Fabrik um weitere Funktionen ergänzt werden.

Nachfolgend werden die Implementierungsdetails der Komponenten vorgestellt. Zunächst wird eine Komponente vorgestellt, die eine Referenz auf einen Baustein aus dem initialen Simulationsmodell extrahiert und als Produktionsergebnis zurückgibt. Hierzu wird noch einmal die Komponente *configSetupComponent* betrachtet, die den Baustein *configSetup* aus dem Simulationsmodell ausliest. Die Implementierungsdetails dieser Komponente sind in Listing 7.3 dargestellt. In der Zeile 3 wird der Baustein aus dem Simulationsmodell ausgelesen, indem die Funktion `getParameter` auf der Referenz des Simulationsmodells aufgerufen wird. Letzteres ist insbesondere möglich, da der geforderte Baustein im initialen Simulationsmodell als Parameter mit dem Namen *configSetup* vorhanden ist. Ferner wird in derselben Zeile die Rückgabe der Funktion in der Variable `configSetup` gespeichert. Diese wird in der Zeile 4 insofern überprüft, als diese nicht den Wert Null hat und dem Datentypen `ConfigSetup` entspricht. Letzteres ist notwendig, da die Funktion ein Objekt des Datentyps `Object` zurückgibt, das in die entsprechende Datenstruktur überführt werden muss. Sofern die Bedingung in der

Zeile 4 erfüllt ist, folgt in der darauffolgenden Zeile 5 die Überführung in die Datenstruktur `ConfigSetup` über den Aufruf der Scala-Funktion `asInstanceOf`. Ebenso wird in dieser Zeile der ausgelesene und überführte Baustein in die umhüllende Datenstruktur eingehüllt und als Produktionsergebnis zurückgeben. Sofern die Rückgabe der Funktion `getParameter` einem Nullwert oder keiner Instanz des Datentyps `ConfigSetup` entspricht, wird in der Zeile 7 eine leere Umhüllung zurückgegeben.

```
1  @combinator object configSetupComponent {
2    def apply: ConfigSetupWrapper = {
3      val configSetup = this.rootAgent.getParameter("configSetup")
4      if (configSetup != null && configSetup.isInstanceOf[ConfigSetup])
5        ConfigSetupWrapper(configSetup.asInstanceOf[ConfigSetup])
6      else
7        ConfigSetupWrapper.empty
8    }
9    val semanticType: Type = SemanticTypes.ConfigSetup
10 }
```

Listing 7.3: Implementierung einer Komponente zur Produktion des Bausteins `ConfigSetup`, der über die Referenz des Simulationsmodells abgerufen wird, und in ein Wrapper-Objekt verpackt wird. Sofern der Baustein nicht abgerufen werden kann, wird ein leeres Wrapper-Objekt produziert.

Als Nächstes wird die Komponente *workerHomeComponent* vorgestellt, die einen Baustein der AnyLogic 8-Bibliothek von Grund auf erzeugt. Die Implementierungsdetails sind in Listing 7.4 dargestellt. Den nativen Zieltypen dieser Komponente bildet die Datenstruktur `PointNode` (vgl. Zeile 2) der AnyLogic 8-Bibliothek. In der Zeile 3 wird eine Instanz dieser Klasse durch den Aufruf des Konstruktors erzeugt. Die darauffolgenden Zeilen 4 bis 6 rufen unterschiedliche Methoden dieser Klasse auf, die insbesondere die Positionierung und Visualisierung des Punktknotens festlegen. In der Zeile 7 wird das erzeugte und angepasste Objekt zurückgegeben.

```
1  @combinator object workerHomeComponent {
2    def apply: PointNode = {
3      val workerHomePointNode = new com.anylogic.engine.markup.PointNode()
4      workerHomePointNode.setPos(90, 400)
5      workerHomePointNode.setLineColor(Color.RED)
6      workerHomePointNode.setRadius(10)
7      workerHomePointNode
8    }
9    val semanticType: Type = SemanticTypes.WorkerHome
10 }
```

Listing 7.4: Implementierung einer Komponente zur Produktion des Punktknotens (`PointNode`), der die Positionierung der Werker im Simulationsmodell verantwortet.

Im Folgenden wird eine exemplarische Komponente vorgestellt, die eine Liste mit Agenten, die Maschinenzellen repräsentieren, produziert. Hierzu wird die Komponente *cuttingMachinesComponent* vorgestellt, deren Implementierungsdetails in Listing 7.5 dargestellt sind. Die Komponente produziert die Agenten zur Repräsentation von Maschinenzellen mit 2D-Laserschneidemaschinen. Die Komponente erwartet ein Objekt des Datentyps `Context` (vgl. Zeile 2) sowie die Konfiguration in Form eines Objekts des Datentyps `InputData` (vgl. Zeile 3) als Argumente. Die Implementierung dieser Komponente charakterisiert sich durch zwei Schritte:

Im ersten Schritt wird die Konfiguration geladen, wobei ausschließlich die Einträge hinsichtlich 2D-Laserschneidemaschinen berücksichtigt werden. Hierzu wird die Funktion `getMachinesOfType(...)` der Klasse `InputData` aufgerufen, welche die geforderten Einträge in der Konfiguration selektiert. Die gefilterten Einträge werden in einer Listendatenstruktur der Programmiersprache Java zurückgegeben, da die Klasse `InputData` als eine Java-Klasse in der Simulationsumgebung implementiert ist. Angesichts dessen wird die Scala-Funktion `asScala` in der Zeile 4 und 5 aufgerufen, um die Java-Datenstruktur in eine Scala-Datenstruktur zu konvertieren, sodass die Liste in den Implementierungsdetails der Komponente verarbeitet werden kann.

Im zweiten Schritt wird jeder Maschineneintrag `machineEntry` der konvertierten Liste iteriert. Für jeden Maschineneintrag der Konfiguration wird in der Zeile 16 eine Maschinenzelle mittels der Fabrikmethode `createCuttingMachineCell(...)` erzeugt. Die benötigten Argumente zum Aufruf dieser Methode werden aus dem Maschineneintrag und dem `Context`-Objekt gelesen. So wird in der Zeile 8 der Maschinenname über den Aufruf der entsprechenden `get`-Methode des Maschineneintrags ausgelesen und in der Variable `machineName` zwischengespeichert. Die Referenzen auf die Lagertürme werden in den Zeilen 10 und 11 Zeile ebenfalls in entsprechende Variablen festgehalten. Hierfür werden mittels der Aufrufe der Funktionen `getSource()` und `getTarget()` die zugewiesenen Türme genauer gesagt deren Nummerierung ermittelt. Anschließend werden diese Nummern als Argumente der Funktion `getTower` des `Storage`-Objekts verwendet, um die geforderten Referenzen auf die Lagerturm-Objekte zu erhalten. Die verbleibenden Argumente der Fabrikmethode werden ebenso ausgelesen und in Variablen zwischengespeichert, ehe diese zur Erzeugung des Maschinenzellen-Agenten in der Zeile 16 verwendet werden. Dieses Vorgehen wird für sämtliche Maschineneinträge der Konfiguration wiederholt.

Das Listing 7.6 zeigt die Implementierung der Top-Level-Komponente, die den Kontext, Agenten zur Repräsentation von Maschinenzellen und die Konfiguration bündelt. Letztere erwartet diese als Argumente (vgl. `apply`-Methode in den Zeilen 2 bis 6 sowie semantische Typspezifikation in Zeile 34). In der Implementierung dieser Komponente werden in der Zeile 8 die synthetisierten Agenten in die Liste `allAgents` kopiert, die zur Positionierung der Maschinen (vgl. Zeile 11) und Erzeugung der Pfade (vgl. Zeile 12) in der Visualisierung verwendet wird. Die Positionierung der Maschinen erfolgt mittels der ausgelagerten Funktion `createMachineNodes`, während die Pfade durch die Funktion `createPaths` erzeugt werden. In der Zeile 15

```
1  @combinator object cuttingMachineCellsComponent {
2    def apply(context: Context,
3              inputData: InputData): Seq[Agent] = {
4    CollectionConverters
5     .asScala(inputData.getMachinesOfType("2D-Lasercutting"))
6     .toSeq
7     .map { machineEntry => {
8       val machineName: String = machineEntry.getName
9       val storage: Storage = context.getStorage
10      val inputTower: StorageTower =
        ↪ storage.getTower(machineEntry.getSource)
11      val outputTower: StorageTower =
        ↪ storage.getTower(machineEntry.getTarget)
12      val worker: ResourcePool[Worker] = context.getWorker
13      val viewArea: ViewArea = context.getViewAreas("Lasercutting")
14      val configSetup = context.getConfigSetup
15
16      bbFactory.createCuttingMachineCell(machineName, storage, inputTower,
        ↪ outputTowerworker, viewArea, configSetup)
17    }
18   val semanticType: Type = SemanticTypes.Context :&:
     ↪ SemanticTypes.IsComplete =>: SemanticTypes.InputData =>:
     ↪ SemanticTypes.CuttingMachineBuildingBlock
19 }
```

Listing 7.5: Implementierung einer Komponente zur Produktion einer Liste mit Maschinenzellen-Agenten. Hierfür erhält die Komponente einen Kontext, der Referenzen auf Bestandteile des Simulationsmodells enthält, und die Eingaben der anwendenden Person als Argumente. Diese werden verwendet, um mittels einer Fabrik eine Liste mit den entsprechenden Agenten zu erzeugen (vgl. Zeile 16).

wird die Liste der Werker aus dem `Context`-Objekt entnommen und in der Zeile 16 folgt die visuelle Positionierung der (insgesamt zehn) Werker. Hierfür wird die synthetisierte Positionierung verwendet, die im `Context`-Objekt enthalten und über die Funktion `getWorkerHome` abgerufen werden kann. Über den Aufruf der Funktion `set_homeNodes` auf der Ressource der Werker wird die Positionierung vorgenommen.

In der Zeile 19 wird ein neues Netzwerk erzeugt, das die zuvor erzeugten Elemente der Visualisierung beinhaltet. Der Konstruktor des Netzwerk-Objekts erhält eine Referenz auf das initiale Simulationsmodell sowie den Namen des Netzwerks. Das Hinzufügen der erzeugten Pfade, Positionierungen der Maschinen sowie initialen Werkerpositionen folgt in den darauffolgenden Zeilen 20 bis 22. Danach wird in der Zeile 23 ein neues Level erzeugt. Letzteres entspricht einer Schicht in der Visualisierung von Simulationsmodellen in AnyLogic 8. Ebenso wie das Netzwerk erhält auch das Level eine Referenz auf das ursprüngliche Simulationsmodell

```
@combinator object simulationModelComponent {
  def apply(context: Context,
            cuttingMachines: Seq[Agent],
            bendingMachines: Seq[Agent],
            punchCuttingMachines: Seq[Agent],
            inputData: InputData): SynthesisResult = {

    val allAgents = cuttingMachines ++ bendingMachines ++
     ↪ punchCuttingMachines

    // create machine nodes and paths in simulation model
    val machineNodes: Seq[PointNode] = allAgents.map(createMachineNodes)
    val paths: Seq[Path] = allAgents.map(createPaths)

    // add worker home to resource pool
    val resourcePool: ResourcePool[Worker] = ctx.getWorker
    resourcePool.set_homeNodes((1 to 10).map(_ =>
     ↪ context.getWorkerHome).toArray)

    // add visualisation to simulation model
    val network = new Network(rootAgent.get, "WorkerPathsNetwork")
    paths.foreach(_ => network.addAll(_))
    machineNodes.foreach(_ => network.addAll(_))
    network.addAll(context.getWorkerHome)
    val level = new Level(rootAgent.get, "WorkerPathsLevel",
     ↪ ShapeDrawMode.SHAPE_DRAW_2D3D, 0)
    level.add(network)
    level.initialize()
    rootAgent.get.getPresentationShape.add(level)

    new SynthesisResult(context, cuttingMachines, bendingMachines,
     ↪ punchCuttingMachines)
  }

  val semanticType: Type = SemanticTypes.Context :&:
   ↪ SemanticTypes.IsComplete =>: SemanticTypes.LaserCuttingMachines =>:
   ↪ SemanticTypes.BendingMachines =>:
   ↪ SemanticTypes.PunchCuttingMachines =>: SemanticTypes.InputData =>:
   ↪ SemanticTypes.TopLevelComponent
}
```

Listing 7.6: Implementierung der Top-Level-Komponente zur Produktion der Bestandteile und den Listen, die die Maschinenzellen-Agenten enthalten. Hierfür erhält die Komponente die synthetisierten Agenten als Argumente. Ferner erzeugt die Komponente die Visualisierung für die synthetisierten Maschinenzellen. Das Ergebnis wird in einer Objektinstanzs des Datentyps `SynthesisResult` zurückgegeben und im Simulationsmodell verarbeitet.

sowie einen Namen. Ferner wird durch die Konstante `ShapeDrawMode.SHAPE_DRAW_2D` als drittes Argument des Konstruktors festgelegt, dass eine zwei-dimensionale Visualisierung gefordert ist. Das letzte Argument gibt die Positionierung der Schicht an, wobei der Wert 0 eine Positionierung auf der obersten Schicht anzeigt. Das zuvor erzeugte Netzwerk wird in der Zeile 24 der Schicht hinzugefügt, ehe die Schicht in der Zeile 25 initialisiert wird.

In der Zeile 26 wird die erzeugte Schicht der Präsentationsschicht des initialen Simulationsmodells hinzugefügt. An dieser Stelle erfolgt eine unmittelbare Modifikation des initialen Simulationsmodells. Abschließend wird in der Zeile 28 eine Objektinstanz des Datentyps `SynthesisResult` erzeugt, das die Elemente bündelt und als Produktionsergebnis zurückgegeben wird. Dieses Objekt wird in der Startroutine des initialen Simulationsmodells verarbeitet. Die Verarbeitung wird im nachfolgenden Abschnitt ausführlicher vorgestellt.

7.3.3 Synchronisation des Simulationsmodells und Ergebnisse der Synthese

Nach der Durchführung der komponentenbasierten Synthese wird das Ergebnis in Form eines Objekts der Datenstruktur `SynthesisResult` in der Startroutine des initialen Simulationsmodells verarbeitet. Hierbei werden die synthetisierten Agenten zur Repräsentation der Maschinenzellen den entsprechenden Populationen im initialen Simulationsmodell hinzugefügt. In Listing 7.7 ist das Hinzufügen von Maschinenzellen, die 2D-Laserschneidemaschinen repräsentieren, dargestellt.

```
1 List<Agent> cuttingMachines = synthesisResult.getCuttingMachineAgents();
2 cuttingMachines.forEach(currAgent ->
  ↪  currAgent.goToPopulation(cuttingMachinePopulation));
3 cuttingMachinePopulation.forEach(currMachineAgent -> {
4   currMachineAgent.updateWorkerProgrammingPosition());
5   currMachineAgent.set_workerPool(this.worker.get(0));
6 });
```

Listing 7.7: Verarbeitung des Synthesergebnisses in der Startroutine des Simulationsmodells, in dem die synthetisierten Agenten (Maschinenzellen zur Repräsentation von Schneidemaschinen) abgerufen (Zeile 1), der Population hinzugefügt (Zeile 2) und erneute Initialisierungen vorgenommen werden (Zeile 3 bis 6).

In der Zeile 1 werden die synthetisierten Agenten in Form einer Liste in der Variable `cuttingMachines` zwischengespeichert. Diese Liste wird in der Zeile 2 iteriert und für jeden Agenten wird die Methode `goToPopulation` aufgerufen, die das Hinzufügen eines Agenten zu einer Population realisiert. Die Variable `cuttingMachinePopulation` beinhaltet die Population der 2D-Laserschneidemaschinen im initialen Simulationsmodell und wird aufgrund dessen als Argument übergeben. In den Zeilen 3 bis 6 wird die Population iteriert, wobei in der Zeile 4 die Positionierung der Werker im aktuell iterierten Agenten aktualisiert und in der Zeile 5 die Werker-Ressource dem Agenten zugewiesen wird.

Daraufhin wird das Vorgehen wiederholt für die synthetisierten Agenten, die Maschinenzellen zur Repräsentation von Biege- und Stanz-Lasermaschinen darstellen. Sofern die Migration und die Synthese um weitere Maschinentypen erweitert wird, so muss neben der Komponentensammlung auch der Quellcode der Startroutine angepasst werden. Hierfür könnte ebenfalls die komponentenbasierte Synthese verwendet werden, sodass die Startroutine nicht händisch adaptiert werden müsste. Nachdem die synthetisierten Agenten den entsprechenden Populationen hinzugefügt wurden, folgt die erneute Initialisierung ausgewählter Prozessbausteine des Simulationsmodells (nicht in Listing 7.7 aufgeführt). Abschließend wird die Simulation gestartet.

7.3.4 Benutzeroberfläche zur Konfiguration der Generierung

Wie eingangs erwähnt, erfolgt die Konfiguration des zu synthetisierenden Fabriksystems über die Angabe der Anzahl der Maschinen und Lagertürme sowie der Zuordnung von Maschinen zu Lagertürmen. Die Datenstruktur `InputData` repräsentiert eine konkrete Konfiguration. Die Konfiguration erfolgt entweder durch den Einsatz einer grafischen Benutzeroberfläche oder wird automatisiert erzeugt. Die grafische Benutzeroberfläche ist in das initiale Simulationsmodell eingebettet und wurde vollständig innerhalb der Simulationsumgebung AnyLogic 8 entwickelt. Die Abbildung 7.7 zeigt den Aufbau dieser Benutzeroberfläche. So ist im oberen Bereich die Festlegung der Anzahl der Lagertürme möglich, während der untere Bereich die Konfiguration der Maschinen erlaubt. Hierfür kann zunächst die Anzahl der Maschinen über einen Schieberegler festgelegt werden, ehe für jede Maschine ein Name und Maschinentyp sowie die entsprechenden Lagertürme ausgewählt wird.

Die Konfigurationsmöglichkeiten entsprechen den erforderlichen Informationen der Datenstrukturen `InputData` und `MachineEntry`. Bei einem Klick auf den *Play*-Knopf werden Instanzen dieser Datenstrukturen erzeugt, die der Konfiguration entsprechen. Ferner wird die komponentenbasierte Synthese angestoßen. Die grafische Benutzeroberfläche ist somit vor der Ausführung des initialen Simulationsmodells geschaltet.

Bei Verwendung der Experimentierfunktion, die im Rahmen der Durchführung der Experimente in diesem Anwendungsfall verwendet wird, erfolgt eine automatisierte Generierung der Konfiguration. Letztere erlaubt die Festlegung einer Mindest- und Maximalanzahl an Maschinen. Mittels eines Skriptes werden sämtliche Kombinationen generiert und für jede dieser Kombination wird ein entsprechendes `InputData`-Objekt erzeugt. Die Anzahl der Lagertürme wird in Abhängigkeit zur Anzahl der Maschinen festgelegt. Weiterhin erfolgt die Zuordnung der Maschinen zu den Lagertürmen zufällig. Für jede Kombination wird die komponentenbasierte Synthese des entsprechenden Simulationsmodells sowie die Ausführung und Auswertung des Simulationsmodells automatisiert gestartet. Nach jedem Simulationslauf werden die Ergebnisse in eine Datenbank übertragen, die nach der Ausführung sämtlicher Läufe einen Vergleich der Konfigurationen ermöglicht.

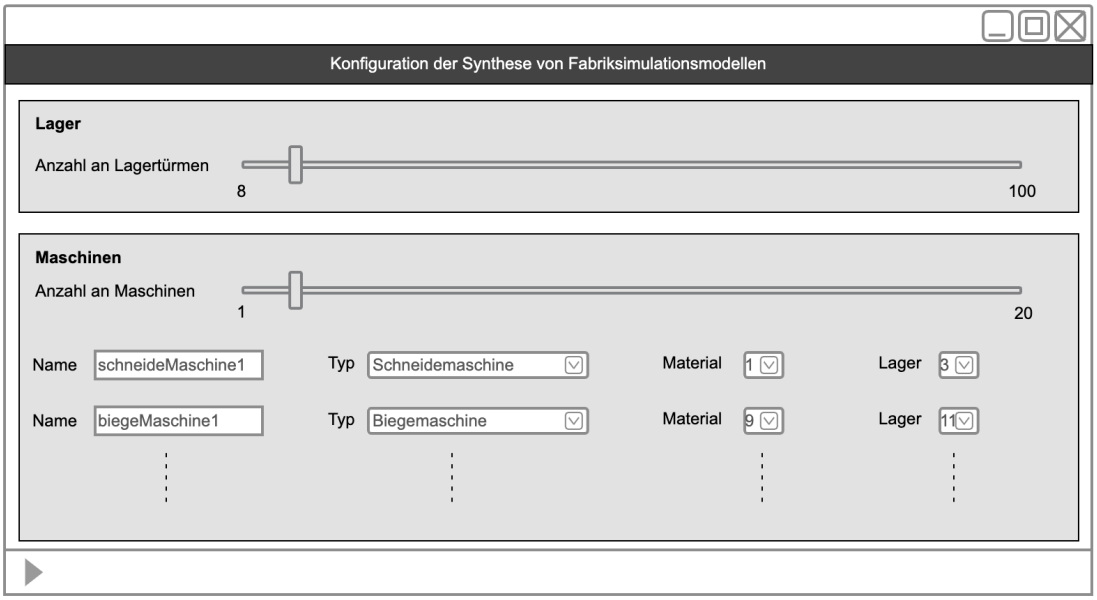


Abbildung 7.7: Skizze der grafischen Benutzeroberfläche zur Konfiguration der komponentenbasierten Synthese eines Simulationsmodells eines Shop-Floor-Systems. Die Benutzeroberfläche ist Teil des Template-Simulationsmodells und erlaubt die Festlegung der Anzahl an Lagertürmen sowie die Auswahl der zu synthetisierenden Maschinen und dazugehörigen Lagertürmen.

7.4 Durchführung von Experimenten

In diesem Abschnitt wird der vorgestellte Lösungsansatz verwendet, um sämtliche Konfigurationen des Shop-Floor-Systems gemäß dem Feature-Diagramm aus der Abbildung 7.2 zu synthetisieren und zu vergleichen. Wie bereits erwähnt, werden in diesem Zusammenhang die Konfigurationen automatisiert generiert, sodass nicht jede Variante in der grafischen Benutzeroberfläche händisch konfiguriert werden muss. Die Generierung der Konfigurationen erfolgt mithilfe der Experimentierfunktion der Simulationsumgebung AnyLogic 8, die eine automatisierte Parametervariation erlaubt. Die Variation der Parameter wird verwendet, um die unterschiedlichen Konfigurationen zu erstellen. Die Tabelle 7.2 zeigt die Parameter sowie deren Wertebereich.

Die ersten drei Parameter beziehen sich auf die Anzahl der Maschinen, während der letzte Parameter Bezug auf die Anzahl der Lagertürme in einer Konfiguration nimmt. Die Anzahl der Lagertürme wird anhand der Anzahl der Maschinen durch ein Skript determiniert. Durch den letzten Parameter werden zudem Konfigurationen mit einem 20 % größerem Lagersystem untersucht, sodass der Einfluss unterschiedlicher Lagersystemgrößen ausgewertet werden kann. Für jede Parametervariation wird eine Instanz des Datentyps `InputData` inklusive der Maschineneinträge des Datentyps `MachineEntry` erzeugt und der CLS-Implementierung als Argument übergeben.

Parameter	Mindestanzahl	Maximalanzahl	Schrittgröße
Anzahl Schneidemaschine	2	5	1
Anzahl Biegemaschine	2	5	1
Anzahl Stanz-Lasermaschine	2	5	1
Faktor Anzahl Lagertürme	1.0	1.2	0.2

Tabelle 7.2: Parameter im Simulationsmodell, die zur automatisierten Generierung der Konfiguration der komponentenbasierten Synthese eingesetzt werden. Diese werden von der Simulationsumgebung automatisiert variiert, wobei die Mindest- und Maxmimalanzahl sowie die Schrittgröße von der anwendenden Person festgelegt werden.

Shop-Floor-Konfiguration				Auslastungen			
Stanz-schneide-maschine	Biege-maschi-ne	Schneide-maschi-ne	Anzahl Lager-türme	Stanz-schneide-maschine	Biege-maschi-ne	Schneide-maschi-ne	RBG
2	2	2	48	52.8 %	80.0 %	71.2 %	72.2 %
2	2	2	58	49.9 %	83.7 %	70.0 %	73.2 %
3	4	2	72	47.3 %	69.8 %	66.0 %	78.7 %
3	4	2	86	51.2 %	69.5 %	62.9 %	80.3 %
5	5	5	120	30.1 %	61.5 %	45.7 %	91.6 %
5	5	5	144	30.9 %	62.8 %	51.9 %	93.0 %

Tabelle 7.3: Zufällige Auswahl der Ergebnisse der Simulationsläufe der 128 synthetisierten Konfigurationen eines Shop-Floor-Systems. In der linken Hälfte ist die jeweilige Konfiguration dargestellt, während die rechte Hälfte die Ergebnisse abbildet, die sich auf die Auslastung der Maschinen und des Regalbediengeräts (RBG) beziehen.

Wie zuvor erwähnt, erfolgt die Zuordnung der Maschinen zu den Lagertürmen sowie die Sicherung der Simulationsergebnisse in einer Datenbank ebenfalls automatisiert. Weiterhin sei zu erwähnen, dass im Rahmen dieses Experiments ein fiktives Auftragsportfolio simuliert wird, das eine feste Anzahl an Aufträgen, unabhängig von der Anzahl der Maschinen, umfasst. Die Aufträge beschreiben hierbei eine Abfolge von zu durchlaufenden Maschinen. Die Auswertung eines Simulationslaufes erfolgt durch die Betrachtung entsprechender Kennzahlen der einzelnen Maschinen und des Regalbediengeräts. Auch in diesem Anwendungsfall sei darauf hinzuweisen, dass das durchgeführte Experiment keinen Anspruch auf einen Erkenntnisgewinn in Bezug auf logistische Fragestellungen beabsichtigt, sondern die Funktionsfähigkeit des Ansatzes demonstrieren möchte. Ferner sind die simulierten Aufträge fiktiv und repräsentieren somit kein realitätsgetreues Auftragsvorkommen eines Shop-Floor-Systems. Daher stellt eine detaillierte Analyse und eine Optimierung hinsichtlich ausgewählter Zielfunktionen den Forschungsgegenstand zukünftiger Arbeiten dar.

Gleichwohl zeigt die Tabelle 7.3 exemplarisch die Auswertung einzelner Simulationsläufe. So werden die einzelnen Auslastungen der Maschinen sowie des Regalbediengeräts verglichen,

wobei bei den Auslastungen der Durchschnitt aller Maschinen betrachtet wird. Jede Konfiguration wurde genau ein Mal simuliert. Dies ist ausreichend, da das Simulationsmodell keinen stochastischen Effekten unterliegt und sich somit unterschiedliche Simulationsläufe nicht unterscheiden. Die Ergebnisse der Simulationsläufe zeigen, dass die verschiedenen Konfigurationen einen Einfluss auf die Maschinenauslastung haben. Insbesondere ist ersichtlich, dass eine höhere Anzahl an Maschinen zu einer geringeren Maschinenauslastung führen, da jede Maschine insgesamt weniger Aufträge bearbeitet. Die komponentenbasierte Synthese einer einzelnen Konfiguration benötigte im Schnitt drei Sekunden und die Simulation (einschließlich der Synthese) sämtlicher 128 Konfigurationen dauerte etwa sieben Minuten. Die Synthese und die Experimente wurden auf einem Standrechner, der mit einem 64 GB großen Arbeitsspeicher und einem Intel i7-Prozessor ausgestattet ist, durchgeführt.

7.5 Diskussion

Eine offene Herausforderung stellen die initialen Kosten der Erstellung der Komponentensammlung dar, da sich diese Kosten häufig erst bei einer hohen Variabilität einer Problemstellung amortisieren. Daher sollte für jeden Anwendungsfall kritisch hinterfragt werden, ob diese tatsächlich den Einsatz der komponentenbasierten Synthese erfordert oder die manuelle Erstellung der Simulationsmodellvarianten die kostengünstigere Alternative darstellt. Jedoch profitieren Problemstellungen mit einer besonders hohen Variabilität von der Skalierbarkeit der komponentenbasierten Synthese. Dies begründet sich in dem komponentenbasierten Ansatz, der die Komplexität einer Problemstellung auf eine Vielzahl von Komponenten aufteilt. Insbesondere im Vergleich zu generischen Simulationsmodellen stellt die Skalierbarkeit einen erheblichen Vorteil dar, da bei generischen Simulationsmodellen mit einer zunehmender Anzahl an Varianten auch die Komplexität des Simulationsmodells steigt. Dies wiederum wirkt sich auf die Qualität des Simulationsmodells und damit auch auf die der Simulationsstudie aus [128].

Eine weitere Herausforderung betrifft die Integration des Syntheseframeworks CLS in einer Simulationsumgebung zur internen Generierung der Simulationsmodelle. Hierbei kann festgehalten werden, dass der Aufwand dieser Integration von den vorhandenen Schnittstellen einer Simulationsumgebung abhängig ist. So zeigte sich beispielsweise, dass das dynamische Hinzufügen von Agenten in AnyLogic 8 nicht direkt möglich ist, jedoch über die Verwendung von Populationen. Auch in der Arbeit von Lechler et al. [68] wurden fehlende Schnittstellen in AnyLogic 8 als eine herausfordernde Limitierung genannt. Andererseits können in der Simulationsumgebung AnyLogic 8 externe Software-Bibliotheken (wie die Scala-Implementierung) ohne großen Aufwand eingebunden werden, da die Simulationsumgebung in der Java Virtual Machine betrieben wird. Allerdings müssen die externen Bibliotheken auch in der Java Virtual Machine lauffähig sein. Im allgemeinen Vergleich zum externen Generierungsansatz erwies sich die Vermeidung von Abhängigkeiten zu proprietären Formaten von Simulationsumgebungen als ein wesentlicher Vorteil. Ebenso stellt die für die anwendende Person vollkommen

abstrahierte komponentenbasierte Synthese einen weiteren Vorteil dar, da hierdurch keine Grundkenntnisse bezüglich der kombinatorischen Logiksynthese vorausgesetzt werden. Dies ist insofern relevant, als fabrikplanende Personen häufig über keinen IT-Hintergrund verfügen. Lediglich die Simulationsfachkraft benötigt Kenntnisse der komponentenbasierten Synthese, da diese für die Erstellung der Komponentensammlung verantwortlich ist. Jedoch verfügen Simulationsfachkräfte in der Regel über Programmierkenntnisse, da die Programmierung ein häufiger Bestandteil bei der Modellierung von Simulationsmodellen darstellt.

7.6 Zusammenfassung und Ausblick

Der Lösungsansatz dieses Anwendungsfalls demonstriert die Eignung der komponentenbasierten Softwaresynthese zur Migration eines komplexen Simulationsbaukastens in eine Produktlinie zur Synthese von Simulationsmodellen. Insbesondere die graduelle Migration ermöglicht die Synthese von Simulationsmodellen, ohne dass ein Simulationsbaukasten oder Simulationsmodell vollständig migriert werden muss. Dieser Umstand reduziert die benötigte Zeitspanne bis zur Erzielung erster Ergebnisse. Ferner erlaubt der interne Generierungsansatz die Abstrahierung der komponentenbasierten Synthese für die anwendenden Personen und ermöglicht als Folge dessen eine Verwendung des Ansatzes ohne tiefgreifende Kenntnisse bezüglich dieser.

Der Forschungsgegenstand zukünftiger Arbeiten könnte die Automatisierung der Migration fokussieren. Dabei stellen die Erstellung der Agenten zur Repräsentation von Maschinenzellen sowie die Komponentisierung dieser Agenten denkbare Automatisierungspunkte dar. Dies könnte durch eine komponentenbasierte Synthese der Agenten realisiert werden, in dem die Bausteine der Maschinen, der Automatisierungseinheiten und weiterer Logiken in Komponenten überführt werden. Mittels des Typsystems könnten dann die zulässigen Anschlüsse eines Maschinenbausteins beschrieben werden, die durch entsprechende Komponenten produziert werden. Damit könnten unterschiedliche Varianten einer Maschinenzelle synthetisiert werden, so etwa Maschinen mit einer automatisierten oder manuellen Be- und Entladung. Die synthetisierten Maschinenzellen könnten dann automatisiert von zusätzlichen Komponenten produziert werden, die im Rahmen der Synthese einer Variante eines Shop-Floor-Systems eingesetzt werden. Die Automatisierung der Erstellung der Komponenten würde eine Reduktion der initialen Kosten erlauben. Weiterhin könnte im Kontext weiterer Forschung eine vollständige Migration des Simulationsbaukastens in Betracht gezogen werden oder eine Simulation mit realen Daten durchgeführt werden.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Dissertation wird die komponentenbasierte Softwaresynthese zur Generierung von diskreten, ereignisorientierten Simulationsmodellen verwendet, die sich hinsichtlich ihrer Strukturen und Kontrollstrategien unterscheiden. Hierfür wurde eine Technologie entwickelt, die es erlaubt, bereits existierende Simulationsmodelle in eine Produktlinie von Simulationsmodellen zu migrieren. Diese Technologie umfasst ein systematisches Vorgehen zur Aufteilung eines Simulationsmodells in Komponenten und zur Typisierung dieser. Die gewählten Typspezifikationen dieser Komponenten erlauben eine Abbildung der logischen Abhängigkeiten eines zu synthetisierenden Simulationsmodells. Ferner umfasst die Technologie eine Auswahl an möglichen Anpassungen an der Komponentensammlung, welche die Markierung von Variabilitätspunkten in einem Simulationsmodell durch die anwendende Person ermöglicht. Die Technologie erlaubt die Migration von ereignisorientierten und agentenbasierten Simulationsmodellen der Simulationsumgebung AnyLogic 8 über eine bedienungsfreundliche Benutzungsoberfläche. Die Technologie wurde mittels der Implementierung anhand mehrerer Anwendungsfälle getestet, in dem bereits existente Simulationsmodelle in Produktlinien migriert wurden. Hierbei fokussierte jeder Anwendungsfall einen separaten Schwerpunkt, wie das Filtern mittels Techniken des SMT-Constraint-Solvings, das automatisierte Erstellen einer Komponentensammlung oder die Integration der komponentenbasierten Synthese innerhalb eines Simulationsmodells. In allen Anwendungsfällen konnte gezeigt werden, dass die Technologie eine Unterstützung in der Entscheidungsfindung darstellte, insofern als eine größere Anzahl an Lösungsvarianten betrachtet werden konnte, als händisch mit wirtschaftlich vertretbarem Rahmen implementiert werden konnten. Dennoch folgt nachfolgend eine kritische Reflexion des Vorgehens, ehe Anknüpfungspunkte für weitere Forschungsarbeiten vorgestellt werden.

8.1 Kritische Reflexion

Im Rahmen einer kritischen Auseinandersetzung mit den Ergebnissen dieser Dissertation kann zunächst die Wahl der komponentenbasierten Softwaresynthese zur Generierung der Simulationsmodelle kritisch hinterfragt werden. Hierbei kann überprüft werden, ob ein modellbasierter Ansatz eine geeignetere Methode darstellt, da dieser ebenso die Abbildung von Strukturvarianz erlaubt und sich die Verwendung durch eine anwendende Person aufgrund der Abstraktion in Form eines Modells möglicherweise intuitiver gestaltet. Jedoch besteht die Herausforderung bei diesem Ansatz darin, dass ein Modell von Grund auf erstellt werden muss. Dies stellt auch einen Vorteil des Vorgehens aus dieser Dissertation gegenüber dem modellbasierten Ansatz dar, denn es muss kein abstrahiertes Modell erarbeitet werden, sondern ein bestehendes Simulationsmodell bildet die Ausgangsbasis. Damit kann zum einen, die gewohnte Umgebung (z. B. eine Simulationsumgebung) zur Modellierung der Simulationsmodelle verwendet werden, und zum anderen kann eine iterative Aufteilung in Komponenten und infolgedessen schrittweise Migration erfolgen. Letzteres stellt einen erheblichen Vorteil im Vergleich zum modellbasierten Ansatz dar, denn die graduelle Migration ermöglicht zu jeder Zeit die Synthese von ausführbaren Simulationsmodellen, während bei einem modellbasierten Ansatz zunächst ein Großteil des abstrahierenden Modells fertiggestellt werden müsste. Ferner ist der Ansatz der komponentenbasierten Softwaresynthese sehr nah am Quellcode. Damit können bereits existierende Quellcodefragmente mit einem überschaubarem Aufwand in die Implementierungsdetails von Komponenten integriert werden. Ferner kann bei der Einbindung von externen Bibliotheken und deren Verwendung in den Implementierungsdetails der Komponenten auf diese Bibliotheken zugegriffen werden. Beispielsweise konnten im dritten Anwendungsfall so Objekte zur Visualisierung von Bestandteilen eines Simulationsmodells instantiiert und mittels entsprechender Funktionen einer Bibliothek angepasst werden.

Ein wesentlicher Bestandteil der Modellierung von Simulationsmodellen wird in dieser Dissertation nicht behandelt. Dieser umfasst die Verifikation und Validierung der synthetisierten Simulationsmodelle. Der Prozessschritt verhält sich ähnlich zu den Tests in der Softwareentwicklung, bei denen überprüft wird, ob eine entwickelte Software den festgelegten Anforderungen entspricht. Die Verifikation überprüft, ob ein Simulationsmodell sich gemäß den Erwartungen der modellierenden Person verhält [67]. Beispielsweise kann die Verifikation dadurch erfolgen, dass überprüft wird, ob das in einer Quelle erzeugte Objekt die Senke in einem Simulationsmodell erreichen kann. Die Validierung hingegen überprüft ein Simulationsmodell hinsichtlich der Übereinstimmung des modellierten Systems mit dem realen System [67]. Letzteres ist in Bezug auf die Planung von Systemen der Produktion und Logistik herausfordernd, da diese System zum Planungszeitpunkt noch nicht erbaut wurden. Son et al. [113] validieren ein Simulationsmodell eines noch nicht existenten Systems, in dem sie die Ergebnisse unterschiedlicher Simulationsläufe miteinander vergleichen. Bei einer deterministischen Parametrisierung erwartet die Autorenschaft etwa identische Ergebnisse. In zukünftigen Forschungsarbeiten sollte die Validierung und Verifikation der synthetisierten Simulationsmodelle automatisiert erfolgen. Damit kann gewährleistet werden, dass die

synthetisierten Varianten des Simulationsmodells auch valide Lösungen darstellen. Diese könnten beispielsweise in Form von weiteren Komponenten implementiert werden, die vor der Produktion eines Simulationsmodells die Verifikation und Validierung übernehmen.

An dem Vorgehen aus dieser Dissertation kann kritisch angemerkt werden, dass es erst bei einer gewissen Variabilität rentabel ist. Sofern die Anzahl der möglichen Varianten nicht zu groß ist, so ist die manuelle Erstellung der Varianten sicherlich kostengünstiger. Dafür skaliert das Vorgehen hervorragend durch den komponentenorientierten Ansatz.

8.2 Weiterführende Arbeiten

Nachfolgend werden offene Forschungspunkte vorgestellt, die in Anknüpfung an den Ergebnissen dieser Dissertation untersucht werden können. Während in dieser Dissertation die Phase der Modellierung eines Simulationsmodells durch die komponentenbasierte Synthese automatisiert wird, kann diese ebenso für den vorgelagerten Schritt des Sammelns und Aufbereitens von Daten verwendet werden. Nach Trybula [119] nimmt diese Phase bis zu 40 % der Zeit einer Simulationsstudie ein. Hierbei kann die komponentenbasierte Synthese eingesetzt werden, um Skripte zu synthetisieren, welche die Verbindung zu den Schnittstellen herstellen und die entsprechenden Daten aus den Datenquellen auslesen. Dies erfolgt beispielsweise im Rahmen einer Abschlussarbeit in [105], in der Systeme zur Berechnung profitabler Touren für Handelsvertreter synthetisiert werden. Hierbei beeinflussen die entwickelten Komponenten unter anderem die Selektion der Daten aus Enterprise-Ressource-Planning und Customer-Relationship-Management Systemen. Auch ist es denkbar, die Phase der Auswertung der Simulationsergebnisse mittels der komponentenbasierten Softwaresynthese zu automatisieren. Hierfür könnten Skripte synthetisiert werden, die eine Auswertung der Ergebnisse in Form einer visuell aufbereiteten Dokumentation produzieren.

Ein weiterer zu untersuchender Forschungspunkt betrifft die Art und Weise der Markierung der Variabilitätspunkte. In dieser Dissertation wird bereits eine Vereinfachung des Prozesses angestrebt. Dennoch erfordert die Markierung der Variabilitätspunkte im ersten und zweiten Anwendungsfall ein grundlegendes Verständnis für den komponentenorientierten Ansatz. Daher sollte in Zukunft untersucht werden, ob eine Abstraktionsschicht zur vereinfachten Beschreibung der Variabilitätspunkte entwickelt werden sollte. Diese Abstraktionsschicht könnte in Form einer XML-Datei implementiert werden und Informationen über die Prozessbausteine und Kompositionen dieser umfassen. Beispielsweise könnte so festgelegt werden, ob ein Prozessbaustein wahlweise ausgelassen werden soll. Anschließend könnte die Datei dazu verwendet werden, die notwendigen Anpassungen an der Komponentensammlung automatisiert durchzuführen. Ferner könnte auch diese Datei synthetisiert werden, um eine automatisierte Markierung der Variabilitätspunkte und damit auch die Bildung von Varianten realisieren.

Weiterhin können die Ergebnisse dieser Dissertation als Ausgangspunkt für Forschungsarbeiten verwendet werden, die eine Schleife nach dem *Generieren, Testen und Lernen*-Muster

realisieren [100]. In dieser Schleife werden die Simulationsmodelle mittels des Vorgehens aus dieser Dissertation synthetisiert und anschließend automatisiert ausgeführt. Die Ergebnisse dieser Ausführung werden ausgewertet und zur Bewertung einer Lösungsvariante verwendet. Anschließend wird diese Bewertung verwendet, um mit dem dazu gewonnenen Wissen die Synthese weiterer Varianten durchzuführen. Beispielsweise kann in einem Simulationsmodell, das eine Produktionsstätte mit Maschinen abbildet, die Anzahl der Maschinen erhöht werden. Sofern das Ergebnis des synthetisierten Simulationsmodells, das eine höhere Anzahl an Maschinen enthält, zufriedenstellender als das ursprüngliche Simulationsmodell ist, so kann daraus geschlossen werden, dass die Erhöhung der Maschinenanzahl zielführend war. Dementsprechend könnte in einer weiteren Iteration die Anzahl der Maschinen weiter erhöht und das Ergebnis des synthetisierten Simulationsmodells erneut bewertet werden. Diese Schleife wird so lange wiederholt, bis entweder zufriedenstellende Ergebnisse der Simulationsläufe erzielt oder eine maximale Anzahl an Iteration durchlaufen wurde. Die Schleife des Generierens, Testens und Lernens stellt somit eine Methode dar, welche die komponentenbasierte Synthese von Simulationsmodellen im Kontext des maschinellen Lernens einbindet.

In diesem Zusammenhang ist eine Forschungsarbeit von Schäfer et al. [104] zu nennen, bei der statistische und lernende Komponenten im Kontext der komponentenbasierten Softwaresynthese verwendet werden, um eine systematische Exploration des Lösungsraums zu realisieren. Dabei bilden die komponentenbasierte Synthese von Programmen durch das Syntheseframework CLS und die Evaluation dieser generierten Lösungen, mit einem numerischen Vektor als Ausgabe, eine Blackbox-Funktion. Letztere wird von einem Mehrzieloptimierungstool verwendet, sodass eine automatisierte, lernende Suchprozedur realisiert wird. Der Ansatz wurde experimentell im Rahmen der Synthese von Motion Planning Programmen validiert. Jedoch kann der Ansatz ebenso auf den Forschungsgegenstand dieser Dissertation oder weiteren Anwendungsgebieten übertragen werden.

Ein weiteres mögliches Einsatzgebiet der komponentenbasierten Synthese stellt die Synthese von Multi-Level-Simulationen dar. Diese Form der Simulation ermöglicht die Simulation von Systemen mittels Simulationsmodellen unterschiedlicher Simulationsmethodiken, die durch den Austausch von Daten miteinander kommunizieren. Insbesondere im Kontext der Produktion und Logistik stellt diese Form der Simulation ein relevantes Werkzeug dar, da oftmals unterschiedliche Simulationsmethodiken zum Einsatz kommen. Prozesse werden etwa innerhalb einer Schneidemaschine durch eine andere Simulationsmethodik als die Warenflüsse im übergeordneten Job-Shop simuliert. Hinsichtlich der Multi-Level-Simulation ergeben sich mehrere Anknüpfungspunkte. So kann unter anderem die Logik zur Kopplung und zum Datenaustausch der unterschiedlichen Simulationsmodelle synthetisiert werden. Ferner kann der Ansatz auf die Synthese weiterer Simulationsmethodiken erweitert werden, um die ganzheitliche Generierung einer Multi-Level-Simulation zu ermöglichen. In einem ersten Schritt können insbesondere Übersetzer für weitere proprietäre Formate anderer prozessbasierter Simulationsumgebungen entwickelt werden. Ferner kann für die Simulationsumgebung AnyLogic 8 die Synthese der Logik eines Agenten durch *State-Charts* implementiert werden. Letztere unterscheiden sich hinsichtlich der Kontrollflussgraphen insofern als die Logik durch

Zustände modelliert werden.

Listingsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] VDI-Fachbereich Fabrikplanung und -betrieb. *VDI 3633 Blatt 1 - Simulation von Logistik-, Materialfluss- und Produktionssystemen - Grundlagen*. 2014. 48 S.

[2] Budi Arief und Neil Speirs. “A UML Tool for an Automatic Generation of Simulation Programs”. In: *Proceedings of the second international workshop on Software and performance - WOSP ’00*. the second international workshop. Ottawa, Ontario, Canada: ACM Press, 2000, S. 71–76. ISBN: 978-1-58113-195-6. DOI: 10.1145/350391.350408.

[3] Dieter Arnold, Hrsg. *Intralogistik: Potentiale, Perspektiven, Prognosen*. Vdi. Berlin: Springer, 2006. 296 S. ISBN: 978-3-540-29657-7.

[4] Jochen Baier, Harald Ruf und Henning Hill. “Verknüpfung von Materialflusssimulation und Planungsdatenbanken: Verringerung der Projektlaufzeit bei Simulationsstudien durch automatisierten Modellaufbau”. In: *Zeitschrift für wirtschaftlichen Fabrikbetrieb* 101.1 (24. Feb. 2006), S. 70–76. ISSN: 2511-0896. DOI: 10.3139/104.100995.

[5] Branislav Bako und Pavol Božek. “Trends in Simulation and Planning of Manufacturing Companies”. In: *Procedia Engineering* 149 (2016), S. 571–575. ISSN: 18777058. DOI: 10.1016/j.proeng.2016.06.707.

[6] Jerry Banks und John S. Carson. “Applying the Simulation Process”. In: *Proceedings of the 20th conference on Winter simulation*. Wsc ’88. New York, NY, USA: Association for Computing Machinery, 1. Dez. 1988, S. 52–55. ISBN: 978-0-911801-42-2. DOI: 10.1145/318123.318152.

[7] Panagiotis Barlas und Cathal Heavey. “Automation of Input Data to Discrete Event Simulation for Manufacturing: A Review”. In: *International Journal of Modeling, Simulation, and Scientific Computing* 07.1 (1. März 2016), S. 1630001. ISSN: 1793-9623. DOI: 10.1142/s1793962316300016.

[8] Clark Barrett, Pascal Fontaine und Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Techn. Ber. Department of Computer Science, The University of Iowa, 2017.

[9] Clark Barrett und Cesare Tinelli. “Satisfiability Modulo Theories”. In: *Handbook of Model Checking*. Hrsg. von Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith und Roderick Bloem. Cham: Springer International Publishing, 2018, S. 305–343. ISBN: 978-3-319-10574-1 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11.

[10] Lindsay Bassett. *Introduction to JavaScript Object Notation: A To-the-Point Guide to JSON*. Ö'Reilly Media, Inc.", 5. Aug. 2015. 126 S. ISBN: 978-1-4919-2945-2.

[11] Soeren Bergmann, Alexander Fiedler und Steffen Straßburger. "Generierung und Integration von Simulationsmodellen unter Verwendung des Core Manufacturing Simulation Data (CMSD) Information Model". In: Okt. 2010.

[12] Sören Bergmann, Sören Stelzer, Sascha Wüstemann und Steffen Strassburger. "Model Generation in SLX Using CMSD and XML Stylesheet Transformations". In: *Proceedings of the 2012 Winter Simulation Conference (WSC)*. Dez. 2012, S. 1–11. DOI: 10.1109/wsc.2012.6464981.

[13] Sören Bergmann und Steffen Strassburger. "Challenges for the Automatic Generation of Simulation Models for Production Systems". In: *Proceedings of the 2010 Summer Computer Simulation Conference*. Scsc '10. Ottawa, Ontario, Canada: Society for Computer Simulation International, 11. Juli 2010, S. 545–549. DOI: 10.5555/1999416.1999486.

[14] Sören Bergmann und Steffen Strassburger. "On the Use of the Core Manufacturing Simulation Data (CMSD) Standard : Experiences and Recommendations". In: (2015), S. 12.

[15] Jan Bessai. "A Type-theoretic Framework for Software Component Synthesis". Diss. Technische Universität Dortmund, 2019.

[16] Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens und Jakob Rehof. "Combinatory Logic Synthesizer". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Hrsg. von Tiziana Margaria und Bernhard Steffen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, S. 26–40. ISBN: 978-3-662-45234-9. DOI: 10.1007/978-3-662-45234-9_3.

[17] Jan Bessai und Anna Vasileva. *User Support for the Combinator Logic Synthesizer Framework*. Bd. 284. 27. Nov. 2018. DOI: 10.4204/eptcs.284.2.

[18] Alan W. Biermann. "The Inference of Regular LISP Programs from Examples". In: *IEEE Transactions on Systems, Man, and Cybernetics* 8.8 (Aug. 1978), S. 585–600. ISSN: 2168-2909. DOI: 10.1109/tsmc.1978.4310035.

[19] Katalin Bimbó. *Combinatory Logic: Pure, Applied and Typed*. 1. Edition. Chapman und Hall/CRC, 27. Juli 2011. 357 S.

[20] Rastislav Bodik und Barbara Jobstmann. "Algorithmic Program Synthesis: Introduction". In: *International Journal on Software Tools for Technology Transfer* 15.5 (Okt. 2013), S. 397–411. ISSN: 1433-2779, 1433-2787. DOI: 10.1007/s10009-013-0287-9.

[21] M. C. Bonney. "Simulation and Analysis of Industrial Systems". In: *Journal of the Operational Research Society* 22.1 (1. März 1971), S. 92–93. ISSN: 1476-9360. DOI: 10.1057/jors.1971.26.

[22] Andrei Borshchev und Nikolay Churkov. "Anylogic Cloud: Cloud-Based Simulation Analytics". In: *Proceedings of the 2018 Winter Simulation Conference*. Wsc '18. Gothenburg, Sweden: IEEE Press, 2018, S. 4245. ISBN: 978153866570.

[23] Christos G. Cassandras und Stéphane Lafortune. *Introduction to Discrete Event Systems*. 2. Aufl. Springer US, 2008. ISBN: 978-0-387-33332-8. DOI: 10.1007/978-0-387-68612-7.

[24] Uwe Clausen, Matthias Brueggenolte, Marc Kirberg, Christoph Besenfelder, Moritz Poeting und Mustafa Gueller. "Agent-Based Simulation in Logistics and Supply Chain Research: Literature Review and Analysis". In: *Advances in Production, Logistics and Traffic*. Hrsg. von Uwe Clausen, Sven Langkau und Felix Kreuz. Lecture Notes in Logistics. Cham: Springer International Publishing, 2019, S. 45–59. ISBN: 978-3-030-13535-5. DOI: 10.1007/978-3-030-13535-5_4.

[25] "Logistikkennzahlen für das Performance Measurement". In: *Handbuch Produktions- und Logistikmanagement in Wertschöpfungsnetzwerken*. Hrsg. von Hans Corsten, Ralf Gössinger und Thomas S. Spengler. De Gruyter Oldenbourg, 25. Juni 2018, S. 904–922. ISBN: 978-3-11-047380-3 978-3-11-047130-4. DOI: 10.1515/9783110473803-047.

[26] Krzysztof Czarnecki und Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Jan. 2000. ISBN: 0-201-30977-7.

[27] Georgios Dagkakis, Ioannis Papagiannopoulos und Cathal Heavey. "ManPy: an open-source software tool for building discrete event simulation models of manufacturing systems". In: *Software: Practice and Experience* 46.7 (2016), S. 955–981. ISSN: 1097-024x. DOI: 10.1002/spe.2347.

[28] Leonardo De Moura und Nikolaj Bjørner. "Satisfiability Modulo Theories: Introduction and Applications". In: *Communications of the ACM* 54.9 (1. Sep. 2011), S. 69. ISSN: 00010782. DOI: 10.1145/1995376.1995394.

[29] Tim Delbrügger, Frederik Döbbeler, Julian Graefenstein, Hendrik Lager, Lisa T. Lenz, Matthias Meißner, Daniel Müller, Philipp Regelmann, David Scholz, Christin Schumacher, Jan Winkels, Andreas Wirtz und Felix Zeidler. "Anpassungsintelligenz von Fabriken im dynamischen und komplexen Umfeld". In: *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb* 112.6 (28. Juni 2017), S. 364–368. ISSN: 0947-0085, 2511-0896. DOI: 10.3139/104.111731.

[30] Shahab Derhami, Jeffrey S. Smith und Kevin R. Gue. "A Simulation-based Optimization Approach to Design Optimal Layouts for Block Stacking Warehouses". en. In: *International Journal of Production Economics* 223 (Mai 2020), S. 107525. ISSN: 0925-5273. DOI: 10.1016/j.ijpe.2019.107525.

[31] Luís M. S. Dias, António A. C. Vieira, Guilherme A. B. Pereira und José A. Oliveira. "Discrete Simulation Software Ranking — A Top List of the Worldwide Most Popular and Used Tools". In: *2016 Winter Simulation Conference (WSC)*. 2016 Winter Simulation Conference (WSC). Dez. 2016, S. 1060–1071. DOI: 10.1109/wsc.2016.7822165.

[32] Uwe Dombrowski und Yannick Dix. "Simulationsbasierte Konfiguration der Produktionsplanung und -steuerung: Adaptive PPS auf Basis verteilter Intelligenz". In: *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb* 112.7 (18. Aug. 2017), S. 491–494. ISSN: 0947-0085, 2511-0896. DOI: 10.3139/104.111752.

[33] Boris Düdder, Jakob Rehof, George Heineman und Armend Hoxha. “Towards Migrating Object-Oriented Frameworks to Enable Synthesis of Product Line Members”. In: *Proceedings of the 19th International Conference on Software Product Line*. 2015, S. 56–60. DOI: 10.1145/2791060.2791076.

[34] Andrej Dudenhefner. *The Combinatory Logic Synthesizer (CL)S Framework in F#*. 14. Okt. 2021.

[35] Frank Eckardt. *Ein Beitrag zu Theorie und Praxis datengetriebener Modellgeneratoren zur Simulation von Produktionssystemen*. Berichte aus der Wirtschaftsinformatik. Aachen: Shaker, 2002. 214 S. ISBN: 978-3-8322-0383-2.

[36] Michael Eley. *Simulation in der Logistik*. Springer-Lehrbuch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-27372-8 978-3-642-27373-5. DOI: 10.1007/978-3-642-27373-5.

[37] Fadil Kallat, Jens Hetzler, Alexander Mages, Carina Mieth, Jakob Rehof, Christian Riest und Tristan Schäfer. “Automatic Component-based Synthesis of User-configured Manufacturing Simulation Models”. In: *Proceedings of the 2022 Winter Simulation Conference*. Winter Simulation Conference. Singapur, Singapur, 2022.

[38] Jonathan Fournier. “Model Building with Core Manufacturing Simulation Data”. In: *Proceedings of the 2011 Winter Simulation Conference (WSC)*. 2011 Winter Simulation Conference - (WSC 2011). Phoenix, AZ, USA: Ieee, Dez. 2011, S. 2214–2222. ISBN: 978-1-4577-2109-0 978-1-4577-2108-3 978-1-4577-2106-9 978-1-4577-2107-6. DOI: 10.1109/wsc.2011.6147933.

[39] John W. Fowler und Oliver Rose. “Grand Challenges in Modeling and Simulation of Complex Manufacturing Systems”. In: *Simulation* 80.9 (Sep. 2004), S. 469–476. ISSN: 0037-5497, 1741-3133. DOI: 10.1177/0037549704044324.

[40] Julian Graefenstein, David Scholz, Michael Henke, Jan Winkels und Jakob Rehof. “Intelligente Orchestrierung von Planungsprozessen: Anwendung von logikbasiertem Constraintsolving in der Fabrikplanung”. In: *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb* 112.4 (28. Apr. 2017), S. 209–214. ISSN: 0947-0085, 2511-0896. DOI: 10.3139/104.111696.

[41] Cordell Green. “Application of Theorem Proving to Problem Solving”. In: *Proceedings of the 1st international joint conference on Artificial intelligence*. Ijcai’69. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Mai 1969, S. 219–239.

[42] I. Grigoryev. *AnyLogic 7 in Three Days: A Quick Course in Simulation Modeling*. Ilya Grigoryev, 2015. ISBN: 978-1-5076-9136-6.

[43] Sumit Gulwani. “Dimensions in Program Synthesis”. In: *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. Ppdp ’10. Hagenberg, Austria: Association for Computing Machinery, 26. Juli 2010, S. 13–24. ISBN: 978-1-4503-0132-9. DOI: 10.1145/1836089.1836091.

[44] Sumit Gulwani, Oleksandr Polozov und Rishabh Singh. *Program Synthesis*. Bd. 4. 1-2. 2017, S. 1–119. DOI: 10.1561/2500000010.

[45] Uwe Gumpert und Michael Ritzschke. “Entwicklung eines simulationsgestützten Arbeitsplatzes für den Produktionsplaner”. In: *Simulationstechnik: 10. Symposium in Dresden, September 1996 Tagungsband*. Hrsg. von Wilfried Krug. Fortschritte in der Simulationstechnik. Wiesbaden: Vieweg+Teubner Verlag, 1996, S. 65–70. ISBN: 978-3-322-86541-0. DOI: 10.1007/978-3-322-86541-0_12.

[46] Kai Gutenschwager, Markus Rabe, Sven Spieckermann und Sigrid Wenzel. *Simulation in Produktion und Logistik: Grundlagen und Anwendungen*. Berlin: Springer Vieweg, 2017. 281 S. ISBN: 978-3-662-55744-0 978-3-662-55745-7. DOI: 10.1007/978-3-662-55745-7.

[47] Marijn Haverbeke. *Eloquent JavaScript, 3rd Edition: A Modern Introduction to Programming*. 3rd Edition. San Francisco: No Starch Press, 2018. 472 S. ISBN: 978-1-59327-950-9.

[48] George T. Heineman, Jan Bessai, Boris Düdder und Jakob Rehof. “A Long and Winding Road Towards Modular Synthesis”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*. Hrsg. von Tiziana Margaria und Bernhard Steffen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, S. 303–317. ISBN: 978-3-319-47166-2. DOI: 10.1007/978-3-319-47166-2_21.

[49] R. Hernandez Morales. *Systematik der Wandlungsfähigkeit in der Fabrikplanung, Fortschritt-Berichte VDI, Reihe 16: Technik und Wirtschaft*. Bd. 149. Düsseldorf: VDI-Verlag; 2003. 1-190. ISBN: 3-18-314916-8.

[50] J Roger Hindley und Jonathan P Seldin. *Lambda-Calculus and Combinators, an Introduction*. 2008.

[51] Edward Huang, Randeep Ramamurthy und Leon F. McGinnis. “System and Simulation Modeling Using SYSML”. In: *2007 Winter Simulation Conference*. 2007 Winter Simulation Conference. Dez. 2007, S. 796–803. DOI: 10.1109/wsc.2007.4419675.

[52] Yilin Huang, Mamadou D. Seck und Alexander Verbraeck. “From Data to Simulation Models: Component-based Model Generation with a Data-driven Approach”. In: *Proceedings of the 2011 Winter Simulation Conference (WSC)*. 2011 Winter Simulation Conference - (WSC 2011). Phoenix, AZ, USA: Ieee, Dez. 2011, S. 3719–3729. ISBN: 978-1-4577-2109-0 978-1-4577-2108-3 978-1-4577-2106-9 978-1-4577-2107-6. DOI: 10.1109/wsc.2011.6148065.

[53] Lars Huber und Sigrid Wenzel. “Trends und Handlungsbedarfe der Ablaufsimulation in der Automobilindustrie”. In: *Industrie Management* 5 (1. Sep. 2011), S. 27–30. ISSN: 1434-1980.

[54] Gesellschaft für Informatik (GI). *Digitaler Zwilling*. URL: https://gi.de/informatiklexikon/digitaler-zwilling/ (besucht am 29. 01. 2021).

[55] Mohsen Jahangirian, Tillal Eldabi, Aisha Naseer, Lampros K. Stergioulas und Terry Young. “Simulation in Manufacturing and Business: A Review”. In: *European Journal of Operational Research* 203.1 (Mai 2010), S. 1–13. ISSN: 03772217. DOI: 10.1016/j.ejor.2009.06.004.

[56] Brenda Jin, Saurabh Sahni und Amir Shevat. *Designing Web APIs: Building APIs That Developers Love*. Sebastopol, CA: O’Reilly Media, Inc, USA, 20. Sep. 2018. 217 S. ISBN: 978-1-4920-2692-1.

[57] Marcus Johansson, Björn Johansson, Swee Leong, Frank Riddick und Y Tina Lee. “A Real World Pilot Implementation of the Core Manufacturing Simulation Data Model”. In: (2008), S. 9.

[58] Fadil Kallat, Carina Mieth, Jakob Rehof und Anne Meyer. “Using Component-based Software Synthesis and Constraint Solving to generate Sets of Manufacturing Simulation Models”. In: *Procedia CIRP*. 53rd CIRP Conference on Manufacturing Systems 2020 93 (1. Jan. 2020), S. 556–561. ISSN: 2212-8271. DOI: 10.1016/j.procir.2020.03.018.

[59] Fadil Kallat, Jakob Pfrommer, Jan Bessai, Jakob Rehof und Anne Meyer. “Automatic Building of a Repository for Component-based Synthesis of Warehouse Simulation Models”. In: *Procedia CIRP*. 54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0 104 (1. Jan. 2021), S. 1440–1445. ISSN: 2212-8271. DOI: 10.1016/j.procir.2021.11.243.

[60] Fadil Kallat, Tristan Schäfer und Anna Vasileva. “CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories”. In: *Electronic Proceedings in Theoretical Computer Science*. Bd. 301. 23. Aug. 2019, S. 51–65. DOI: 10.4204/eptcs.301.7. arXiv: 1908.09481.

[61] Parastu Kasaie und W. David Kelton. “Guidelines for Design and Analysis in Agent-based Simulation Studies”. In: *2015 Winter Simulation Conference (WSC)*. 2015 Winter Simulation Conference (WSC). Huntington Beach, CA, USA: Ieee, Dez. 2015, S. 183–193. ISBN: 978-1-4673-9743-8. DOI: 10.1109/wsc.2015.7408163.

[62] Stefan Kassen, Holger Tammen, Maximilian Zarte und Agnes Pechmann. “Concept and Case Study for a Generic Simulation as a Digital Shadow to Be Used for Production Optimisation”. In: *Processes* 9.8 (2021). ISSN: 2227-9717. DOI: 10.3390/pr9081362.

[63] Abdelhak Khemiri, Claude Yugma und Stéphane Dauzère-Pérès. “Towards A Generic Semiconductor Manufacturing Simulation Model”. In: *2021 Winter Simulation Conference (WSC)*. Dez. 2021, S. 1–10. DOI: 10.1109/wsc52266.2021.9715349.

[64] John R. Koza. “Genetic Programming as a Means for Programming Computers by Natural Selection”. en. In: *Statistics and Computing* 4.2 (Juni 1994), S. 87–112. ISSN: 1573-1375. DOI: 10.1007/bf00175355.

[65] Axel Kuhn, Katja Klingebiel, Achim Schmidt und Nils Luft. “Modellgestütztes Planen und kollaboratives Experimentieren für robuste Distributionssysteme”. In: *HAB-Forschungsseminar der Hochschulgruppe für Arbeits-und Betriebsorganisation* (2011), S. 177–198.

[66] Andreas Lattner, Tjorben Bogon, Yann Lorion und Ingo Timm. “A knowledge-based approach to automated simulation model adaptation”. In: Spring Simulation Multiconference 2010, SpringSim’10. 1. Jan. 2010, S. 153. DOI: 10.1145/1878537.1878697.

[67] Averill M. Law. *Simulation Modeling and Analysis*. Fifth edition. McGraw-Hill series in industrial engineering and management science. Dubuque: McGraw-Hill Education, 2013. ISBN: 978-0-07-340132-4.

[68] Tobias Lechler, Eva Fischer, Maximilian Metzner, Andreas Mayr und Jörg Franke. “Virtual Commissioning – Scientific Review and Exploratory Use Cases in Advanced Production Systems”. en. In: *Procedia CIRP*. 52nd CIRP Conference on Manufacturing Systems (CMS), Ljubljana, Slovenia, June 12-14, 2019 81 (Jan. 2019), S. 1125–1130. ISSN: 2212-8271. DOI: 10.1016/j.procir.2019.03.278.

[69] C.H. Lee und Richard N. Zobel. “Representation of Simulation Model Components for Model Generation and a Model Library”. In: *Proceedings of the 29th Annual Simulation Symposium*. Proceedings of the 29th Annual Simulation Symposium. Apr. 1996, S. 193–201. DOI: 10.1109/simsym.1996.492167.

[70] John D. C. Little. “A Proof for the Queuing Formula: $L = \lambda W$”. In: *Operations Research* 9.3 (1961), S. 383–387. ISSN: 0030-364x.

[71] VDI-Fachbereich Technische Logistik. *VDI-Richtlinie 4400 Blatt 1, Logistikkennzahlen für die Beschaffung*. Mai 2001.

[72] VDI-Fachbereich Technische Logistik. *VDI-Richtlinie 4400 Blatt 2, Logistikkennzahlen für die Produktion*. Dez. 2004.

[73] VDI-Fachbereich Technische Logistik. *VDI-Richtlinie 4400 Blatt 3, Logistikkennzahlen für die Distribution*. Juli 2002.

[74] Peter Lorenz und Thomas Schulze. “Layout Based Model Generation”. In: *Winter Simulation Conference Proceedings, 1995*. 1995 Winter Simulation Conference. Arlington, VA, USA: Ieee, 1995, S. 728–735. ISBN: 978-0-7803-3018-4. DOI: 10.1109/wsc.1995.478850.

[75] C M Macal und M J North. “Tutorial on Agent-based Modelling and Simulation”. In: *Journal of Simulation* 4.3 (1. Sep. 2010), S. 151–162. ISSN: 1747-7786. DOI: 10.1057/jos.2010.3.

[76] Charles M. Macal und Michael J. North. “Agent-based Modeling and Simulation: Desktop ABMS”. In: *2007 Winter Simulation Conference*. 2007 Winter Simulation Conference. Washington, DC, USA: Ieee, Dez. 2007, S. 95–106. ISBN: 978-1-4244-1305-8 978-1-4244-1306-5. DOI: 10.1109/wsc.2007.4419592.

[77] Dominik Mäckel, Jan Winkels und Christin Schumacher. “Synthesis of Scheduling Heuristics by Composition and Recombination”. In: *Optimization and Learning*. Hrsg. von Bernabé Dorronsoro, Lionel Amodeo, Mario Pavone und Patricia Ruiz. Communications in Computer and Information Science. Cham: Springer International Publishing, 2021, S. 283–293. ISBN: 978-3-030-85672-4. DOI: 10.1007/978-3-030-85672-4_21.

[78] Sajjad Mahmood, Richard Lai und Y.S. Kim. “Survey of Component-based Software Development”. In: *Software, IET* 1 (1. Mai 2007), S. 57–66. DOI: 10.1049/iet-sen:20060045.

[79] Zohar Manna und Richard J. Waldinger. “Toward Automatic Program Synthesis”. In: *Communications of the ACM* 14.3 (März 1971), S. 151–165. ISSN: 0001-0782. DOI: 10.1145/362566.362568.

[80] Lothar März, Wilfried Krug, Oliver Rose und Gerald Weigert, Hrsg. *Simulation und Optimierung in Produktion und Logistik*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-14535-3 978-3-642-14536-0. DOI: 10.1007/978-3-642-14536-0.

[81] S. C. Mathewson. “The Application of Program Generator Software and Its Extensions to Discrete Event Simulation Modeling”. In: *IIE Transactions* 16.1 (1. März 1984), S. 3–18. ISSN: 0740-817x. DOI: 10.1080/07408178408974663.

[82] Leonardo de Moura und Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von C. R. Ramakrishnan und Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, S. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[83] Dimitris Mourtzis. “Simulation in the Design and Operation of Manufacturing Systems: State of the Art and New Trends”. In: *International Journal of Production Research* 58.7 (2. Apr. 2020), S. 1927–1949. ISSN: 0020-7543. DOI: 10.1080/00207543.2019.1636321.

[84] Dimitris Mourtzis, Michael Doukas und Dimitra Bernidaki. “Simulation in Manufacturing: Review and Challenges”. In: *Procedia CIRP* 25 (2014), S. 213–229. ISSN: 22128271. DOI: 10.1016/j.procir.2014.10.032.

[85] Dimitris Mourtzis, Nikolaos Papakostas, Dimitris Mavrikios, Sotiris Makris und Kosmas Alexopoulos. “The Role of Simulation in Digital Manufacturing: Applications and Outlook”. In: *International Journal of Computer Integrated Manufacturing* 28.1 (2. Jan. 2015), S. 3–24. ISSN: 0951-192x, 1362-3052. DOI: 10.1080/0951192x.2013.800234.

[86] Ashkan Negahban und Jeffrey S. Smith. “Simulation for Manufacturing System Design and Operation: Literature Review and Analysis”. In: *Journal of Manufacturing Systems* 33.2 (Apr. 2014), S. 241–261. ISSN: 02786125. DOI: 10.1016/j.jmsy.2013.12.007.

[87] Adrian Neyrinck, Armin Lechler und Alexander Verl. “Automatic Variant Configuration and Generation of Simulation Models for Comparison of Plant and Machinery Variants”. In: *Procedia CIRP*. The 22nd CIRP Conference on Life Cycle Engineering 29 (1. Jan. 2015), S. 62–67. ISSN: 2212-8271. DOI: 10.1016/j.procir.2015.02.069.

[88] Jan Christoph Nöcker. *Zustandsbasierte Fabrikplanung*. Unter Mitarb. von Fraunhofer-Institut für Produktionstechnologie IPT. 1. Aufl. Ergebnisse aus der Produktionstechnik Produktionssystematik 2012,6. Aachen: Apprimus-Verl, 2012. 287 S. ISBN: 978-3-86359-059-8.

[89] W.B. Nordgren. “Steps for Proper Simulation Project Management”. In: *Winter Simulation Conference Proceedings, 1995*. 1995 Winter Simulation Conference. Arlington, VA, USA: Ieee, 1995, S. 68–73. ISBN: 978-0-7803-3018-4. DOI: 10.1109/wsc.1995.478707.

[90] Peter Nyhuis, Hrsg. *Wandlungsfähige Produktionssysteme: heute die Industrie von morgen gestalten*. Garbsen: PZH Produktionstechnisches Zentrum, 2008. 166 S. ISBN: 978-3-939026-96-9. DOI: 10.2314/gbv:633626406.

[91] Object Management Group (OMG). *Systems Modeling Language Version 1.6 Specification*. 2019. URL: https://www.omg.org/spec/SysML/1.6/.

[92] Jakob Pfrommer und Anne Meyer. “Autonomously Organized Block Stacking Warehouses: A Review of Decision Problems and Major Challenges”. In: *Logistics Journal Proceedings* (Okt. 2020). DOI: 10.2195/lj_Proc_pfrommer_en_202012_01.

[93] Michael Pidd und Stewart Robinson. “Organising Insights into Simulation Practice”. In: *2007 Winter Simulation Conference*. 2007 Winter Simulation Conference. Dez. 2007, S. 771–775. DOI: 10.1109/wsc.2007.4419672.

[94] Thomas Plümer und Egbert Steinfatt. *Produktions- und Logistikmanagement*. 1 Edition. De Gruyter Studium. Boston/Berlin: De Gruyter, 2017. ISBN: 978-3-11-041389-2.

[95] *Refactoring: Improving the Design of Existing Code*. Usa: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201485672.

[96] Jakob Rehof. “Minimal Typings in Atomic Subtyping”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Popl ’97. New York, NY, USA: Association for Computing Machinery, 1. Jan. 1997, S. 278–291. ISBN: 978-0-89791-853-4. DOI: 10.1145/263699.263738.

[97] Jakob Rehof. “Towards Combinatory Logic Synthesis”. In: *1st International Workshop on Behavioural Types, BEAT*. 2013, S. 12.

[98] Heiner Reinhardt, Marek Weber und Matthias Putz. “A Survey on Automatic Model Generation for Material Flow Simulation in Discrete Manufacturing”. In: *Procedia CIRP* 81 (2019), S. 121–126. ISSN: 22128271. DOI: 10.1016/j.procir.2019.03.022.

[99] Frank H. Riddick und Yung-Tsun T. Lee. “Core Manufacturing Simulation Data (CMSD): A Standard Representation for Manufacturing Simulation-related Information”. In: (31. Aug. 2010).

[100] Blaž Rodič. “Industry 4.0 and the New Simulation Modelling Paradigm”. In: *Organizacija* 50.3 (1. Aug. 2017), S. 193–207. ISSN: 1581-1832. DOI: 10.1515/orga-2017-0017.

[101] Mathias Rohl und Stefan Morgenstern. “Composing Simulation Models Using Interface Definitions Based on Web Service Descriptions”. In: *2007 Winter Simulation Conference*. 2007 Winter Simulation Conference. Dez. 2007, S. 815–822. DOI: 10.1109/wsc.2007.4419677.

[102] Mathias Röhl. “Definition und Realisierung einer Plattform zur modellbasierten Komposition von Simulationsmodellen”. Dissertation. Universität Rostock, 2008. DOI: 10.18453/rosdok_id00000298.

[103] Tristan Schäfer, Jim A Bergmann, Rafael Garcia Carballo, Jakob Rehof und Petra Wiederkehr. “A Synthesis-based Tool Path Planning Approach for Machining Operations”. In: *Procedia CIRP* 104 (2021), S. 918–923.

[104] Tristan Schäfer, Jan Bessai, Constantin Chaumet, Jakob Rehof und Christian Riest. “Design Space Exploration for Sampling-Based Motion Planning Programs with Combinatory Logic Synthesis”. In: *Proceedings of the Fifteenth Workshop on the Algorithmic Foundations of Robotics*. Washington, DC, USA: Springer Cham, 2022. ISBN: 978-3-031-21089-1.

[105] Daniel Scholtyssek. “Synthese von Variationen eines Systems zur Berechnung profitabler Touren für Handelsvertreter unter Zuhilfenahme eines Inhabitationsalgorithmus”. Magisterarb. Technische Universität Dortmund, Aug. 2021.

[106] M. Schönfinkel. “Über die Bausteine der mathematischen Logik”. In: *Mathematische Annalen* 92.3 (1. Sep. 1924), S. 305–316. ISSN: 1432-1807. DOI: 10.1007/bf01448013.

[107] Severyn-Luka Anufriyev. “Entwicklung eines Syntheseziel-Editors für die Komponentenbasierte Synthese von Simulationsmodellen”. Bachelorarbeit. 2021.

[108] D. E. Shaw, W. Swartout und C. C. Green. “Inferring LISP Programs From Examples”. In: *Ijcai*. 1975. DOI: 10.7916/d89k4k6x.

[109] David Canfield Smith. “PYGMALION: A Creative Programming Environment”. en. Diss. Stanford Univ Ca Dept Of Computer Science, Juni 1975.

[110] Jeffrey S. Smith. “Survey on the Use of Simulation for Manufacturing System Design and Operation”. In: *Journal of Manufacturing Systems* 22.2 (1. Jan. 2003), S. 157–171. ISSN: 0278-6125. DOI: 10.1016/s0278-6125(03)90013-6.

[111] Armando Solar-Lezama. “Program Sketching”. In: *International Journal on Software Tools for Technology Transfer* 15.5 (1. Okt. 2013), S. 475–495. ISSN: 1433-2787. DOI: 10.1007/s10009-012-0249-7.

[112] Young Jun Son, Albert T Jones und Richard A Wysk. “Component-based Simulation Modeling from Neutral Component Libraries”. In: *Computers & Industrial Engineering* 45.1 (1. Juni 2003), S. 141–165. ISSN: 0360-8352. DOI: 10.1016/s0360-8352(03)00023-8.

[113] Young Jun Son und Richard A Wysk. “Automatic Simulation Model Generation for Simulation-based, Real-time Shop Floor Control”. In: *Computers in Industry* 45.3 (Juli 2001), S. 291–308. ISSN: 01663615. DOI: 10.1016/s0166-3615(01)00086-0.

[114] Wolfgang H Staehle. *Kennzahlen und Kennzahlensysteme als Mittel der Organisation und Führung von Unternehmen*. Springer-Verlag, 1969.

[115] Steffen Straßburger, Sören Bergmann und Hannes Müller-Sommer. “Modellgenerierung im Kontext der Digitalen Fabrik - Stand der Technik und Herausforderungen”. In: (2010), S. 8. DOI: 10.5445/ksp/1000019635.

[116] Phillip D. Summers. “A Methodology for LISP Program Construction from Examples”. In: *J. Acm* 24.1 (Jan. 1977), S. 161–175. ISSN: 0004-5411. DOI: 10.1145/321992.322002.

[117] Peter Tabeling. *Softwaresysteme und ihre Modellierung: Grundlagen, Methoden und Techniken*. eXamen.press. Berlin Heidelberg: Springer-Verlag, 2006. ISBN: 978-3-540-25828-5. DOI: 10.1007/3-540-29276-4.

[118] The SMT-LIB Initiative. *SMT-LIB The Satisfiability Modulo Theories Library - SMT Solvers*. SMT-LIB The Satisfiability Modulo Theories Library. URL: https://smtlib.cs.uiowa.edu/solvers.shtml (besucht am 10. 01. 2022).

[119] W.J. Trybula. "Building Simulation Models Without Data". In: *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*. Bd. 1. 1994, 209–214 vol.1. DOI: 10.1109/icsmc.1994.399838.

[120] Alexander Verbraeck und Edwin C. Valentin. "Design Guidelines for Simulation Building Blocks". In: *2008 Winter Simulation Conference*. 2008 Winter Simulation Conference (WSC). Miami, FL, USA: Ieee, Dez. 2008, S. 923–932. ISBN: 978-1-4244-2707-9. DOI: 10.1109/wsc.2008.4736158.

[121] António Vieira, Luis Dias, Guilherme Pereira, José Oliveira, Maria Carvalho und P. Martins. "Using simio to automatically create 3d warehouses and compare different storage strategies". In: 43 (1. Jan. 2015), S. 335–343. DOI: 10.5937/fmet1504335V.

[122] António Vieira, Luis Dias, Maribel Santos, Guilherme Pereira und J. Oliveira. "Setting an Industry 4.0 Research and Development Agenda for Simulation – a Literature Review". In: *International Journal of Simulation Modelling* 17.3 (15. Sep. 2018), S. 377–390. ISSN: 17264529. DOI: 10.2507/ijsimm17(3)429.

[123] Richard J. Waldinger und Richard C. T. Lee. "PROW: A Step Toward Automatic Program Writing". In: *Proceedings of the 1st international joint conference on Artificial intelligence*. Ijcai'69. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Mai 1969, S. 241–252.

[124] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz und Giles Reger. "The SMT Competition 2015–2018". In: *Journal on Satisfiability, Boolean Modeling and Computation* 11.1 (1. Jan. 2019), S. 221–259. DOI: 10.3233/sat190123.

[125] Gerald Weigert, Oliver Rose, Pavel Gocev und Gottfried Mayer. "Kennzahlen zur Bewertung logistischer Systeme". In: KIT Scientific Publishing, Karlsruhe, 2010, S. 599–606.

[126] Sigrid Wenzel, Jana Stolipin, Ulrich Jessen und Universität Kassel. "Ablaufsimulation in Industrie 4.0". In: (2018), S. 4.

[127] Sigrid Wenzel, Jana Stolipin, Jakob Rehof und Jan Winkels. "Trends in Automatic Composition of Structures for Simulation Models in Production and Logistics". In: *2019 Winter Simulation Conference (WSC)*. 2019, S. 2190–2200. DOI: 10.1109/wsc40007.2019.9004959.

[128] Sigrid Wenzel, Matthias Weiß, Simone Collisi-Böhmer, Holger Pitsch und Oliver Rose. *Qualitätskriterien für die Simulation in Produktion und Logistik: Planung und Durchführung von Simulationsstudien*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-35272-3 978-3-540-35276-1. DOI: 10.1007/978-3-540-35276-1.

[129] Walter Wincheringer, Marko Sekulic und Marec Kexel. "Generisches Simulationsmodell für automatische Hochregallagersysteme". In: *Proceedings ASIM SST 2020*. 25. ASIM Symposium Simulationstechnik. ARGESIM Publisher Vienna, 2020, S. 389–395. ISBN: 978-3-901608-93-3. DOI: 10.11128/arep.59.a59054.

[130] Walter Wincheringer, Tobias Sohny und Marec Kexel. "Automatische Erstellung von digitalen Simulationszwillingen von Produktionssystemen". In: *Proceedings ASIM SST 2020*. 25. ASIM Symposium Simulationstechnik. ARGESIM Publisher Vienna, 2020, S. 327–333. ISBN: 978-3-901608-93-3. DOI: 10.11128/arep.59.a59046.

[131] Jan Winkels, Julian Graefenstein, Lisa Lenz, Kai Christian Weist, Kevin Krebil und Mike Gralla. "A Hybrid Approach of Modular Planning – Synchronizing Factory and Building Planning by Using Component based Synthesis". In: *Proceedings of the HICSS 2020 : Hawaii International Conference on System Sciences*. Hawaii International Conference on System Sciences. Hawaii, USA, 1. Jan. 2020. DOI: 10.24251/hicss.2020.806.

[132] Jan Winkels, Julian Graefenstein, Tristan Schäfer, David Scholz, Jakob Rehof und Michael Henke. "Automatic Composition of Rough Solution Possibilities in the Target Planning of Factory Planning Projects by Means of Combinatory Logic". In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*. Hrsg. von Tiziana Margaria und Bernhard Steffen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, S. 487–503. ISBN: 978-3-030-03427-6. DOI: 10.1007/978-3-030-03427-6_36.