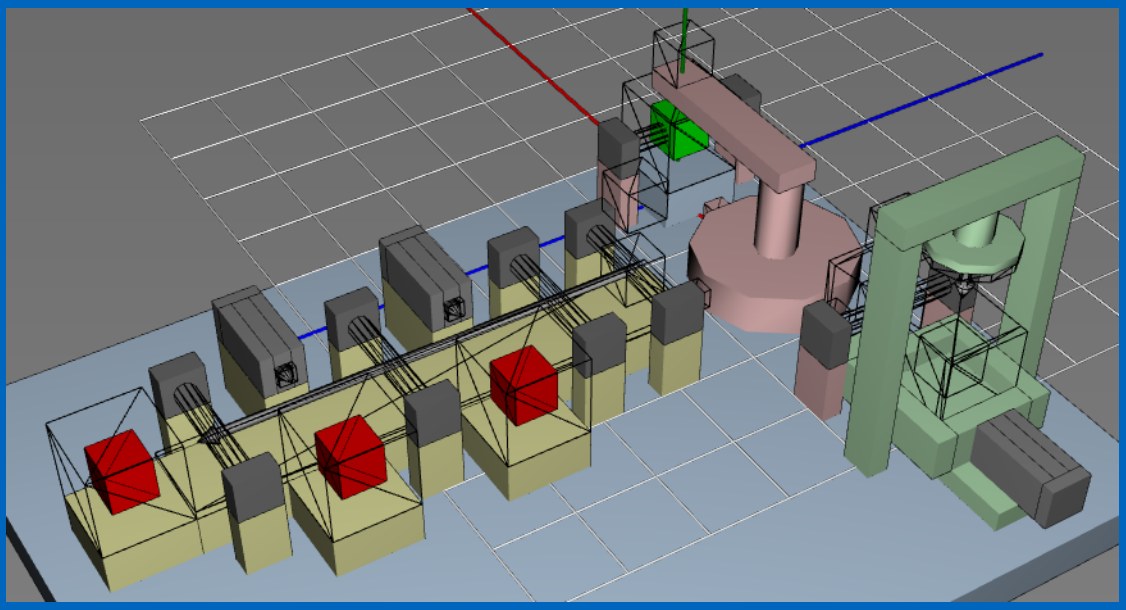
Test-driven conceptual design of cyber-physical manufacturing systems

Georg Hackenberg



Technische Universität München

Institut für Informatik
der Technischen Universität München

Test-driven conceptual design of cyber-physical manufacturing systems

Georg Hackenberg

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr.-Ing. Jörg Ott
Prüfer der Dissertation:	
1.	Prof. Dr. Dr. h.c. Manfred Broy
2.	Prof. Dr.-Ing. Michael Zäh

Die Dissertation wurde am 26.04.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 14.08.2018 angenommen.

Abstract

Today, manufacturing system engineering companies face three major trends: First, the systems have to implement more functions and, hence, grow in complexity. Then, besides mechanical engineering discipline the electrical and software engineering disciplines contribute more and more to the functionality of the system and, hence, require increasing efforts. And finally, the integration of the components from the individual engineering disciplines has become a costly task taking up to 25% of the project budget.

When talking to management personnel from manufacturing system engineering companies one can observe several practical challenges, which need to be addressed: For one, todays projects are dominated typically by mechanical design decisions and, thus, synergies between the different engineering disciplines cannot be exploited. Then, the design decisions taken in one discipline are not synchronized properly with the other disciplines leading to inconsistent and incompatible designs. Furthermore, the quality of the overall system design is not evaluated sufficiently and, hence, design flaws remain undetected until commissioning. And finally, the different engineering activities are carried out sequentially in a waterfall fashion potentially leading to costly design iterations.

In contrast, when looking at related work on the design of manufacturing systems, one can observe a number of problems that remain unsolved until today: In the first place, the approaches typically cover only a subset of design information and, therefore, do not represent each engineering discipline appropriately and sufficiently. Secondly, even if the approaches cover a wide range of design information an integrated formal foundation is missing, which defines the syntactic and semantic relations between the design elements precisely and unambiguously. As a consequence, an automated evaluation of the design information cannot be performed or can be performed only over a limited subset of design information. And finally, a practical methodology is missing fostering early verification and validation of the design decisions.

To overcome the current situation, this doctoral thesis provides six contributions: Foremost, the principles and ideas of test-driven and top-down, compositional software development methods are adapted to the cyber-physical manufacturing system domain. Then, an existing modeling technique and underlying formalism is adapted and extended to cover design information about customer requirements, manufacturing processes, and test cases in addition to part geometries, motion profiles, and energy as well as signal flow

behaviors. Subsequently, a taxonomy and formal definition of quality issues is developed including syntactic completeness and consistency as well as extrinsic and intrinsic semantic execution constraints. In the following, a prototypical tooling is proposed, which demonstrates how the modeling technique and the quality issues can be implemented in practice. Then, the overall approach is applied to an industry-close showcase, the pick and place unit installed at the Institute for Automation and Information Systems, Technical University of Munich. Finally, based on the experiment and data collected during tool usage the feasibility of the test-driven method for the conceptual design of cyber-physical manufacturing systems is discussed, the validity of the system model and the underlying modeling technique is analyzed, and the relevancy of the syntactic and semantic quality issues in practical applications is shown.

Kurzfassung

Hersteller fertigungstechnischer Maschinen und Anlagen werden heutzutage mit drei zentralen Trends konfrontiert: Zunächst müssen ihre Maschinen und Anlagen immer mehr Funktionalitäten bieten, weshalb die Komplexität der Maschinen und Anlagen stetig wächst. Dann tragen Software- und Elektrotechnikingenieure neben den klassischen Maschinenbauingenieuren immer mehr zur Funktionalität der Maschinen und Anlagen bei, weshalb sich die Arbeitsaufwände zwischen den Ingenieursdisziplinen verschieben. Und schließlich hat sich die Integration der Komponenten aus den unterschiedlichen Disziplinen zu einer kostspieligen Aufgabe entwickelt, welche bis zu 25% des Projektbudgets beanspruchen kann.

Wenn man mit Mitarbeitern aus dem Management der Hersteller fertigungstechnischer Maschinen und Anlagen spricht, man kann unterschiedliche praktische Herausforderungen erkennen, die adressiert werden sollten: Zunächst sind heutige Projekte typischerweise durch die Entwurfsentscheidungen von Maschinenbauingenieuren dominiert, wes-halb Synergien zwischen den verschiedenen Ingenieursdisziplinen nicht ausgenutzt werden können. Dann werden Entwurfsentscheidungen, die in einer Ingenieursdisziplin getroffen wurden, nicht ordentlich mit den anderen Ingenieursdisziplinen synchronisiert, was zu inkonsistenten und inkompatiblen Systementwürfen führt. Des Weiteren wird die Qualität des Gesamtsystementwurfs nicht ausreichend evaluiert, weshalb Entwurfsmängel bis zur Inbetriebnahme der Systeme unerkannt bleiben können. Und schließlich werden die verschiedenen Ingenieursaktivitäten sequenziell nach dem Wasserfallprinzip ausgeführt, was zu kostspieligen Überarbeitungsschleifen führen kann.

Demgegenüber kann man eine Reihe von ungelösten Problemen feststellen, wenn man sich verwandte Arbeiten zum Entwurf von fertigungstechnischen Maschinen und Anlagen ansieht: An erster Stelle decken bestehende Ansätze typischerweise nur eine Teilmenge von Entwurfsinformationen ab und repräsentieren deshalb die eine oder andere Ingenieursdisziplin nur unzureichend. Zweitens fehlt Ansätzen, die prinzipiell eine gute Abdeckung der Entwurfsinformationen bieten, eine integrierte formale Grundlage, welche die syntaktischen und semantischen Beziehungen zwischen den Entwurfs-elementen präzise und eindeutig definiert. Daraus folgt dass eine automatische Bewertung der Entwurfsinformationen entweder nicht erfolgen kann oder nur auf einer Teilmenge der Entwurfsinformationen durchgeführt werden kann. Und schließlich fehlt eine praktische

interdisziplinäre Entwurfsmethodik, welche eine frühe Verifikation und Validierung von Entwurfsentscheidungen fördert.

Um die aktuelle Situation zu verbessern bietet diese Doktorarbeit sechs Beiträge: In erster Linie werden die Prinzipien und Ideen der testgetriebenen und verfeinernden, komponentenbasierten Softwareentwicklungsmethoden auf die Domäne der cyberphysischen fertigungstechnischen Maschinen und Anlagen übertragen. Dann wird eine bestehende Modellierungstechnik und der darunter liegende mathematische Formalismus angepasst und erweitert, sodass Entwurfsinformationen bezüglich der Kundenanforderungen, der fertigungstechnischen Prozesse und der Testfälle zusätzlich zu Entwurfsinformationen bezüglich der Bauteilgeometrien, der Bewegungsprofile, sowie der Energie- und Signalflüsse erfasst werden können. Danach wird eine Taxonomie und formale Definition von Qualitätsproblemen entwickelt, welche syntaktische Vollständigkeit und Konsistenz sowie extrinsische und intrinsische semantische Ausführungsbedingungen umfassen. Im Folgenden wird ein prototypisches Werkzeug vorgestellt, welches demonstriert, wie die Modellierungstechnik und die Qualitätsprobleme in der Praxis umgesetzt werden können. Dann wird der vorgeschlagene Ansatz auf ein industrienahes Vorzeigeprojekt angewandt, nämlich eine Hub- und Schwenkeinheit, welche am Institut für Automatisierungs- und Informationssysteme der Technischen Universität München aufgestellt ist. Schließlich werden basierend auf dem Experiment und den Daten, die während der Werkzeugnutzung gesammelt wurden, die Durchführbarkeit der testgetriebenen Methode für cyberphysische fertigungstechnische Maschinen und Anlagen diskutiert, die Gültigkeit des Systemmodells und der darunter liegenden Modellierungstechnik analysiert, und die Relevanz der syntaktischen und semantischen Qualitätsprobleme in praktischen Anwendungen gezeigt.

Acknowledgements

I am grateful to my doctoral advisors, my former colleagues at the Chair for Software & Systems Engineering, Technische Universität München, my friends, my family, and my wife for their valuable feedback and support during the past years.

Publications

This doctoral thesis is based on a number of publications the author has contributed to. In the following, the publications are listed distinguishing between publications as *main author* and publications as *co-author* as well as advised *Bachelor's* and *Master's theses*.

Main author

- G. Hackenberg, M. Irlbeck, V. Koutsoumpas, and D. Bytschkow. Applying formal software engineering techniques to smart grids. [HIKB12]
- G. Hackenberg and D. Bytschkow. Towards Early Emergent Property Understanding: Merging Behavior Space Exploration and Model-based Software Engineering. [HB12]
- G. Hackenberg, C. Richter, and M. F. Zäh. Durchgängig modellbasierte Entwicklung von Werkzeugmaschinen. [HRZ13]
- G. Hackenberg, C. Richter, and M. F. Zäh. A Multi-disciplinary Modeling Technique for Requirements Management in Mechatronic Systems Engineering. [HRZ14]
- G. Hackenberg, M. Irlbeck, V. Koutsoumpas, and D. Bytschkow. A Rapid Prototyping Approach for Smart Energy Systems Based on Partial System Models. [HIKB14]
- G. Hackenberg, A. Campetelli, C. Legat, J. Mund, S. Teufl, and B. Vogel-Heuser. Formal Technical Process Specification and Verification for Automated Production Systems. [HCL+14]
- G. Hackenberg. EnergyFOCUS Modellbasierte Entwicklungsmethode für die Informations- und Kommunikationstechnologie (IKT) intelligenter Energiesysteme. [Hac15]
- G. Hackenberg, C. Richter, and M. F. Zäh. From Conception to Refinement in Mechatronic Systems Engineering. [HRZ15a]

– G. Hackenberg, C. Richter, and M. F. Zäh. Integrierte modellbasierte Entwicklung mechatronischer Systeme im Maschinen- und Anlagenbau (IMoMeSA). [HRZ15b]

– G. Hackenberg, M. Gleirscher, T. Stocker, C. Richter, and G. Reinhart. MaCon: Consistent Cross-Disciplinary Conception of Manufacturing Systems. [HGS+16]

– G. Hackenberg and J. Mund. Cyber-physical manufacturing systems development: A test-driven approach and exploratory case study. [HM16]

Co-author

– S. Eder, H. Femmer, and G. Hackenberg. ifedit ermglicht innovative Schnittstellenspezifikationen. [EFH13]

– M. F. Zäh, C. Richter, and G. Hackenberg. Herausforderungen im mechatronischen Entwicklungsprozess: Anforderungsanalyse bei ausgewählten Werkzeugmaschinenherstellern. [ZRH13a]

– M. F. Zäh, C. Richter, and G. Hackenberg. Integrierte modellbasierte Entwicklung mechatronischer Systeme im Maschinen- und Anlagenbau (IMoMeSA). [ZRH13b]

– C. Richter, G. Hackenberg, and M. F. Zäh. Modellbasierte Entwicklungsmethode für modulare Maschinen und Anlagen. [RHZ13]

– M. Irlbeck, D. Bytschkow, G. Hackenberg, and V. Koutsoumpas. Towards a bottom-up development of reference architectures for smart energy systems. [IBHK13]

– A. Campetelli, G. Hackenberg, M. Junker. Modelling of SPES XT Case Studies with FOCUS Components. [CHJ13]

– C. Richter, G. Hackenberg, and M. F. Zäh. Interdisziplinäre Funktionsmodellierung für die Generierung von robustem Steuerungscode für fehlertolerantes Systemverhalten. [RHZ14a]

– C. Richter, M. F. Zäh, and G. Hackenberg. Interdisziplinäre Modellierungstechnik zur Entwicklung mechatronischer Systeme. [RHZ14b]

– A. Vogelsang, S. Eder, G. Hackenberg, M. Junker, and S. Teufl. Supporting Concurrent Development of Requirements and Architecture – A Model-based Approach. [VEH+14]

– C. Legat, J. Mund, A. Campetelli, G. Hackenberg, J. Folmer, D. Schütz, M. Broy, and B. Vogel-Heuser. Interface Behavior Modeling for Automatic Verification of Industrial Automation Systems' Functional Conformance. [LMC+14]

– C. Richter, M. F. Zäh, and G. Hackenberg. Integrierte modellbasierte Entwicklung von mechatronischen Systemen - Abschluss des AiF-Projektes "IMoMeSA". [RZH14]

– D. Ascher and G. Hackenberg. Early Estimation of Multi-Objective Traffic Flow. [AH14]

– A. Campetelli, M. Gleischer, G. Hackenberg, M. Junker. Konzepte und Werkzeug-Prototypen für die mechatronische Modellierung von Embedded Systems. [CGHJ14]

– A. Campetelli and G. Hackenberg. Performance Analysis of Adaptive Runge-Kutta Methods in Region of Interest. [CH15]

– S. Teufl and G. Hackenberg. Efficient Impact Analysis of Changes in the Requirements of Manufacturing Automation Systems. [TH15]

– C. Richter, G. Hackenberg, P. Stich, and G. Reinhart. Modellbasierte Konzeption von Benutzerschnittstellen im Entwicklungsprozess von mechatronischen Systemen. [RHSR15]

– C. Richter, G. Hackenberg, and M. F. Zäh. Integrated Requirements and Systems Modeling in the Mechatronic Development Process. [RHZR15]

– D. Ascher and G. Hackenberg. Integrated transportation and power system modeling. [AH15]

– D. Ascher and G. Hackenberg. The TransP-0 framework for integrated transportation and power system design. [AH16]

– D. Ascher and G. Hackenberg. The passenger extension of the TRANSP-0 system design framework. [AH17]

Advisor

– D. Ascher. A Model-based Approach for Specification and Validation of Smart E-Mobility Control Requirements. [Asc13]

– J. Färber. Entwicklung eines Visualisierungsframeworks für das Verhalten intelligenter Energiesysteme auf Basis von XTREAM. [Fär14]

– S. Einwang. Entwurf, Implementierung und Demonstration eines 3D Animationsframework für das Explorationswerkzeug XTREAM. [Ein14]

– L. Kaiser. Development of a model for the intelligent operation of a hydro power plant chain. [Kai15]

– T. Stocker. Implementierung und Evaluation des Entwicklungsprozesses "IMoMeSA" für den Maschinen- und Anlagenbau. [Sto15]

– D. Ascher. Entwicklung eines verteilten Ansatzes zur näherungsweisen numerischen Lösung von zeitdiskreten optimalen Steuerungsproblemen. [Asc15]

– T. Mulenko. Advanced experiment data collection and analysis for the MaCon approach. [Mul16]

Contents

1 Introduction **1**
- 1.1 Considered systems . . . 1
  - 1.1.1 Manufacturing systems . . . 1
  - 1.1.2 Machine tools . . . 3
- 1.2 Recent trends . . . 4
  - 1.2.1 System complexities . . . 4
  - 1.2.2 Engineering efforts . . . 5
  - 1.2.3 Commissioning costs . . . 6
- 1.3 Practical challenges . . . 7
  - 1.3.1 Mechanical dominance . . . 8
  - 1.3.2 Lacking synchronization . . . 9
  - 1.3.3 Insufficient evaluation . . . 9
  - 1.3.4 Sequential engineering . . . 10
- 1.4 Claimed contributions . . . 10
  - 1.4.1 Test-driven method . . . 12
  - 1.4.2 Modeling technique . . . 12
  - 1.4.3 Quality issues . . . 13
  - 1.4.4 Prototypical tooling . . . 13
  - 1.4.5 Industry-close showcase . . . 14
  - 1.4.6 Critical discussion . . . 14
- 1.5 Intended audience . . . 14
  - 1.5.1 System builders . . . 16
  - 1.5.2 Tool providers . . . 16
  - 1.5.3 Scientific researchers . . . 17
- 1.6 Summary and outlook . . . 18

2 Differentiation **19**
- 2.1 Related work . . . 19
  - 2.1.1 Diagram-based techniques . . . 20
  - 2.1.2 Physics-based techniques . . . 22

2.1.3 Component-based techniques . . . 25
2.1.4 Matrix-based techniques . . . 27
2.1.5 Function-based techniques . . . 30
2.1.6 Ontology-based techniques . . . 33
2.1.7 Commercial tools . . . 37
2.2 Remaining problems . . . 41
2.2.1 Information coverage . . . 43
2.2.2 Integrated formalism . . . 43
2.2.3 Automated evaluation . . . 44
2.2.4 Practical methodology . . . 45
2.3 Research objectives . . . 46
2.4 Summary and outlook . . . 46

3 Test-driven method **47**
3.1 Preparation phase . . . 48
3.1.1 Requirement specification . . . 49
3.1.2 Process specification . . . 50
3.1.3 Test specification . . . 52
3.2 Implementation phase . . . 53
3.2.1 Test selection . . . 55
3.2.2 Architecture specification . . . 57
3.2.3 Behavior specification . . . 59
3.2.4 Part specification . . . 61
3.3 Summary and outlook . . . 63

4 Theoretical foundation **65**
4.1 Focus on components and streams (FOCUS) . . . 65
4.1.1 Streams . . . 65
4.1.2 Channels . . . 66
4.1.3 Components . . . 68
4.1.4 State transition systems . . . 70
4.2 Spatio-temporal engineering models (STEM) . . . 72
4.2.1 Transformable collision spaces . . . 72
4.2.2 Spatio-temporal components . . . 73
4.2.3 Extended spatio-temporal components . . . 78
4.3 Summary and outlook . . . 84

5 Modeling technique **85**
5.1 Basic concepts . . . 85
5.1.1 Observations . . . 86
5.1.2 Executables . . . 87
5.1.3 Expressions . . . 89
5.1.4 Volumes . . . 92
5.1.5 Transforms . . . 94
5.2 Revised concepts . . . 95
5.2.1 Components . . . 95
5.2.2 Ports . . . 110
5.2.3 Channels . . . 115
5.2.4 Parts . . . 116
5.2.5 Behaviors . . . 117
5.3 Added concepts . . . 118
5.3.1 Requirements . . . 119
5.3.2 Properties . . . 119
5.3.3 Monitors . . . 121
5.3.4 Scenarios . . . 122
5.4 Summary and outlook . . . 124

6 Quality issues **125**
6.1 Syntactic issues . . . 126
6.1.1 Incompleteness issues . . . 126
6.1.2 Inconsistency issues . . . 129
6.2 Semantic issues . . . 143
6.2.1 Extrinsic issues . . . 146
6.2.2 Intrinsic issues . . . 148
6.3 Summary and outlook . . . 155

7 Prototypical tooling **157**
7.1 Modeling interface . . . 157
7.1.1 Toolbar view . . . 158
7.1.2 Explorer view . . . 158
7.1.3 Editor view . . . 159
7.1.4 Scene view . . . 163
7.1.5 Issues view . . . 164
7.1.6 Changes view . . . 165
7.1.7 Attributes view . . . 166

7.2 Testing interface . . . 167
7.2.1 Toolbar view . . . 168
7.2.2 Explorer view . . . 169
7.2.3 Editor view . . . 170
7.2.4 Scene view . . . 173
7.2.5 Issues view . . . 174
7.2.6 Results view . . . 175
7.2.7 Attributes view . . . 176
7.3 Summary and outlook . . . 177

8 Industry-close showcase 179
8.1 Templates . . . 180
8.1.1 White/gray/black workpiece . . . 180
8.1.2 Component sensor . . . 181
8.1.3 Workpiece sensor . . . 181
8.1.4 Abstract static cylinder . . . 182
8.1.5 Concrete static cylinder . . . 183
8.1.6 Concrete static cylinder / Piston . . . 184
8.1.7 Concrete dynamic cylinder . . . 184
8.1.8 Concrete dynamic cylinder / Tip . . . 186
8.2 Components . . . 186
8.2.1 PPU . . . 187
8.2.2 PPU / Distributor . . . 192
8.2.3 PPU / Distributor / Twister . . . 197
8.2.4 PPU / Distributor / Lifter . . . 198
8.2.5 PPU / Distributor / Lifter / Arm . . . 199
8.2.6 PPU / Distributor / Controller . . . 200
8.2.7 PPU / Stamper . . . 201
8.2.8 PPU / Stamper / Basket . . . 205
8.2.9 PPU / Stamper / Stamp . . . 205
8.2.10 PPU / Stamper / Controller . . . 206
8.2.11 PPU / Sorter . . . 207
8.2.12 PPU / Sorter / Actuator . . . 211
8.2.13 PPU / Sorter / Controller . . . 212
8.3 Summary and outlook . . . 213

9 Critical discussion **215**
9.1 Method feasibility . . . 216
9.1.1 Activity sequences . . . 216
9.1.2 Design revisions . . . 217
9.1.3 System increments . . . 219
9.2 Model validity . . . 220
9.2.1 System architecture . . . 221
9.2.2 System behavior . . . 226
9.3 Issue relevancy . . . 235
9.3.1 Syntactic issues . . . 235
9.3.2 Semantic issues . . . 238
9.4 Study validity . . . 242
9.4.1 Internal validity . . . 242
9.4.2 External validity . . . 245
9.5 Summary and outlook . . . 245

10 Conclusion **247**
10.1 Summary . . . 247
10.1.1 Practical challenges . . . 247
10.1.2 Remaining problems . . . 248
10.1.3 Claimed contributions . . . 248
10.2 Outlook . . . 249
10.2.1 Method efficiency . . . 249
10.2.2 Model suitability . . . 250
10.2.3 Issue completeness . . . 251
10.2.4 Tool usability . . . 253

1 Introduction

Test-driven conceptual design of cyber-physical manufacturing systems represents a novel approach to manufacturing system engineering, which provides a number of advantages over existing techniques. To introduce the interested reader, in this chapter first the class of manufacturing systems is explained in Section 1.1, which the proposed approach addresses specifically. Then, three major trends in the manufacturing systems domain are described in Section 1.2, which have been documented in literature and shape today's and future manufacturing systems products and related manufacturing systems engineering approaches. Subsequently, four practical challenges manufacturing systems builders are facing today are illustrated in Section 1.3, which have been collected in a number of interviews with practitioners from industry and from which the goals of this doctoral thesis are derived. Thereafter, the contributions of this doctoral thesis are summarized in Section 1.4, which are meant to tackle the practical challenges and, thus, reach the underlying research goals and answer the respective research questions. Finally, the intended audience of this document is described in Section 1.5, which is targeted specifically and which is expected to benefit most from the research directions explored and results achieved throughout the course of this work.

1.1 Considered systems

For those who are unfamiliar with the term "manufacturing systems", some basic background information is provided here in this first section. Therefore, subsequently the class of *manufacturing systems* is introduced in Section 1.1.1, before describing the subclass of *machine tools* in Section 1.1.2. Note that this thesis was elaborated in close collaboration with seven small- and medium-size machine tool builders, which is why the discussion concentrates on the subclass of machine tools. Also, note that machine tools probably represent the most well-known subclass of manufacturing systems.

1.1.1 Manufacturing systems

Manufacturing systems fall into the more general class of *production systems* [Gro15]. The purpose of production systems is to transform *resources* into *goods* by means of one

or more *production processes*. Hereby, both resources and goods can be (solid, liquid, or gaseous) *material*, (electric, pneumatic, pneumatic, thermal, kinetic, kinematic, etc.) *energy*, and *information* or any combination of them, while the production processes describe the transformation. In contrast, the more specific manufacturing systems typically produce *solid material goods* only such as metal gears or motor blocks. Furthermore, Warnecke and Westkämper [WW10] as well as the DIN 8580[1] provide a classification of related production processes shown in Figure 1.1.

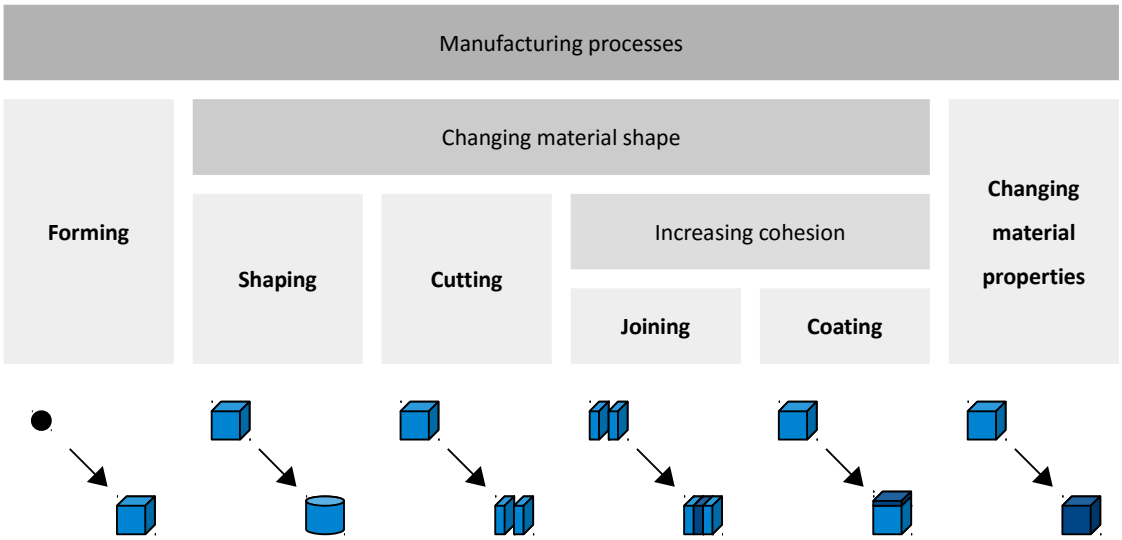


Figure 1.1: Production processes implemented by manufacturing systems [WW10].

According to the classification, in principle six main groups of production processes can be distinguished, which are implemented typically by manufacturing systems: (1) *Forming*, (2) *shaping*, (3) *cutting*, (4) *joining*, (5) *coating* and (6) *changing material properties*. Hereby, forming is concerned with the production of solid bodies for example from the suitable combination of fluids and powders. Shaping, on the other hand, refers to changing the shape of material while keeping its mass. Instead, cutting implies changing the shape of material while reducing its mass. Then, joining refers to creating solid connections between two or more independent solid bodies. Furthermore, coating represents superimposing thin layers of shapeless material onto solid bodies. Finally, changing material properties denotes improving material characteristics such as hardness for future use. One key characteristic of the production processes is that both input and output

[1] http://www.beuth.de/en/standard/din-8580/65031153

are determined mostly by their geometry, while only forming and changing material properties represents an exception.

1.1.2 Machine tools

Machine tools represent one specific subclass of manufacturing systems, which is concerned with only a subset of production processes introduced in the previous section. Hirsch [Hir16] as well as the DIN 69651-1[2] provide a refined classification of related production processes shown in Figure 1.2. Note that this thesis mainly refers to metal processing machine tools as opposed to – for example – wood processing machine tools.

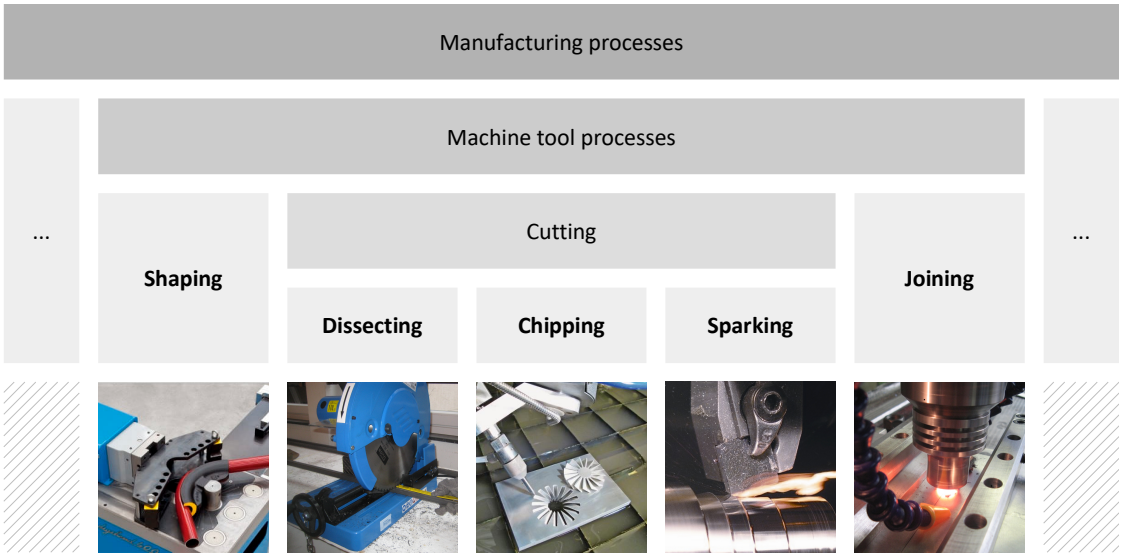


Figure 1.2: Production processes implemented by machine tools [Hir16].

According to the refined classification, metal processing machine tools typically employ three main groups of production processes: (1) *Shaping*, (2) *cutting*, and (3) *joining*. Shaping can be implemented for example by pressing or hammering the material. Cutting, on the other hand, can be subdivided into (a) *dissecting*, (b) *chipping*, and (c) *sparking*. Dissecting can be achieved for example by shearing, while chipping can performed by means of grinding or milling, and sparking can be realized by electrochemical processes. Finally, joining can be implemented by means of welding or soldering. Furthermore, one can distinguish between the *workpiece*, that is being processed by a

[2] http://www.beuth.de/en/draft-standard/din-69651-1/2361625

machine tool, and the *tool* itself, that is being used. For example, the workpiece can be a metal blank, while the tool can be a hammer or driller. Typically, machine tools also are able to exchange their tools during system operation such that more flexible production processes can be implemented using the same equipment / platform.

1.2 Recent trends

In recent years three major trends have been observed in the manufacturing systems domain, which are expected to have a significant impact on the way we design and develop such systems: Increasing *system complexities* (see Section 1.2.1), shifting *engineering efforts* (see Section 1.2.2), and high *commissioning costs* (see Section 1.2.3). In the following, the trends as well as their implications are explained in more detail.

1.2.1 System complexities

In 2013, Gausemeier et al. [GTD13] described the relation between growing complexity of industrial products and the performance of discipline-specific engineering methods over time. Note that manufacturing systems (see Section 1.1.1) and machine tools (see Section 1.1.2) represent industrial products according to their definition. The relation between product complexity and method performance is depicted in Figure 1.3.

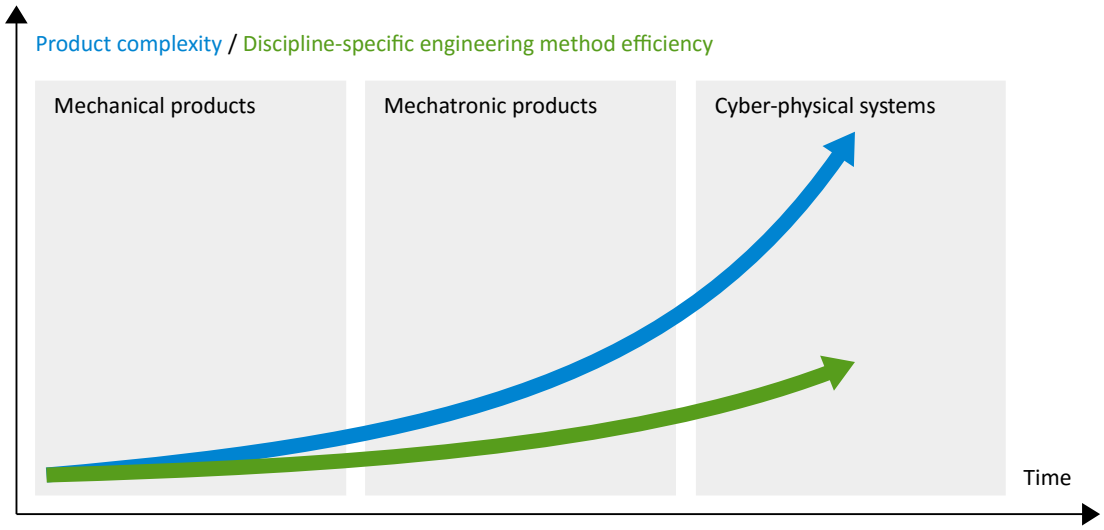


Figure 1.3: Relative product complexity and engineering method efficiency [GTD13].

Industrial products have evolved over time from purely *mechanical products* (comprising only mechanical elements) over *mechatronic products* (adding electrics, electronics, and information processing) to *cyber-physical systems* (adding networking capabilities). Along this evolution both the product complexity and the performance of discipline-specific engineering methods have increased. However, a much stronger growth of product complexity can be observed indicating that discipline-specific engineering methods are not suitable for the development of today's and future industrial products anymore. Consequently, Gausemeier et al. argue for novel multidisciplinary engineering approaches that embrace all relevant aspects of the system under development.

1.2.2 Engineering efforts

Then, in 2007 Reinhart et al. [RW07] reported a drastic shift in engineering efforts for the development of mechatronic and cyber-physical production systems. Again, note that manufacturing systems (see Section 1.1.1) and machine tools (see Section 1.1.2) represent mechatronic and cyber-physical production systems according to their definition. The main result of the study is provided in Figure 1.4.

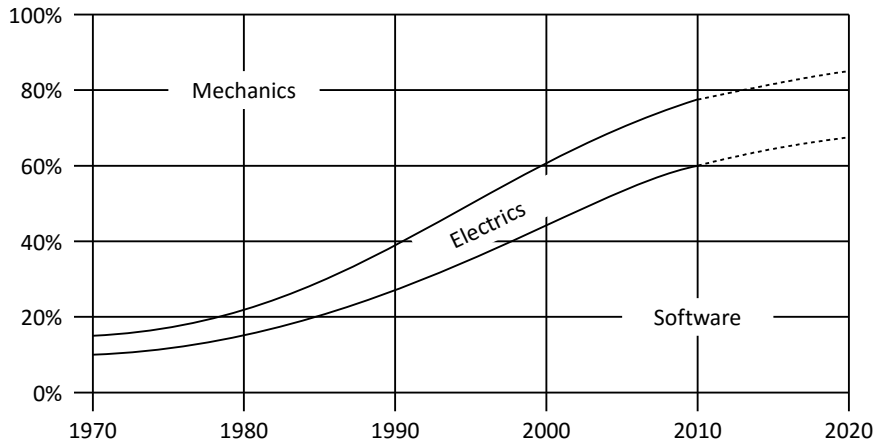


Figure 1.4: Relative efforts of mechanical, electrical and software engineering [RW07].

In the 1970ies mechanical engineering required more than 80% of the engineering effort, while electrical engineering and software engineering took less than 20% only. Then, in the 1990ies the share of mechanical engineering was reduced already to 60% of

the engineering effort, while electrical engineering claimed slightly more than 10% and software engineering demanded slightly less than 30%. Today, the effort for mechanical engineering requires approximately 20%, while electrical engineering has grown to almost 20% and software engineering claims around 60%. In summary, a large portion of the engineering efforts has shifted from mechanical engineering to the other engineering disciplines, which needs to be reflected by the engineering approaches.

1.2.3 Commissioning costs

Finally, the same study by Reinhart et al. [RW07] also observed rather high commissioning cost in industrial practice. Hereby, commissioning refers to the physical assembly of the mechatronic production system, the installation of the control software, the execution of test runs, and the removal of implementation bugs until the test runs are passed. The main result of the study is shown in Figure 1.5.

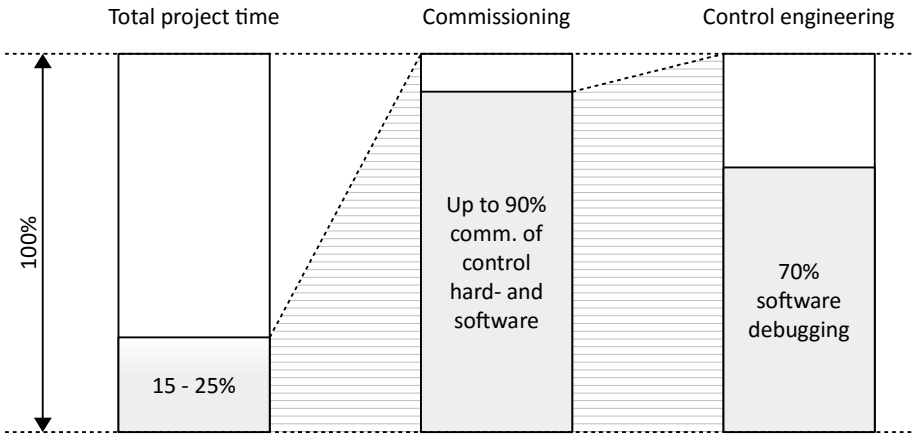


Figure 1.5: Relative efforts of control hard- and software commissioning [RW07].

Commissioning usually takes between 15% and 25% of the total project time. Moreover, 90% of the commissioning effort goes into control hardware and control software commissioning. Finally, 70% of the control hardware and control software commissioning efforts are claimed by software debugging. Consequently, more than half of the commissioning efforts are required for software debugging. The authors argue that novel engineering methods are required, which allow one to reduce the commissioning costs. As a solution, the authors propose the idea of *virtual commissioning*, which allows one

to perform control hardware and/or software commissioning respectively debugging with respect to a computer-based model of the electro-mechanical machine.

1.3 Practical challenges

Subsequently, prior to the actual research work of this doctoral thesis an empirical study [HRZ13] was performed with seven small- and medium-size manufacturing system builders to also learn about their engineering processes as well as practical challenges, which hinder dealing with the increasing system complexity (see Section 1.2.1), the shifting discipline efforts (see Section 1.2.2), and the high commissioning costs (see Section 1.2.3). A slightly revised version of the study's main result is provided in Figure 1.6.

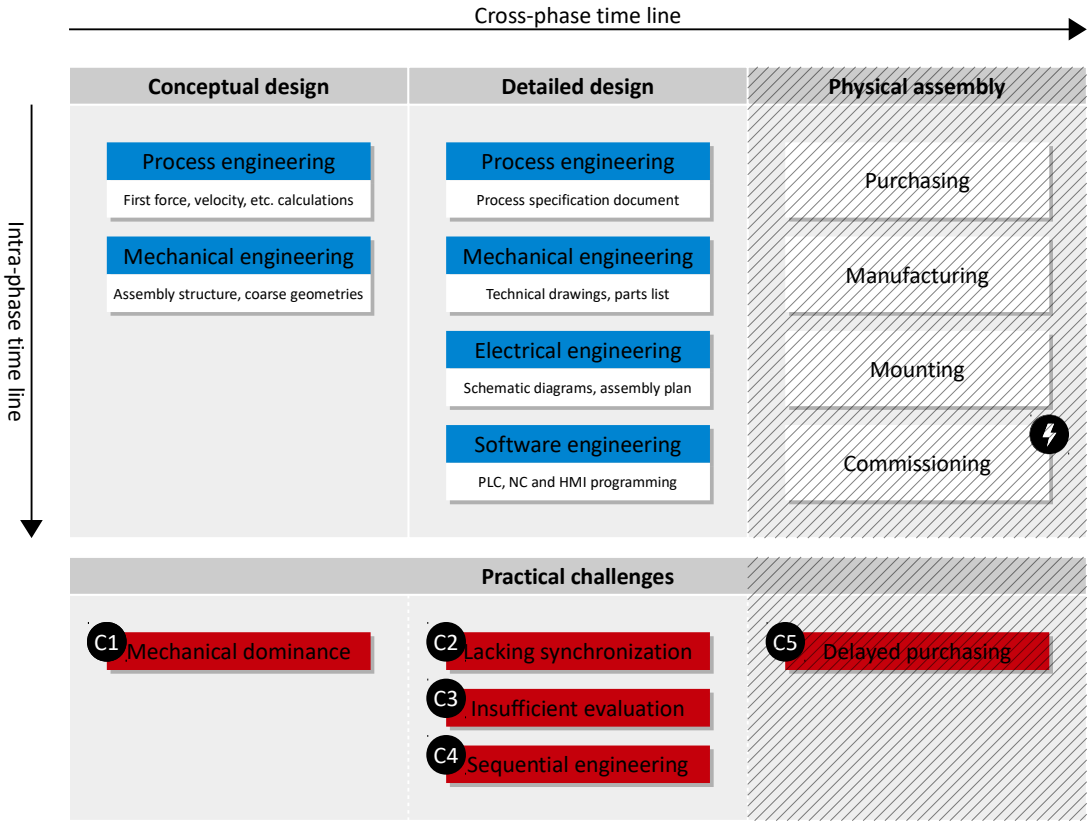


Figure 1.6: Practical challenges in manufacturing system engineering [HRZ13].

In accordance to Pahl and Beitz [PBFG06], the observations showed that manufacturing system engineering projects are split typically into three phases, namely (1) a conceptual design phase, (2) a detailed design phase, and (3) a physical assembly phase. The *conceptual design phase* starts with process engineers decomposing the manufacturing problem (e.g. manufacturing gears from blanks) into process steps (e.g. first milling, then grinding the gear teeth) and performing first force and velocity calculations. Then, mechanical engineers define the assembly structure as well as the coarse geometries and motion profiles for achieving the defined manufacturing process. If the conceptual design is accepted, the detailed design phase is initiated. The *detailed design phase*, again, starts with process engineers working out a detailed process specification. Then, mechanical engineers elaborate the technical drawings of mechanical parts and derive part lists. Afterwards, electrical engineers construct schematic diagrams and assembly plans of electrical components, from which, again, part lists are derived. Finally, software engineers develop programmable logic controller (PLC), numerical control (NC), and human-machine interaction (HMI) components. If the detailed design is accepted, the physical assembly phase is initiated. The *physical assembly phase* starts with purchasing third-party mechanical and electrical components from the part lists. Then, proprietary mechanical and electrical components are manufactured in-house. Thereafter, the purchased and manufactured physical components are mounted. Finally, the software components are installed onto the computing equipment, before first commissioning tests are performed. As noted previously (see Section 1.2.3), during commissioning typically design flaws are discovered which are resolved in a costly debugging process. Consequently, practitioners were asked about the main reasons for commissioning failure. From the obtained answers the following five key *challenges* have been extracted, which need to be tackled by future approaches for cyber-physical manufacturing system engineering: Mechanical dominance (see Section 1.3.1), lacking synchronization (see Section 1.3.2), insufficient evaluation (see Section 1.3.3), sequential engineering (see Section 1.3.4), and delayed purchasing. This doctoral thesis concentrates on the first four challenges leaving the fifth challenge for future research.

1.3.1 Mechanical dominance

Firstly, *mechanical dominance* refers to the observation that during the conceptual design phase typically only process and mechanical engineers are involved in the projects, while electrical and software engineers do not enter before starting the detailed design phase. Consequently, the conceptual design and, hence, the detailed design are dominated by mechanical design decisions. However, the mechanical design decisions might have a negative impact on the overall engineering effort, system quality, and project cost. For

example, the engineering effort might increase or the system quality might decrease because existing/tested electrical and software components cannot be reused with the mechanical design.

Conclusion

Ideally, modern engineering methods, techniques, and tools enable electrical and software engineers to contribute already during the conceptual design phase. Then, early design decisions can be taken with respect to the expected impacts on all engineering disciplines involved. Consequently, negative impacts can be avoided and synergies can be exploited more easily.

1.3.2 Lacking synchronization

Then, *lacking coordination* refers to the observation that during the detailed design phase (i.e. after including all engineering disciplines) design decisions might not be propagated between mechanical, electrical, and software engineers. Consequently, design decisions taken in one discipline might not be reflected by the other disciplines despite their potential impact. For example, sensors added and/or removed in the mechanical and electrical design have an impact on the control software because different information might be available about the physical system state and different control strategies might be required.

Conclusion

Ideally, modern engineering methods, techniques, and tools enable all design decisions taken in one discipline to be propagated to all other disciplines as soon as possible. Consequently, the other disciplines are able to react to the design decisions instantaneously. In particular, the other disciplines can adapt their discipline-specific designs much earlier to the new circumstances saving superfluous engineering efforts. Note that this approach might entail much more iterations, but with much smaller scope.

1.3.3 Insufficient evaluation

Subsequently, *insufficient evaluation* refers to the observation that during the detailed design phase (i.e. after including all engineering disciplines) quality assurance for the mechanical, electrical, and software design is not performed sufficiently. Consequently, the developed design might include design flaws, which cannot be detected and resolved until physical commissioning. However, detecting design flaws during physical commissioning

can be costly, especially, in case the mechanical design has to be changed, which might entail changes to the electrical and the software design. For example, one might notice during physical commissioning that the system cannot be operated without collisions and, thus, considerable damage to physical components.

Conclusion

Ideally, modern engineering methods, techniques, and tools enable quality assurance to be performed frequently on the designs such that design flaws can be detected as early as possible. Consequently, design flaws can be resolved much faster reducing their impact on the overall system design and, hence, the overall costs of the engineering project. Furthermore, note the close relation between this goal and the idea of virtual commissioning [RW07].

1.3.4 Sequential engineering

Finally, *sequential engineering* refers to the observation that during the detailed design phase mechanical engineering is performed before electrical engineering, and electrical engineering is performed before software engineering. Consequently, the complete mechanical design must exist before the electrical design is elaborated and the complete electrical design must exist before the software design is elaborated. However, mechanical and electrical design flaws might not be detected until the software design is available. For example, the software design might require an additional sensor, whose spatial extent entails substantial mechanical and electrical design revisions.

Conclusion

Ideally, modern engineering methods, techniques, and tools enable all mechanical, electrical, and software engineering to be performed in parallel. Consequently, design decisions taken in one discipline can be considered much earlier by the other disciplines and negative impacts can be avoided as well as synergies can be exploited more easily. Furthermore, costly design revisions due to late feedback can be avoided.

1.4 Claimed contributions

To tackle the practical challenges in the engineering and – more specifically – the conceptual design of cyber-physical manufacturing systems (see previous Section 1.3) this

doctoral thesis claims six contributions. Figure 1.7 provides an overview of these contributions and their relationships.

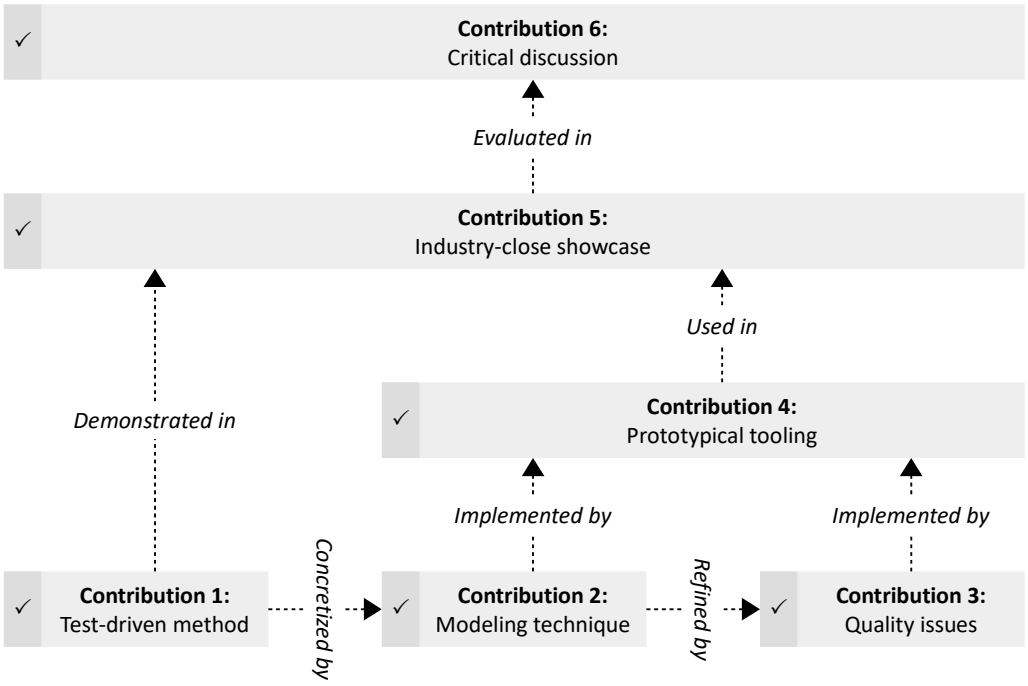


Figure 1.7: Overview of the claimed contributions and their relationships.

In the first step, (1) a test-driven method for the conceptual design of cyber-physical manufacturing systems is proposed fostering initial requirement, material, manufacturing process, and test specification, subsequent mechatronic and discipline-specific architecture, behavior, and shape specification, fundamentally iterative and incremental development, as well as frequent constraint- and test-based verification and validation of iterations and increments (see Section 1.4.1). Then, to realize such test-driven method (2) an interdisciplinary modeling technique is devised covering everything from customer requirements and manufacturing processes to part geometries and control behaviors as well as an underlying formalism defining the syntactic and semantic relations of the design content (see Section 1.4.2). Furthermore, to support frequent verification and validation activities (3) a taxonomy of syntactic and semantic quality issues is provided, which allow one to assess the completeness and consistency of design content, as well as a formal definition of the underlying structural and behavioral constraints for au-

tomation purposes (see Section 1.4.3). Subsequently, to enable a practical evaluation of the previous ideas (4) prototypical tooling is developed including complementary, but integrated views onto and graphical editors for the design content as well as access to the results of continuous syntactic and user-triggered (i.e. spontaneous) semantic quality assessment (see Section 1.4.4). Then, (5) the approach is evaluated experimentally with an industry-close showcase, which covers the selective transportation and manipulation of different kinds of material, comprises three mechatronic subsystems, and comes with publicly accessible documentation (see Section 1.4.5). And finally, (6) a critical discussion of the feasibility of the test-driven method, the validity of the modeling technique, and the relevancy of the quality issues is provided, which is based on data collected during the experiment (see Section 1.4.6). In the following, each contribution is described in more detail.

1.4.1 Test-driven method

Initially, in Chapter 3 a novel *test-driven method* is proposed for the conceptual design of cyber-physical manufacturing systems. As known from the software domain, the proposed test-driven approach promotes writing test cases before working on the details of their implementation. Consequently, the verification of implementation details can be automated to a large portion and verification can be performed much earlier in the development process assuming that respective test cases have been specified. Furthermore, the approach promotes developing the system in increments and testing the increments thoroughly before extending the system scope. As a result, working virtual prototypes are available much earlier in the process and validation can be performed in collaboration with the customers and other stakeholders. Opposed to test-driven methods in the software domain, the proposed approach includes dedicated activities for the conceptual design of cyber-physical manufacturing systems such as (input/output) material, manufacturing process, and (mechanical) part specification.

1.4.2 Modeling technique

Then, in Chapter 5 a novel interdisciplinary *modeling technique* is devised for capturing the design content generated by the test-driven method from the previous section. In particular, the modeling technique provides an object-oriented data model for documenting requirement, manufacturing process, and test specifications, mechatronic and discipline-specific architectures, part geometries and motion profiles, as well as energy flow and signal flow (i.e. control) behaviors. Furthermore, the object-oriented data model is underpinned with formal syntax and semantics, which is derived from previous work

explained in Chapter 4. In particular, the formal syntax and semantics gives precise meaning to the elements of the design content and their relations. Consequently, not only the structural dependencies among the elements of the design content can be analyzed automatically, but also the behavioral semantics regarding energy and signal flow as well as motion of mechanical parts.

1.4.3 Quality issues

Subsequently, in Chapter 6 a taxonomy of *quality issues* is provided that can be detected over design content expressed using the modeling technique from the previous section. In accordance to the formal syntax and semantics, this thesis distinguishes between syntactic and semantic quality issues. The syntactic issues cover the completeness and consistency of the design content. Consequently, the engineers can be informed about missing information (e.g. the geometry of a mechanical part) and contradictions within existing information (e.g. the data types of two connected signal ports). Similarly, the semantic issues cover aspects like the correct implementation of requirements and manufacturing processes or the ability to operate the cyber-physical manufacturing system without collisions of mechanical parts. Overall, a precise definition of the quality issues is emphasized such that they can be evaluated automatically and implemented by potential tool vendors easily.

1.4.4 Prototypical tooling

Thereafter, in Chapter 7 *prototypical tooling* is developed that tightly integrates the modeling technique and the automated evaluation of the quality issues from the previous two sections. The prototypical tooling represents an answer to how the modeling technique and the taxonomy of quality issues could be exploited in practice. In particular, the tooling provides fourteen different views onto the design content, which are distributed across a modeling and a testing screen. Thereby, one key challenge lies in displaying the relations between logical and geometrical elements such as component architectures, state machines, part geometries, and motion profiles. Furthermore, the tooling supports four graphical notations for different elements of the design content as well as one three-dimensional view onto the system geometry. Finally, the tooling integrates continuous evaluation and display of syntactic issues as well as spontaneous evaluation and display of semantic issues.

1.4.5 Industry-close showcase

Then, in Chapter 8 this thesis describes the result of applying the test-driven method, the interdisciplinary modeling technique, the quality issues, and the prototypical tooling to an *industry-close showcase*: The pick and place unit installed at the Institute for Automation and Information Systems, Technische Universität München[3]. In particular, all elements of the conceptual design are outlined from the overall system and the system-level (mechatronic) modules to the module-level components. For each system, system-level module, and module-level component information about the underlying design decisions is provided from the requirement, manufacturing process, and test as well as the mechatronic system, mechanical, electrical, and software engineers. In essence, the chapter demonstrates how conceptual design documents could look like in the future when using the proposed interdisciplinary approach. Implicitly, the showcase also proves the maturity and usability of the prototypical tooling.

1.4.6 Critical discussion

Finally, in Chapter 9 this thesis *discusses* the results of the experiment *critically* with respect to several research questions. In particular, the feasibility of the test-driven design method for the domain of cyber-physical manufacturing systems is assessed. Then, the validity of the interdisciplinary modeling technique and its formal foundation is evaluated. Furthermore, the relevancy of the quality issues covered by the proposed taxonomy is contemplated. And finally, the potential threats to the internal and external validity of the study are analyzed. To show the feasibility of the test-driven design method, data collected during usage of the prototypical tooling is studied with respect to modeling and simulation activities. In contrast, to determine the validity of the interdisciplinary modeling technique, the design content and its execution semantics are compared to the original SysML documentation as well as the behavior of the physical system. Lastly, to demonstrate the relevancy of the selected quality issues, the tool data is evaluated with respect to the appearance and disappearance of quality issues.

1.5 Intended audience

At last, the audience is described this doctoral thesis is intended for. The first group are *system builders*, i.e. people constructing cyber-physical manufacturing systems (see Section 1.5.1). The second group are *tool providers*, i.e. people developing software

[3] http://www.ais.mw.tum.de/en/

tools that are used by system builders for designing cyber-physical manufacturing systems (see Section 1.5.2). Finally, the third and last group are *scientific researchers*, i.e. people working on theoretical and practical contributions in the field of cyber-physical manufacturing system design. Figure 1.8 provides an overview of the intended audience and its relations to the contributions of this doctoral thesis. Note that additional relations might exist. However, the discussion concentrates on those relations only which are considered to be most important.

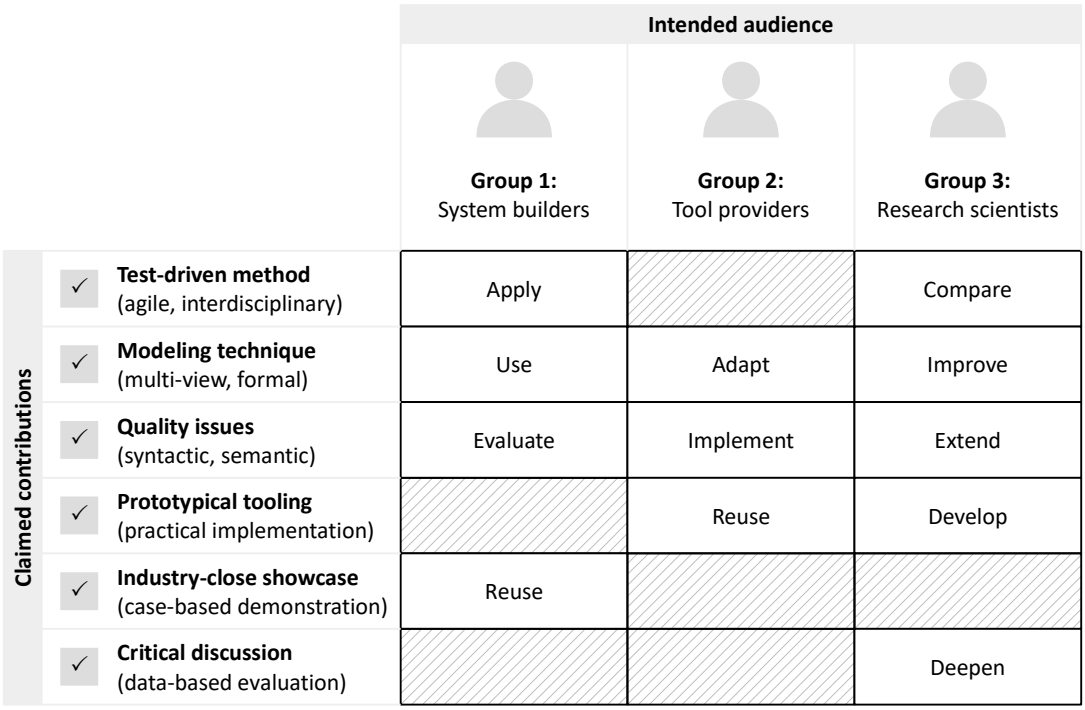


Figure 1.8: Overview of the indented audience of this doctoral thesis.

In the following, the intended audience is explained in more detail. In particular, the description focuses on how each group of people introduced previously can benefit from the different contributions claimed by this doctoral thesis (see Section 1.4). Note that here intentions are stated only. Still, this section should help reading this doctoral thesis from different points of view and different professional roles.

1.5.1 System builders

System builders are expected to benefit from this doctoral thesis in several ways: Most importantly, this thesis provides novel ideas on how to organize the conceptual design process in a more agile fashion (see Chapter 3). In particular, the proposed approach relies on iterations and increments for developing a common understanding of the design problem as well as the conceptual solution already early in the conceptual design process. Hereby, the increments allow one to start with a reduced initial design scope, which is extended gradually until the complete final design scope is reached. In contrast, the iterations allow one to revise the documented understanding of a given increment until a common understanding is reached. Consequently, misunderstandings between the customer and the system builder as well as between the different engineering disciplines involved in the design process can be identified and resolved much earlier than before. Similarly, actual design flaws introduced by the different engineering disciplines can be uncovered and resolved much earlier also. Resolving misunderstandings and/or design flaws already early in the process, in turn, reduces their impact on the overall design process and, therefore, the project costs. Furthermore, this thesis provides novel ideas on how to describe system requirements, manufacturing processes, and test cases as well as implementation details (see Chapter 5) such that the implementation details can be verified automatically both on a syntactic and a semantic level (see Chapter 6). Automatic verification, in turn, allows one to reduce the verification costs as well as to increase the verification frequency leading to better quality in shorter time and with lower budget. Finally, this thesis includes an industry-close showcase (see Chapter 8) demonstrating how the novel approach can be applied in practice. The industry-close showcase can serve as a blueprint of how future design documents for cyber-physical manufacturing systems could look like. System builders could use this blueprint for structuring their own design documents more systematically. Furthermore, the showcase also can help explaining the ideas and principles behind the proposed approach to engineering personnel.

1.5.2 Tool providers

Similarly, tool providers are expected to benefit from this doctoral thesis in several ways: Most importantly, this thesis provides novel ideas on how to model cyber-physical manufacturing systems from several viewpoints as well as how to check the syntactic and semantic quality of the models automatically. In particular, the viewpoints include customer requirements, manufacturing processes and test cases as well as implementation details such as component architectures, part geometries, motion profiles, and

energy/signal flow behaviors. Consequently, the proposed approach provides a broad coverage of design information which is relevant during the conceptual design phase. Some commercial tools also provide such broad coverage of design information. However, these tools are lacking an underlying formalism describing the syntactic and semantic relations between the design elements. As a result, the tools only support limited checks of the syntactic and semantic quality of the design elements. Typically, the checks are limited to a single viewpoint (e.g. the signal input/output behavior of the software controller or the behavior of mechanical parts under physical stress) or a subset of viewpoints (e.g. the motion of mechanical parts given some motion profiles). To overcome this situation, the proposed multi-view modeling technique (see Chapter 5) comes with an underlying formalism describing the syntactic and semantic relations of the design elements within single and between different viewpoints. Furthermore, this thesis provides formal definitions of the syntactic and semantic quality checks (see Chapter 6) which can be applied to the design elements. Most importantly, the semantic quality checks cover the behavior of the cyber-physical system in different scenarios, which has to satisfy certain geometrical as well as energy and signal flow related constraints. Tool providers can use the results of this doctoral thesis to improve the syntactic and semantic foundation of their own models and quality checking mechanisms. Furthermore, tool providers can use the ideas provided on prototypical tooling (see Chapter 7) to integrate syntactic and semantic quality checks over their multi-view models into their user interfaces.

1.5.3 Scientific researchers

Finally, research scientists are expected to benefit from this doctoral thesis in the following ways: First, this doctoral thesis contains novel ideas on how to approach the conceptual design of cyber-physical manufacturing systems in a more agile, interdisciplinary fashion (see Chapter 3). However, the contribution is limited to showing that such agile, interdisciplinary approach is feasible in the cyber-physical manufacturing systems domain. On the other hand, the questions remains open how efficient the proposed method is in comparison to other agile and non-agile methods. Then, this doctoral thesis proposes a novel formal modeling technique that covers a wide range of design information from customer requirements, manufacturing processes, and test cases to implementation details such as (cyber-physical) component architectures, part geometries, motion profiles, as well as energy and signal flow behaviors (see Chapter 5). While this contribution represents a significant advance in unifying the syntax and semantics of the design models and languages of the different engineering disciplines there still is huge space for improvement. In particular, the question remains how intuitive the prosed modeling technique is for practitioners and whether more intuitive and, hence, appropri-

ate modeling techniques exist. Similarly, huge potential lies in extending the taxonomy of quality issues and their formal definitions (see Chapter 6). While this doctoral thesis covers and structures a significant number of quality issues, the evaluation is limited to showing their practical relevance. However, it remains unclear how complete the taxonomy of quality issues is. Hence, evaluating the completeness of the taxonomy and extended the taxonomy if necessary remain a major challenge, for which this doctoral thesis can serve as a solid foundation. Subsequently, a key contribution of this doctoral thesis is the prototypical tooling (see Chapter 7). The prototypical tooling and its built-in tracking allows one to perform a case-based empirical study easily. In particular, the built-in tracking of user actions has proven to deliver useful data for empirical analysis. Other researchers might adopt this tracking-based approach, also because it represents no overhead to the study participants. Finally, the critical discussion based on tracking data (see Chapter 9) can serve as a blueprint for the evaluation of other design methods and modeling techniques. It will be interesting to see how the evaluation based on tracking data will evolve in the future. Possibly, standard measurement and interpretation techniques will be developed to obtain more reliable evaluation outcomes.

1.6 Summary and outlook

This first chapter explained what cyber-physical manufacturing systems are (see Section 1.1), what trends can be observed in the manufacturing system domain in the past years and decades (see Section 1.2), what practical challenges manufacturing system builders are facing today (see Section 1.3), what the claimed contributions of this doctoral thesis are to tackle the challenges (see Section 1.4), and who the intended audience of this doctoral thesis is (see Section 1.5). The following chapter describes related work on the conceptual design of cyber-physical manufacturing systems and compare existing techniques to the proposed approach.

2 Differentiation

In principle, various approaches exist that can be used for the conceptual design of cyber-physical manufacturing systems. To provide an overview over the field, the most prominent approaches are summarized in Section 2.1, which have been proposed in the past and get applied in industry today. In particular, their strengths and weaknesses are discussed with respect to the practical challenges described in the previous chapter. Then, remaining problems are pinpointed in Section 2.2, which have not been addressed sufficiently by existing approaches so far and which differentiate the existing body of scientific work from the contributions of this doctoral thesis. Finally, the research objectives of this doctoral thesis are summarized in Section 2.3, which aims at closing the gaps left by the state of the art that is available today.

2.1 Related work

The related work is drawn from a number of review and survey papers published over past twenty years. In 1998, Hsu and Woon [HW98] provide one of the first surveys on conceptual design of mechanical products. Note that back then the integration of the different engineering disciplines was not as prominent as it is today. Two years later Regli et al. [RHAS00] summarize work on design rationale systems. Compared to classical design techniques, design rationale systems concentrate on the rationale behind design decisions rather than the design itself. Subsequently, Sinha et al. [SPLK01] describe a range of modeling and simulation methods, which can be applied to system design in general. In particular, the authors already embrace the multidisciplinary nature of the design task. Then, Wang et al. [WSX+02] review techniques for collaborative conceptual design, i.e. conceptual design in the context of teams and organizations. Hereby, the techniques specifically focus on the parallel evolution and synchronization of the design content. More recently, Jimeno and Puerta [JP07] describe applications of virtual reality techniques in the design of manufacturing processes. Specifically, the authors focus on assembly problems and human-machine interaction. In the same year, Estefan [Est07] publishes a comprehensive survey of model-based systems engineering methodologies. In particular, the author focuses on the design processes rather than the design deliverables.

One year later, Erden et al. [EKvB+08] summarize function modeling techniques. Compared to other design techniques, function modeling techniques focus on the function (or the purpose) of the designed system or product rather than its actual implementation. Finally, in 2013 Chandrasegaran et al. [CRS+13] provide a comprehensive survey on knowledge representations in product design. Hereby, the term *knowledge* subsumes everything from function respectively purpose to design rationale to implementation.

In the following, the most important approaches for the conceptual design of cyber-physical manufacturing systems are discussed. Thereby, the description distinguishes between diagram-based techniques (see Section 2.1.1), physics-based techniques (see Section 2.1.2), component-based techniques (see Section 2.1.3), matrix-based techniques (see Section 2.1.4), function-based techniques (see Section 2.1.5), and ontology-based techniques (see Section 2.1.6) as well as commercial tools (see Section 2.1.7). As mentioned previously, particularly the strengths and weaknesses of the techniques and / or tools are focused with respect to the practical challenges posed in the previous chapter.

2.1.1 Diagram-based techniques

One of the most popular movements in conceptual design, which is rooted in object-oriented programming [Ren82] and software design [RRE91], are what in this thesis is called diagram-based techniques. Typically, diagram-based techniques provide different types of diagrams for describing different aspects of the system under development at different phases of the development process. Here, (1) the *Unified Modeling Language*, (2) the *Systems Modeling Language*, and (3) the *Specification Technique for the Consistent Description of Manufacturing Operations and Resources* are presented.

Unified Modeling Language (UML)

The most popular diagram-based technique for object-oriented analysis and design of software systems is the *Unified Modeling Language* (UML) [BJR+96]. In the most recent version the UML provides 14 different *structural* and *behavioral diagrams*. The structural diagrams describe static (i.e. time-independent) aspects of the system under development. The static aspects include, for example, the *classes* of objects the software system is composed of or the *packaging* of these classes into reusable units. In contrast, the behavioral diagrams describe dynamic (i.e. time-dependent) aspects of the system under development. The dynamic aspects comprise, for example, the *use cases* of the software system from the perspective of different actors (e.g. the machine operator) or the *activities* performed by objects of the software system (e.g. updating the database). Finally, the UML provides a clear methodology for the development of software systems

leading from customer requirements to implementation details [BME+07]. Furthermore, a variety of commercial tools exist on the market.

An advantage of the UML is that it provides diagrams for many different aspects of the system under development. Consequently, a large body of design knowledge can be captured graphically. However, one major drawback of the UML is the lack of formal semantics leading to misinterpretations and inconsistencies, which are hard to uncover automatically. Furthermore, the UML mostly targets software systems neglecting physical aspects such as the geometry and spatial motion of components.

Systems Modeling Language (SysML)

Another popular diagram-based technique for the conceptual design of systems in general, which can be considered as the de-facto industry standard, is the *Systems Modeling Language* (SysML) [Hau06b]. At its core, the SysML is based on the UML described in the previous section. However, compared to the UML the SysML provides only nine different diagrams, which partially overlap with the UML diagrams. In particular, the SysML reuses the specification of *use cases* from the perspective of actors (e.g. machine operators) or *activities* performed by the system during operation (e.g. manipulating workpieces). Furthermore, the SysML introduces new diagram types for specifiying the *requirements* of the system under development (e.g. manufacturing some product from given resources), designing the *component* structure of the system (e.g. tool and workpiece fixtures), or reusing *parameterized* components (e.g. different tool sizes). Note that compared to the UML and its object-oriented perspective, the SysML takes a more component-oriented perspective on the system under development, where components interact based on flows of material, energy, and information.

An advantage of the SysML compared to the UML is that requirements can be expressed explicitly. Furthermore, component reuse is fostered using an extended parameterization mechanism. Finally, the SysML adopts a component-oriented perspective, which is more prominent in systems thinking than the object-oriented perspective of the UML. However, similar to UML a rigorous formal foundation is missing rendering model interpretation ambiguous and error-prone. Also, the language misses important design knowledge such as component geometries.

Specification Technique for the Consistent Description of Manufacturing Operations and Resources (CONSENS)

Finally, a more recent diagram-based technique, which is tailored to cyber-physical manufacturing system domain, is the *Specification Technique for the Consistent Descrip-*

tion of Manufacturing Operations and Resources (CONSENS) proposed by Rehage et al. [RBG14]. The CONSENS approach provides eight different views onto (or so-called *partial models* of) the system under development. In the initial project phases the *environment*, *application scenario*, *requirement*, and *system of objectives* models help clarifying the task. For example, the environment model contains the system under development as well as relevant *spheres of influences* such as the user or adjacent technical systems. In contrast, the application scenario model describes high-level *states* of the system under development as well as *events* and possible state *transitions*. Then, in later project phases the *function*, *active structure*, *behavior*, and *shape* models allow one to capture implementation details. For example, the active structure model decomposes the system into *system elements*, their *attributes*, and the *relations* between the system elements. Furthermore, system elements can be combined to from *logical groups*. In contrast, the behavior model describes the system *states*, the *activities* that trigger state transitions, and state *sequences* that describe the interaction between system elements.

An advantage of CONSENS compared to the UML or the SysML is that it includes additional means for modeling the system context, usage scenarios, function hierarchies as well as the geometric appearance of components. While the approach captures the modeling needs for mechatronic product development more appropriately, again a rigorous formal foundation is missing. In particular, the semantics of the different diagrams and modeling elements are ambiguous, which hinders the automation of quality assessment and verification tasks.

2.1.2 Physics-based techniques

The next category of techniques is concerned with describing directly the physical objects and behaviors that comprise the designed system and / or product. Note that this category is concerned mostly with mechanical and electrical phenomena, while software aspects are ignored mostly. Consequently, the techniques are more appropriate for the conceptual design of electromechanical rather than cyber-physical systems. Nevertheless, the techniques contain interesting ideas worth mentioning in the context of this thesis. Also, some of the subsequent techniques are based on the ideas and concepts introduced in this category. In the following (1) *multi-body system dynamics*, (2) the *qualitative process theory*, and (3) *qualitative kinematics* are presented.

Multi-body system dynamics

One of the most well-known techniques for the design of mechanical systems is *multi-body system dynamics*. The technique has been developed originally in the field of

aerospace engineering in the late 1970s for the mechanical design of satellites [Sch97]. Then, in the late 1990s the technique has been adopted for machine tool design, e.g. by Reinhart and Weissenberger [RW99] and later by Zäh and Seidl [ZS07]. Mutli-body system dynamics allows one to describe the mechanical system in terms of solid *bodies* and *joints* representing fixed mechanical connections between the bodies. The bodies are described in terms of their *shape*, their *mass*, and their *center of mass*. In contrast, the joints are decomposed into *revolute joints* supporting translation along a fixed axis, *prismatic joints* supporting rotation about a fixed axis, and *spherical joints* supporting rotation about a fixed point. Finally, one can specify *external forces* that are applied to the bodies such as gravity. From the external forces and the joints the *motion* of the bodies and potential *collisions* can be derived. Furthermore, from the collisions *impact forces* can be derived, which, in turn, take effect on the motion of the bodies. While the basic theory only supports solid bodies, extensions exist for describing flexible bodies and their deformation [Sha97].

An advantage of multi-body system dynamics is that mechanical parts and connections can be described rather intuitively in terms of their shape, spatial arrangement, and degrees of freedom with respect to motion. Furthermore, the motion profiles can be derived easily from the underlying kinematic equations. In particular, possible side effects can be discovered, e.g. due to undesired collisions. However, multi-body dynamics does not cover electrical and software components at all. Furthermore, the technique bears a practical design methodology starting with the customer requirements and leading to implementation details systematically.

Qualitative process theory

Another successful technique providing a more abstract view on physical systems than, for example, multi-body system dynamics is the *qualitative process theory*, which has been proposed by Forbus in his doctoral thesis [For84]. As the name states, the theory allows one to describe physical processes qualitatively rather than quantitatively (e.g. using differential equations), while still enabling formal reasoning. The core building blocks of the theory are physical *objects* such as liquids and solids, which are described in terms of physical *quantities* such as temperature or pressure. Each physical quantity, in turn, takes a value from a selected *quantity space* (i.e. an abstraction of concrete values). Finally, physical *processes* are described in terms of the interacting *objects*, general *preconditions* on the relations between and the quantities of the interacting objects as well as imposed *relations* between and *influences* on the quantities of the interacting objects. Relations between quantities can be, for example, *qualitative proportional*. Influences on the quantities, on the other hand, can be *direct* or *indirect*. Formal reasoning,

that is supported by the theory, includes finding all feasible physical processes as well as analyzing the limits of certain quantities.

An advantage of the qualitative process theory is the high level of abstraction. In particular, quantities such as temperature and pressure as well as their relations can be described qualitatively (e.g. *low* and *high*, or *slow* and *fast*). However, the theory does not provide much guidance on the representation of material geometries and motion profiles, which limits the practical use for mechanical engineers. Furthermore, software components and processes cannot be represented well, which limits the general use of the theory for cyber-physical systems. Finally, the theory bears a practical design methodology starting with customer requirements.

Qualitative kinematics

To cope with arbitrary material geometries and motion profiles, Faltings [Fal90] extended the qualitative process theory to form *qualitative kinematics*. The main observation behind qualitative kinematics is that the motion of mechanical parts usually is constrained by joints. Consequently, the author argues that only the actual degrees of freedom of each part have to be considered forming the *configuration space* of the mechanical system. Hereby, each point in the configuration space represents a distinct spatial arrangement of all mechanical parts involved. However, not all points in the configuration space actually yield valid spatial arrangements, e.g. due to overlaps among the part volumes. Consequently, the author decomposes the configuration space into the *free space* (yielding valid configurations) and the *blocked space* (yielding invalid configurations). Furthermore, the boundary between the two subspaces is given by constraints. In particular, the author distinguishes *vertex constraints* concerning a single touch point and *boundary constraints* concerning multiple touch points of part volumes. Then, the author decomposes the free space further into *places*, i.e. configurations with similar touch points and qualitative behavior over the configuration space parameters. Finally, the author introduces the possible *transitions* between the places to from the *place graph*. Hereby, the place graph describes the qualitative kinematic behavior of the mechanical system in terms of place paths.

An advantage of qualitative kinematics over qualitative process theory is that a clear guideline is provided for dealing with part geometries and motion profiles. Also, qualitative kinematics achieves a higher degree of abstraction than multi-body system dynamics due to the concepts of places and place graphs. However, software components and software behavior still cannot be represented well, limiting the general use of the technique for cyber-physical systems. Also, the theory bears a practical design methodology starting with customer requirements.

2.1.3 Component-based techniques

While the physics-based techniques already capture certain implementation-level design knowledge quite well, they lack one important aspect, namely the logical architecture of the system. The logical architecture typically describes a decomposition of the system into modules (or components) and their interactions with the goal to reduce complexity and foster reuse. To address this issue, a wide variety of component-based techniques has been proposed. In the following (1) *Modelica*, (2) the *Mechatronic UML*, and (3) the *Spatio-Temporal Engineering Models* are presented, which are the most well-suited techniques.

Modelica

One of the most popular component-based techniques for the (conceptual) design of cyber-physical systems is the *Modelica* language [MEO98]. The core building block of the Modelica language are so-called *models* (or classes of components), which comprise *parameters*, *subcomponents* (i.e. instances of other models), *connectors*, and *equations*. The parameters represent constants, which have to be defined during model instantiation such as the actual electrical resistance of a concrete electrical resistor. Connectors, on the other hand, represent ports for specifying interactions between components. Connectors are described further using *variables*, which can be observed during simulation such as the voltage and current at electrical pin connectors. Finally, equations relate the parameters and the connector variables of both the parent component and subcomponents. Special expressions are provided for connecting connectors (and their variables respectively). Furthermore, differential equations can be used to relate connector variables such as acceleration and speed. At last, the Modelica language supports the specification discrete-time control logic based on *algorithms* and *state machines*.

An advantage of the Modelica language is that the behavior of cyber-physical systems can be described quite intuitively and accurately. Furthermore, new models can be composed easily from existing models and their suitable parameterization. However, while Modelica is well-suited for developing simulation models for cyber-physical systems, it does not cover the entire engineering process from customer requirements to manufacturing process design, to test specification, to implementation properly. Furthermore, important aspects such as the system geometry cannot be described as first-class model citizens and potential collisions between mechanical parts cannot be studied.

Mechatronic UML

Another component-based technique, which provides formal verification in addition to simulation support, is the *Mechatronic UML* proposed by Burmester et al. [BGT05]. Note that the Mechatronic UML is based on the UML, which has been described in Section 2.1.1, but provides only a reduced set of diagram types and adds underlying formal semantics. In particular, the Mechatronic UML provides three different types of diagrams, namely *component*, *coordination protocol*, and *real-time statechart* diagrams. The component diagrams describe the *components* of the system, their *subcomponent* and *ports*, as well as the *connections* between the ports and *coordination protocols* governing the connections. Then, the coordination protocol diagrams distinguish between *roles*, role *connectors*, which indicate message exchange between the roles, and *quality of service assumptions*, which constrain the message exchange over role connectors. Finally, the real-time statechart diagrams allow one to describe both component and coordination protocol behavior using *states*, *transitions*, *guards*, and *actions*. Alternatively, MATLAB Simulink models (see Section 2.1.7) can be used to also. Finally, the Mechatronic UML supports the verification of coordination protocol behavior with respect to quality of service assumptions and component behavior with respect to coordination protocol behavior.

An advantage of the Mechatronic UML over Modelica is that requirements can be formulated precisely in terms of coordination protocols and quality of service assumptions. Furthermore, implementation details (i.e. component behavior) can be verified with respect to the requirements. Consequently, the consistency checks can be performed automatically and design flaws can be discovered more easily. However, a drawback of the technique is that only message and signal flows can be considered, while aspects like system geometry and motion are neglected. Also, coordination protocols might already include design decisions blurring the distinction to the actual user requirements.

Spatio-Temporal Engineering Models (STEM)

Finally, another component-based technique, which takes into account geometrical aspects of the system under development, are the *Spatio-Temporal Engineering Models* (STEM) proposed by Hummel [Hum09]. At its core, STEM is based on FOCUS [BS01], a component-based technique for the design of embedded software systems. The core concept of STEM is the *spatio-temporal component*, which provides a description of its *syntactic interface* and *semantic interface* (or behavior). The syntactic interface is given by the message *input* and *output ports*. Furthermore, the syntactic interface defines the geometric *parts* of the component, *movers* for changing the translation and orientation

of components, *detectors* for reacting to spatial collision with other components, as well as dynamic port *bindings* for interacting with colliding components. In contrast, the semantic interface specifies the reaction of spatio-temporal components to (energy or data) inputs and collisions. The reactions might include producing some (energy or data) output, moving another component in space, changing the geometry of detectors and parts, or adapting the dynamic bindings with colliding components. Finally, STEM provides a composition operator which can be used to form larger, more complex spatio-temporal components from smaller, less complex units and their interactions.

An advantage of STEM over Modelica and the Mechatronic UML is that geometry, spatial motion, collision, and collision-based dynamic interaction are considered as first class modeling entities. Consequently, geometrical design knowledge can be captured explicitly and its effect on system behavior can be studied. For example, one can determine whether the parts of certain components collide during system operation. However, a drawback of the technique is that customer requirements, manufacturing processes, and test cases are not considered. Consequently, the approach also lacks a clear methodology leading from customer requirements to implementation details systematically.

2.1.4 Matrix-based techniques

While the component-based techniques already provide the means for defining the logical architecture of the system in terms of reusable modules and their interactions, they usually do not take the customer requirements into account. Furthermore, the question remains what is an appropriate architecture for a given design problem. To tackle these issues, matrix-based techniques have been proposed. Matrix-based techniques generally seek to capture certain design knowledge such as customer requirements or system components as well as the relations between them (in terms of an adjacency matrix). Then, different properties of the adjacency matrices are used to guide subsequent design decisions such as the allocation of customer requirements to system components. In the following (1) *quality function deployment*, (2) the *design structure matrix*, and (3) *axiomatic design* are presented, which probably are the most well-known matrix-based techniques.

Quality function deployment

Probably the most well-known and widely used matrix-based technique is the *quality function deployment*, developed in Japan in the late 1960s and early 1970s by Akao [CW02]. Later, Hauser and Clausing [HC88] also coined the term *house of quality*. The main goal of the technique is to increase the focus on the customer requirements

and to derive implementation details systematically. The proposed methodology comprises ten steps. In the first step, the type of *product* is defined, e.g. a vacuum cleaner. Then, in the second step the types of *customers* are specified including the actual end users as well as people potentially influencing the purchase decisions. In the third step, the *customer requirements* are derived. Hereby, one can distinguish between *expressed* and *implicit requirements*. The implicit requirements can be divided further into *basic* (e.g. safety and reliability) and *excitement* (i.e. sales point) *requirements*. Afterwards, in the fourth step an *importance rating* is performed over the customer requirements and customer groups. In the fifth step, a *competitive benchmarking* is developed showing the relation between different products on the market and the customer requirements. Thereafter, in the sixth step the *design parameters* (i.e. observable product characteristics) are derived based on available expertise as well as time and budget constraints. In the seventh step, the *relationship* between the customer requirements and the design parameters is specified using a weight matrix. Then, in the eighth step the *targets* for the design parameters (such as the weight of the vacuum cleaner) are determined, e.g., from an analysis of competitor products. In the ninth step, a *performance benchmarking* is performed to cross-check whether important requirements are addressed by high-scored design parameters. Finally, in the tenth step the *correlations* between the design parameters are estimated in terms of conflicting and supporting relationships, from which points of conflict and trade-off can be derived.

An advantage of the quality function deployment is the strong focus on customer requirements and the competition in the market. Furthermore, specific checks are supported such as the strong relation of important customer requirements with design parameters and the unique selling point of the product with respect to the competition. Finally, the quality function deployment provides a high level of abstraction suitable for early project phases. On the downside, the semantics of the customer requirements (e.g. desired material in- and output) and design parameters (e.g. manufacturing processes and component architectures) as well as the relation and correlation matrices are not defined formally such that inconsistencies cannot be detected easily.

Design structure matrix

Another well-known and widely used matrix-based technique is the *design structure matrix* proposed by Steward [Ste81]. Originally, the technique has been developed for project planners dealing with circular dependencies among project tasks. However, in the following years the design structure matrix has been applied to numerous problems such as system modularization also [Bro01]. The core idea behind the technique is to represent *dependencies* between arbitrary *design variables* (e.g. the passenger capacity

and the total weight of an electrical car) using *matrices*. Hereby, the design variables define both the rows and the columns of the matrix, while the dependencies are stated in the cells. Then, through reordering of rows and columns *block triangular matrices* or even *triangular matrices* can be obtained. Triangular matrices directly determine the order, in which the design variables can be resolved. In the case of block triangular matrices the design variables contain circular dependencies. Consequently, estimates (called *tears*) have to be provided for certain design variables, from which the remaining design variables can be derived. This process might comprise several iterations and reviews until adequate estimates are obtained.

An advantage of the design structure matrix is its high level of abstraction because design variables can be anything from design activities to system components to organizational units. Also, the semantics of the dependencies between design variables can be selected freely. On the other hand, the semantics of a design structure matrix cannot be processed automatically, e.g., for detecting inconsistencies. Also, the technique does not provide any guidance on the types of design variables and dependencies, which need to be considered during cyber-physical manufacturing system design.

Axiomatic design

Another popular matrix-based technique is the *axiomatic design*, originally proposed by Suh for mechanical systems [Suh95] and later extended to systems in general [Suh98] and manufacturing systems specifically [SCL98]. The general idea behind the technique is to represent the designed system in terms of *functional requirements*, *design parameters*, and *process variables*. Hereby, the functional requirements represent the original customer needs, while the design parameters represent physical objects or software codes, and the process variables represent physical resources (e.g. human or financial) and software subroutines. Then, *design matrices* describe the relationship between the functional requirements and the design parameters as well as the design parameters and the process variables. Hereby, both linear and non-linear relationships can be expressed. Furthermore, *modules* can be used to capture and reuse the design information related to one particular functional requirement. Finally, the technique states two design axioms (hence the name *axiomatic design*), namely the *independence axiom* and the *information axiom*. The independence axiom states that ideally diagonal or triangular design matrices should be used as known from the design structure matrix approach from the previous section. In contrast, the information axiom states that the design with the least information content should be preferred.

An advantage of axiomatic design is that the customer needs are represented explicitly. Furthermore, the technique enables reuse of design information, which is of special inter-

est for practical applications. Finally, the design axioms can be described formally over the design theoretic concepts. However, the technique does not provide any guidance on the design parameters and process variables to be used. Furthermore, the semantic meaning of the design matrices and, hence, the dependencies among and between the function requirements, design parameters, and process variables remains unclear.

2.1.5 Function-based techniques

While the matrix-based techniques already provide strong methodological focus, they lack important semantical aspects such as the meaning of customer (respectively functional) requirements, design parameters, and the relations between them. To overcome this problem, function-based techniques provide more detailed models of the desired system functions and their (physical) implementations, while still maintaining focus on methodology. Furthermore, the techniques support advanced analyses such as the realizability of a function given the design parameters, which helps to uncover inconsistencies and design flaws early in the design process. In the following, (1) *design prototypes*, (2) the *function-behavior-state* approach, (3) the *functional representation*, and (4) the *Schemebuilder* are introduced, which are the most prominent representatives of this category.

Design prototypes

One of the first function-based techniques are the so-called *design prototypes* proposed by Gero [Ger90]. The general idea behind the technique is to derive design prototypes from similar *design cases* and to document the design prototypes using a predefined *schema*. Then, given a design problem the designer can search the design prototypes and instantiate them in the novel context. The schema itself separates between the *function* of the design prototype, its (physical) *structure*, as well as its *expected* and *actual behavior* all expressed in terms of *variables*. Furthermore, the schema contains different types of knowledge. The *relational knowledge* defines general relations between the functional, structural, and behavioral variables similar to the matrix-based approaches (see Section 2.1.4). Then, *qualitative knowledge* specifies qualitative relations between the variables such as qualitative proportionality as known from physics-based approaches (see Section 2.1.2). Moreover, *computational knowledge* determines computational relations between the variables, which can be used in simulations. At last, *context knowledge* introduces exogenous variables, which describe the environment of and stimuli to the system. Finally, design prototypes also can be *partitioned* into their constituents and *categorized* within a classification system.

An advantage of the design prototypes is the integration of requirements and implementation details into a common framework. Furthermore, the relationships between the variables can be expressed at various levels of detail (informally, qualitatively, and computationally). However, it remains unclear how the composition of design prototypes actually works. Furthermore, checks such as whether the implementation details are consistent with the requirements are not supported hindering automated quality assurance and verification tasks.

Function-behavior-state

Probably the most well-known function-based technique, which addresses design composition and function realizability, is the *function-behavior-state* approach proposed by Umeda et al. [UTTY90, UIY+96]. The approach distinguishes between *functional knowledge* and *physical features*. The functional knowledge comprises the *functions* themselves, their *decompositions* into networks of sub-functions, and their *embodiments*. Functions, in turn, are described in terms of *verbs*, *objects*, and *modifiers* (e.g. move material fast). In contrast, decompositions are hierarchical relationships between functions. Hereby, the authors distinguish between *task* and *causal decomposition*. In task decomposition the sub-functions are rather independent (e.g. first mill, then grind), while in causal decomposition the sub-functions are causally related (e.g. first generate voltage, then generate light). Finally, embodiments are relations between functions and physical features. Physical features, in turn, are described in terms of *components* (e.g. gears) and *physical phenomena* acting on the components. Hereby, physical phenomena are represented using qualitative process theory, which has been explained previously (see Section 2.1.2). Consequently, the system behavior can be expressed using quantities, qualitative conditions, qualitative relations, and influences.

An advantage of the function-behavior-state approach over, e.g. design prototypes, is that compositional techniques can be used both for function and implementation details. Furthermore, simulation can be used to check whether certain physical phenomena can occur and, hence, whether certain functions are realizable. However, the specification of functions remains informal. Consequently, one cannot determine whether the related physical phenomena actually implement the desired functions. Hence, only limited support is provided for design verification. Furthermore, the approach does not represent geometry and motion explicitly hindering its use by mechanical engineers.

Functional representation

Another function-based technique, which addresses function formalization and design verification, is the *functional representation* proposed by Chandrasekaran et al. [CGI93]. The core element of the technique are *components* (e.g. devices), which provide *ports* for their interaction with the environment. Then, components comprise *subcomponents* and their *relations* (e.g. spatial or functional). Hereby, also relations between the ports of these components are supported. Finally, components define *functions* that can be performed. Functions, in turn, are described in terms of *start conditions* (or preconditions over the input ports), a *function predicate* (or postconditions over the output ports), and a *causal process description*. Finally, causal process descriptions are described in terms of *states* and *transitions*. States are represented as predicates over so-called state variables. Transitions can occur due to the completion of another causal physical process (e.g. a light bulb turning from the inactive to the active state), due to the completion of the function of a subcomponent (e.g. a light bulb producing light), or due to a domain law (e.g. incandescence, i.e. the electromagnetic radiation of hot bodies) that is not specified further.

An advantage of the functional representation is that design verification can be performed by means of simulation. Consequently, one can determine whether a component is able to perform its provided functions by means of the causal processes. Furthermore, the representation can be used to generate diagnostic knowledge from the causal processes including their states and transitions. On the downside, the function formalization and, hence, the design verification is limited to basic input-output patterns. Furthermore, the technique does not represent the shape and motion of mechanical components explicitly. Consequently, the necessary design information cannot be captured, verified, and validated holistically.

Schemebuilder

Then, a function-based technique, which comes with commercial tool integration, is the *Schemebuilder* approach proposed by Bracewell and Sharpe [BS96]. The core of the approach is a *function-means tree*. The function-means tree decomposes the function of the entire system into *sub-functions* and eventually into *means*. Functions themselves are represented in terms of their *type* and their *interface*. Function types can be arbitrary human-readable character strings (e.g. generate voltage). Function interfaces, on the other hand, are defined in terms of input and output *ports* (e.g. input control signal and output voltage). Hereby, different types of ports can be distinguished, namely *material*, *power*, and *information*, while power is subdivided further into mechanical

(i.e. rotational and translational), fluid, and electrical. Finally, means are described in terms of the supported *function types* (i.e. human-readable character strings), their interface (i.e. input and output ports), and a *MATLAB Simulink model* of the behavior explained later (see Section 2.1.7). Note that the matching between functions and means is achieved through the function types.

An advantage of the Schemebuilder approach is that system designs can be composed quickly from existing means. In particular, available means can be discovered based on the required function type and interface. Hereby, different alternatives can be retrieved, e.g., for generating voltage. Furthermore, the approach is integrated with MATLAB Simulink, which allows to model accurate physical behavior and which is used widely in industry. However, a disadvantage of the approach is that the required function can be specified only in terms of the ports, while desired input-output relations (also known as the *semantic interface* [BS01]) cannot be described formally. Consequently, one cannot determine whether a function actually can be realized given the means.

2.1.6 Ontology-based techniques

Function-based techniques already provide a strong methodological focus, a high level of abstraction, and various simulation or formal verification capabilities to uncover certain design flaws. However, still certain design knowledge cannot be captured and certain relations among elements of the design knowledge cannot be expressed such as part geometries, motion profiles, and collision-based interaction. To overcome this problem, ontology-based techniques seek to structure design knowledge in terms of its concepts and their relations. Note that the ontology-based movement was boosted particularly by the development of the Web Ontology Language (OWL) [HPSvH03], which laid the technical foundation for representing and reasoning about design knowledge in computer programs. In the following (1) a *requirement ontology*, (2) the *PhysSys ontology*, (3) the *functional basis* approach, (4) a *functional concept ontology*, and (5) a *port ontology* are discussed each focusing a different and partially complementary aspect of the design knowledge available at different phases of the design process.

Requirement ontology

One of the first ontological approaches taking customer requirements and their relation to implementation details into account is the *requirement ontology* proposed by Lin et al. [LFB96]. The requirement ontology is based on a *product ontology*, which comprises *parts*, *features*, and *parameters*. Hereby, both parts and features are organized in containment hierarchies such that one can distinguish primitive and composite ele-

ments. Furthermore, features such as nubs can associated with parts such as bricks. Finally, parameters are used to describe certain properties of parts and features such as the diameter of a nub or the size of a brick. Then, the requirement ontology comprises a representation of the *requirements* themselves. Requirements essentially define logical expressions over the parts, features, and parameters such as the maximum diameter of the nub (i.e. the so-called implementation details). Hereby, similar to parts and features requirements are organized in a containment (or decomposition) hierarchy. Consequently, the authors formalize certain properties such as faithful decomposition (i.e. cases where the decomposition logically entails the original requirement), derived or explicit requirement, external or internal requirement, and physical, structural, performance, functional, or cost requirement.

An advantage of the requirement ontology is that both requirements and implementation details as well as their relations can be captured in a rather formal manner. Consequently, it is possible to check the consistency among requirements (indicating specification flaws) as well as between requirements and implementation details (indicating implementation flaws). However, the design knowledge is limited to basic structural and parametric characteristics. In particular, the function and behavior of the designed system or product is neglected, which plays an important role as described in cyber-physical and, hence, software-intensive manufacturing systems.

PhysSys ontology

Then, Borst et al. [BAT97] proposed the *PhysSys ontology*, which concentrates on the logical structure and physical behavior of systems. The PhysSys ontology is built in a modular fashion and comprises a *component ontology* for describing the system structure, a *process ontology* for describing the system behavior from a high-level, structural perspective, and a *mathematical ontology* for describing the system behavior from a low-level, computational perspective. Hereby, the component ontology provides *mereological* and *topological* information about *components*. The mereological information defines a containment hierarchy over the components. In contrast, the topological information adds connections between components for energy exchange. Then, the process ontology comprises *physical domains* such as electric, magnetic, or thermal, which are defined in terms of their *effort* and *flow* quantities such as charge and voltage. Furthermore, the process ontology comprises typical *mechanisms* that can be found in every physical domain such as stores (e.g. electrical capacitors) and dissipators (e.g. electrical resistors). Finally, the process ontology allows one to connect mechanisms with so-called *bonds*, which represent the energy exchange in the component ontology. Note that the process ontology is based on the bond graph theory [Pay60]. At last, the mathematical ontol-

ogy maps the mechanisms and bonds to mathematical equations describing the system dynamics, which can be used by simulators.

An advantage of the PhysSys ontology is that physical systems can be described from various viewpoints (i.e. structural, behavioral, and / or mathematical). Furthermore, the ontology provides a unified abstraction of physical mechanisms across physical domains. Consequently, complex physical systems can be described by engineers from different domains (e.g. thermal and electric) using a shared vocabulary (e.g. stores and dissipators). However, a disadvantage of the technique is that design knowledge about software components cannot be captured. Also, customer requirements and functional design details are not considered. Consequently, consistency checks and verification capabilities are limited to implementation details only.

Functional basis

Then, a popular ontology-based approach for structuring functional design knowledge is the *functional basis* proposed by Hirtz et al. [HSM+02]. The general idea behind the approach is to provide a basic vocabulary for describing the functions of technical systems. In particular, the functional basis approach distinguishes between *function* and *flow*, while function is described in terms of a *verb* (or action) acting upon an *object*, i.e. the flow. Note that this basic representation of function is taken from Pahl and Beitz [PBFG06]. Then, the authors develop a *three-level taxonomy* for both function and flow. Hereby, nine primary classes of function and three primary classes of flow can be distinguished. The functions comprise branching, channeling, connecting, controlling some magnitude, converting, provisioning, signaling, and supporting. In contrast, the flows include material (e.g. solid or liquid material), signal (e.g. status or control signal), and energy (e.g. acoustic or electric energy). The secondary and tertiary classes define more specific functions and flows in each category. Consequently, one can describe the function of a technical system using a combination of function and flow terms from any level of the taxonomy. For example, some function might be the *branching of material* (more abstract) or the *removal of solid material* (more concrete).

An advantage of the functional basis approach is that a rich set of standard terms is provided for describing the functions of technical systems in a unique and repeatable manner. Furthermore, the approach allows one to define and explore functional design alternatives quickly enabling practical design optimization. However, the approach does not cover customer requirements, structure, behavior, and other implementation details. Consequently, the relations between the functions and other design knowledge cannot be expressed and examined. Furthermore, the functional basis lacks a semantic foundation leading to potential ambiguities.

Functional concept ontology

Another ontology-based approach concentrating on device functionality is the *functional concept ontology* proposed by Kitamura et al. [KKFM04, KM04]. The functional concept ontology comprises a *device ontology* and a *functional ontology*. The device ontology allows one to describe physical systems in terms of *devices* with input and output ports as well as *operands*, which represent the input and output flows such as substances, energy, or information. Then, the device *behavior* is defined in terms of the change of operand state between input and output ports. In contrast, the functional ontology comprises a *meta-function layer* and a *base-function layer*. The meta-function layer describes the types of base functions and their relationships. Typical meta-functions are *maintaining* or *making*, which are similar to the functional packages of Keuneke [Keu91]. Instead, the base functions are concerned with the change of operand states such as generating electricity. Furthermore, base functions can be decomposed into sub-functions such as vaporizing fuel and steaming water, which represents the *method of function achievement*. Additionally, any structural or physical justification for the decomposition can be provided such as heat transfer, which is called the *way of function achievement*.

An advantage of the functional concept ontology is that it provides a clear separation between the desired function and the way of function achievement. In particular, different ways of function achievement can be explored in parallel during design optimization. Furthermore, the ontology establishes a relationship between the implementation details (i.e. the device ontology) and the functional design (i.e. the functional ontology). However, the approach lacks formal semantics leading to potential ambiguities and misinterpretation. Furthermore, customer requirements are not considered directly.

Port ontology

Finally, another interesting ontology-based approach concentrating on the connections between different components of a technical system is the *port ontology* porposed by Liang and Paredis [LP04]. The port ontology comprises an *attribute layer* and a *port ontology layer*. The attribute layer distinguishes between *form*, *function*, and *behavior attributes* of ports. Form attributes are, for example, the location and shape of ports, while the shape is described using classical geometry representations such as constructive solid geometry [RV77]. In contrast, function attributes are described in terms of the functional basis approach introduced previously [HSM+02], i.e. standard *function* and *flow* term pairs such as to convert (= function) electrical energy (= flow). Finally, behavior attributes associate each port to variables in some behavioral model. As an example, the authors provide a mapping between the port ontology and Modelica connectors, which

have been explained before (see Section 2.1.3). Furthermore, the approach supports the specification of interaction models, which can be placed at the interface between two components during simulation.

An advantage of the port ontology is that a wide variety of design knowledge about ports can be captured, which is not represented holistically in other approaches. Furthermore, axioms can be defined to constrain possible instantiations of ports and connections between them. Another strong feature are the interaction models, which allow one to simulate different aspects of interaction. However, a disadvantage is that both behavior and function attributes are described only informally. Furthermore, customer requirements and implementation details as well as their relation to the port ontology are not discussed at all.

2.1.7 Commercial tools

Finally, some important commercial tools are mentioned, which are used for the (conceptual) design of cyber-physical (manufacturing) systems. Note that the commercial tools mostly pick up and integrate the ideas and concepts introduced in the previous sections such as component-based (see Section 2.1.3) and function-based (see Section 2.1.5) modeling. Consequently, the previous techniques can be considered as the "theoretical" foundations for the commercial tools, while the commercial tools provide practical implementations of the ideas and concepts with focus on quality aspects such as performance, reliability, and usability. In particular, the discussion focuses on the tool suites offered by (1) *Siemens Product Lifecycle Management Software*, (2) *Dassault Systèmes*, (3) *The MathWorks*, and (3) *No Magic*, which all are leading tool vendors in their respective field of expertise. Note that other tool vendors with similar tool suites exist, which are neglected in the presentation.

Siemens Product Lifecycle Management (PLM) Software

One of the oldest tool vendors for system or product design is the *Siemens Product Lifecycle Management* (PLM) *Software*[1] with headquarters in Plano, Texas, United States. Siemens PLM Software was formerly known as the *UGS Corporation*, which was founded in 1963. Today, the Siemens PLM Software has more than 7,000 employees. The Siemens PLM Software tool suite comprises *Teamcenter*, which represents a product lifecycle management solution for the Siemens PLM Software tools. Consequently, Teamcenter provides the means for capturing all information related to a product or system during its lifetime. In particular, the Teamcenter user interface allows one to capture customer

[1] https://www.plm.automation.siemens.com/

requirements in text documents as well as the functional design in diagram drawings, while more specialized tools are provided for other product information. For example, Siemens PLM Software provides the *NX* tool, which can be used for assembly structure and part geometry design. NX also supports static and dynamic analysis techniques such as the finite element method [Clo60] and the finite volume method [JST81]. Furthermore, Siemens PLM Software supplies the *Mechatronics Concept Designer*, which integrates NX-based structure and geometry as well as trigger- respectively sequence-based behavior modeling. Hereby, the behavioral model integrates multi-body system dynamics, which has been introduced previously (see Section 2.1.2). Consequently, realistic kinematic and control behavior can be designed and evaluated. Other Siemens PLM Software tools include *Knowledge Fusion* for capturing design knowledge and automating design processes, *Tecnomatix* for manufacturing process planning and discrete event based process simulation, and *PCBexchange* for integrating NX-based assembly structures and part geometries with electrical design information.

An advantage of the Siemens PLM Software tool suite is that a wide range of product information can be captured and integrated using the Teamcenter platform. Furthermore, specialized tools exist for various design tasks from capturing customer requirements to defining function hierarchies, assembly structures, part geometries, control behaviors, and production processes. However, the design information is missing a seamless semantic foundation such that inconsistencies might occur, which remain undiscovered. For example, one has to decide manually whether the implementation details (e.g. part geometries as well as physical and control behaviors) actually implement the production processes and, hence, satisfy the customer requirements.

Daussault Systèmes

One of the largest tool vendors for system or product design is *Dassault Systèmes*[2] with headquarters in Vélizy-Villacoublay, France. Daussault Systèmes was founded 1981, 18 years after the Siemens PLM Software. Today, Dassault Systèmes has more than 13,000 employees. The Daussault Systèmes tool suite comprises *Enovia*, which provides a product lifecycle management solution similar to Teamcenter from Siemens PLM Software. In particular, Enovia provides features for capturing customer requirements in text documents, developing product configurations based on features and options, managing part revisions including their geometry, mechanical connectivity, and function, as well as performing annotation-based design reviews. Then, Dassault Systèmes provides *CATIA* for assembly structure and part geometry design similar to NX from Siemens PLM

[2] http://www.3ds.com/

Software. In particular, CATIA integrates electrical design capabilities from schematic diagrams over geometry design to cable harness. Furthermore, CATIA provides means for specifying system architectures in terms of components and their interactions based on Modelica, which has been explained before (see Section 2.1.3). Finally, Dassault Systèmes supplies *Simulia*, which comprises and integrates a variety of simulation tools based on the functional mockup interface (FMI) standard [BOA+11]. In particular, Simulia comprises multi-body system dynamics (see Section 2.1.2), the finite element method [Clo60], and the finite volume method [JST81] similar to NX from Siemens PLM Software. Other tools from Dassault Systèmes include *Delmia* for manufacturing process planning based on products, resources, and their relationships, manufacturing process simulation for virtual NC program testing, as well as assembly process simulation based on collision-free product disassembly trajectories, or *Geovia* for modeling and simulating, e.g., mining activities.

An advantage of the Dassault Systèmes tool suite over the Siemens PLM Software tool suite is that cyber-physical system architectures are integrated more thoroughly into the data model based on the Modelica language. Consequently, components can be defined which comprise both mechanical and electrical interfaces respectively behavior. Still, the tool suite lacks a seamless formal foundation such that certain inconsistencies and design flaws can be detected only manually. In particular, the requirements and manufacturing processes as well as their relation to implementation details remain informal hindering, for example, automatic design verification.

The MathWorks

Then, another popular tool vendor for controller design is *The MathWorks*[3] with headquarters in Natick, Massachusetts, United States. The MathWorks was founded in 1984, three years after Dassault Systèmes. Today, The MathWorks has more than 3,000 employees. The MathWorks tool suite comprises *MATLAB*, which provides a programming language specifically designed for mathematical computations. In particular, the language comes with native support for linear algebra operations over vectors and matrices. Then, based on the MATLAB core The MathWorks ships *Simulink*, which allows one to compose simulation models for complex cyber-physical systems from predefined and / or user-defined *blocks*. Blocks may provide input and output *signals*, which can be connected via *signal lines*. Furthermore, blocks may provide *parameters* (such as the resistance of an electrical resistor) for configuring the block behavior. Finally, the block behavior can be expressed in the MATLAB programming language or some other

[3] https://www.mathworks.com/

programming language, which is supported by the platform. Also, Simulink provides simulation and debugging capabilities, design verifiers based on test case generators and model checking technology [CGP99], as well as code generators for deploying blocks on different hardware platforms. Then, based on Simulink, The MathWorks ships *Stateflow* for modeling and simulating decision logic based on state machines and flow charts as well as *Simspace* for modeling and simulating physical systems based on differential equations.

An advantage of The MathWorks tool suite over the previous tool suites is that much more powerful support is provided for controller design and verification. In particular, the controller design can be verified automatically with respect to the physical system using test inputs and validation components. However, the tool suite is not suited for designing part geometries or assembly structures. Furthermore, multi-body system dynamics (see Section 2.1.2) is more difficult to describe in Simscape than in traditional tool suites. Also, The MathWorks does not provide support for the finite element method [Clo60] and the finite volume method [JST81].

No Magic

Finally, one popular tool vendor for diagram-based software and system design is *No Magic*[4] with headquarters in Allen, Texas, United States. No Magic was founded in 1995, eleven years after The MathWorks. Today, No Magic has between 200 and 500 employees. The No Magic tool suite comprises *Teamwork*, which provides the means for storing, versioning, and sharing system models as well as for enabling collaboration and communication between the team members. In a sense, Teamwork is similar to product lifecycle management solutions such as Enovia from Dassault Systèmes or TeamCenter from Siemens PLM Software. However, Teamwork only provides limited functionality with respect to manufacturing supplier integration, enterprise resource planning (e.g. bill of materials), and workflow management. On top of Teamwork, No Magic supplies *MagicDraw*, which provides the core graphical modeling and system design features based on the UML (see Section 2.1.1). Besides the usual diagramming features, MagicDraw integrates the object constraint language (OCL) [WK99]. The OCL can be used to specify modeling constraints, which can be evaluated during system design to uncover design flaws automatically. Alternatively, traditional programming languages can be used for specifying and evaluating modeling constraints. Finally, based on MagicDraw, No Magic ships *Cameo*, which provides a dedicated solution for conceptual system design based on the SysML (see Section 2.1.1). Besides the classical SysML diagrams, Cameo

[4] https://www.nomagic.com/

supports advanced requirements management based on storyboards and domain-specific templates as well as mockup-based user interface design. Furthermore, Cameo integrates a simulation toolkit based on FUML [MLK12], which allows one to investigate the runtime behavior of the system.

An advantage of the No Magic tool suite over the previous tool suites is that the requirements, system structure, and state-based system behaviors are moved into focus rather than the part geometries, assembly structures, and differential equation-based behaviors. Consequently, one can think more freely about the engineering problem that needs to be solved and the principle solutions that exist. However, on the downside one cannot go beyond this coarse picture of the system under development. In particular, geometrical information might be relevant already during conceptual design (e.g. because they have been prescribed by the customer or collision-free system operation cannot be guaranteed easily).

2.2 Remaining problems

In the following, problems are summarized that remain unsolved by the approaches presented in the previous section. In Table 2.1 an overview of the related work as well as four evaluation criteria is provided, which have been derived from the challenges presented in the first chapter (see Section 1.3). In particular, the evaluation criteria include

- the *coverage* of *design information* (see Section 2.2.1),
- the *integration* of the *underlying formalisms* (see Section 2.2.2),
- the *automation* of *verification tasks* (see Section 2.2.3), and
- the *practicability* of the *design method* (see Section 2.2.4).

The table provides a numeric rating for each related approach and evaluation criterion, which indicates a degree of satisfaction ranging between *low* (= 0.0 points), *medium* (= 0.5 points), and *high* (= 1.0 points). Furthermore, the table provides an aggregate numeric rating over all evaluation criteria. The aggregate rating is calculated from the sum of the ratings for the individual evaluation criteria. Consequently, the aggregate rating indicates how well the individual approaches address all evaluation criteria simultaneously with 4.0 points representing a perfect match. Subsequently, each evaluation criterion is discussed in more detail before summarizing the derived research objectives in the next section.

Related work			Evaluation criteria				
Sec.	Approach		2.2.1	2.2.2	2.2.3	2.2.4	∑
2.1.1	UML	[BJR+96, BME+07]	○	◐	◐	○	1.0
	SysML	[Hau06b]	◐	◐	◐	◐	2.0
	CONSENS	[RBG14]	●	◐	◐	●	3.0
2.1.2	Multi-body system dynamics	[Sch97, Sha97]	◐	●	◐	○	2.0
	Qualitative physics	[For84]	◐	●	◐	○	2.0
	Qualitative kinematics	[Fal90]	◐	●	◐	○	2.0
2.1.3	Modelica	[MEO98]	◐	●	◐	○	2.0
	Mechatronic UML	[BGT05]	◐	●	●	◐	3.0
	STEM	[Hum09]	◐	●	◐	○	2.0
2.1.4	Quality function deployment	[CW02, HC88]	◐	◐	◐	◐	2.0
	Design structure matrix	[Ste81, Bro01]	○	◐	◐	◐	1.5
	Axiomatic design	[Suh95, Suh98, SCL98]	○	◐	◐	◐	1.5
2.1.5	Design prototypes	[Ger90]	◐	○	○	○	0.5
	Function-behavior-state	[UTTY90, UIY+96]	◐	●	●	◐	3.0
	Functional representation	[CGI93]	◐	◐	◐	◐	2.0
	Schemebuilder	[BS96]	◐	◐	◐	◐	2.0
2.1.6	Requirement ontology	[LFB96]	◐	●	◐	◐	2.5
	PhysSys ontology	[BAT97]	◐	◐	◐	○	1.5
	Functional basis	[HSM+02]	◐	○	○	○	0.5
	Functional concept ontology	[KKFM04, KM04]	◐	◐	◐	◐	2.0
	Port ontology	[LP04]	◐	◐	◐	○	1.5
2.1.7	Siemens PLM Software	[1]	●	◐	◐	●	3.0
	Dassault Systèmes	[2]	●	◐	◐	●	3.0
	The MathWorks	[3]	◐	◐	●	◐	2.5
	No Magic	[4]	◐	◐	◐	◐	2.0
		∑	12.5	16.5	13.0	9.0	

Table 2.1: Evaluation of the existing approaches with respect to four key criteria.

2.2.1 Information coverage

From the challenge of *mechanical dominance* (see Section 1.3.1) the need for an improved coverage of design information during the conceptual design phase is derived. Typically, related approaches only consider a small portion of the conceptual design information, while neglecting other aspects. For example, the physics-based approaches (see Section 2.1.2) concentrate on implementation details, while omitting customer requirements and system architectures. Similarly, function-based approaches (see Section 2.1.5) concentrate on customer requirements and function decomposition, while omitting part geometries and spatial collisions as well as their effects on the system design. At least commercial tools from Siemens PLM Software and Dassault Systèmes (see Section 2.1.7) provide high coverage of design information through their product lifecycle management solutions. However, coming from a mechanical engineering background these tools typically lack knowledge about the software design as well as corresponding verification techniques. Also, the diagram-based technique CONSENS (see Section 2.1.1) provides high coverage, but performs less well regarding the other evaluation criteria – amongst other things – because it does not provide a complete integrated formalism underlying the design information.

Conclusion

The next generation of conceptual design tools and techniques for cyber-physical manufacturing systems should provide a similar high coverage of design information as commercial product lifecycle management solutions or CONSENS, while providing integrated formalisms enabling automated evaluation of the design information and fostering practical design methodologies.

2.2.2 Integrated formalism

Then, from the challenge of *lacking synchronization* (see Section 1.3.2) the need for an integrated formalism is derived underpinning the covered design knowledge, which describes the static and dynamic relationships among the individual pieces of the design knowledge. For example, the diagram-based techniques (see Section 2.1.1) only provide a semi-formal graphical syntax for capturing the design knowledge, while formal semantics is missing completely. In contrast, the matrix-based approaches (see Section 2.1.4) provide a formal mathematical syntax for describing some portions of design knowledge, but the approaches still are lacking formal semantics. Then, commercial tools (see Section 2.1.7) provide both formal syntax and semantics, but the capabilities are limited to certain design knowledge only (such as part geometries as well as their motion and

deformation). In particular, the customer requirements and functional designs as well as the relation between customer requirements, the functional design, and implementation details remain informal. Only, physics-based approaches (see Section 2.1.2) and component-based approaches (see Section 2.1.3) as well as the function-behavior-state approach (see Section 2.1.5) and the requirement ontology (see Section 2.1.6) provide completely integrated formalisms both for syntax and semantics. However, these approaches provide only limited coverage of design knowledge.

Conclusion

The next generation of conceptual design tools and techniques for cyber-physical manufacturing systems should provide similar well integrated formalisms as physics-based, component-based, and some function-based approaches, while providing a higher coverage of the necessary design information and fostering practical design methodologies.

2.2.3 Automated evaluation

Furthermore, form the challenge of *insufficient assurance* (see Section 1.3.3) the need for an automated evaluation of the design information with respect to relevant quality criteria such as completeness and consistency is derived. Note that an automated evaluation requires an underlying formalism, such that quality properties can be expressed unambiguously and calculated algorithmically. For example, approaches like the design prototypes (see Section 2.1.5) or the functional basis (see Section 2.1.6) only support very limited evaluation capabilities such as text-based search. Then, the diagram-based techniques (see Section 2.1.1) already provide more advanced evaluation support regarding the completeness and consistency of the design information based on the semi-formal graphical syntax, while missing semantic evaluation capabilities, e.g., with respect to system dynamics and satisfaction of requirements. Thereafter, the physics-based approaches (see Section 2.1.2) in principle support the automated evaluation of the semantics, but lack a representation of requirements against which the implementation details can be checked. Only, the Mechatronic UML (see Section 2.1.3), the Function-behavior-state approach (see Section 2.1.5), and The MathWorks tool suite (see Section 2.1.7) provide comprehensive evaluation capabilities both for syntactic and semantic quality properties. However, all three approaches only capture limited design knowledge such that important quality properties cannot be evaluated automatically. In particular, all three approach miss an explicit representation of part geometries and spatial collision, which might have a considerable effect on the system design.

Conclusion

The next generation of conceptual design tools and techniques for cyber-physical manufacturing systems should provide a similar high degree of automation of verification tasks as the Mechatronic UML, the Function-behavior-state approach, and The MathWorks tool suite, while covering more design information and defining suitable design methodologies.

2.2.4 Practical methodology

Finally, form the challenge of *sequential engineering* (see Section 1.3.4) the need for a practical methodology is derived, which reduces the time until electrical and software engineers start taking design decisions and which promotes early design verification and validation. For example, the physics-based approaches (see Section 2.1.2) do not provide any practical methodology leading systematically from customer requirements to implementation details, but concentrate on the implementation details and their simulation only. Similarly, the ontology-based approaches (see Section 2.1.6) provide only limited insight regarding practical design methodologies. In contrast, the Mechatronic UML (see Section 2.1.3) already provides an interdisciplinary approach to system decomposition and behavioral design. However, design knowledge regarding manufacturing materials, manufacturing processes, and part geometries is omitted. Similarly the matrix-based approaches (see Section 2.1.4) and the function-based approaches (see Section 2.1.5) contain strong methodological ideas, but provide limited information coverage. Only CONSENS (see Section 2.1.1) as well as the Siemens PLM Software and the Dassault Systèmes tool suites (see Section 2.1.7) provide both a high degree of information coverage as well as support for interdisciplinary design methodologies. However, though being interdisciplinary, the methodologies typically are aligned with the V-model [BD93] hindering early verification and validation of partial designs. Furthermore, the approaches lack an integrated formalism as well as automated evaluation capabilities such that changes might not be synchronized properly and design flaws might not be detected until late.

Conclusion

The next generation of conceptual design tools and techniques for cyber-physical manufacturing systems should provide a similar strong support for interdisciplinary design methodologies as the CONSENS approach or the Siemens PLM Software and the Dassault Systèmes tool suites, while supporting more agile design practices based on formal models and automated evaluation capabilities.

2.3 Research objectives

As shown in the previous section none of the related approaches (see Section 2.1) achieves a perfect rating indicating a general research gap. The diagram-based approach CONSENS [RBG14], the component-based approach Mechatronic UML [BGT05], the function-based approach Function-behavior-state [UTTY90, UIY+96], as well as the commercial tools vendors Siemens PLM Software[1] and Dassault Systèmes[2] receive the highest aggregate ratings with 3.0 points. While CONSENS, Siemens PLM Software, and Dassault Systèmes provide highly suitable abstractions and practical methodologies, they lack a high degree of integration of the underlying formalisms and automation of verification tasks. In contrast, the Mechatronic UML and the Function-behavior-state approach provide a high degree of integration of the underlying formalisms and automation of verification tasks, but lack highly suitable abstractions and practical methodologies. Consequently, either the degree of integration of the underlying formalisms and automation of verification tasks has to be improved, or the suitability of the abstractions and the practicability of the methodologies has to be enhanced with respect to the current state of the art.

Conclusion

The overall aim of this doctoral thesis is to propose a novel approach for the conceptual design of cyber-physical manufacturing systems that provides a highly suitable abstraction of the relevant design information and a practical (i.e. interdisciplinary and agile) design methodology, which both are complemented by a high degree of integration of the underlying formalisms and automation of verification tasks.

2.4 Summary and outlook

This chapter summarized related work on conceptual design of cyber-physical manufacturing systems (see Section 2.1). Then, problems were derived that remain unsolved with respect to the challenges from the first chapter (see Section 2.2). Finally, the research objectives of this doctoral thesis were summarized (see Section 2.3). The next chapter describes a novel test-driven method, which is meant to address the sequential engineering challenge.

3 Test-driven method

To tackle the sequential engineering challenge described in the first chapter, a test-driven method for the conceptual design of cyber-physical manufacturing systems is proposed. The basic ideas are taken from test-driven software development [Bec02] and component-based software development [BS01]. The key principle in test-driven development is to write test cases first. Then, the implementation proceeds incrementally over the test cases until the software is finished. In each increment the software structure is refactored potentially to maintain the software quality. Studies exist that confirm the positive impact of test-driven development on software quality [MW03, WMV03, GW03, BN06, NMBW08]. In contrast, the key principle behind component-based software development is to decompose systems into components. Consequently, complex engineering tasks can be divided into less complex engineering tasks. Now, Figure 3.1 provides an overview of the proposed test-driven design method for cyber-physical manufacturing systems.

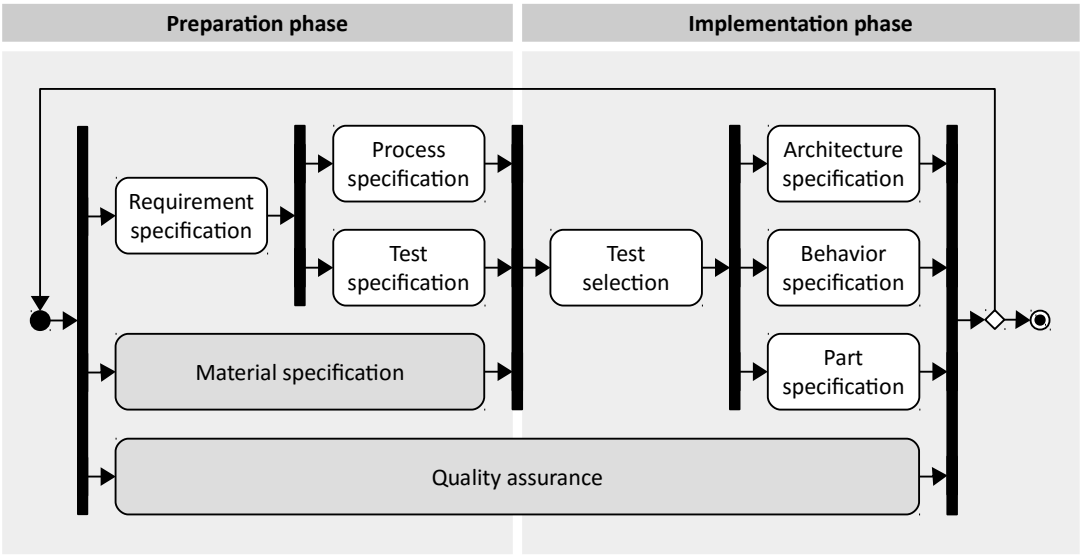


Figure 3.1: Proposed method for the conceptual design of manufacturing systems.

The proposed method separates between a *preparation phase* and an *implementation phase*. In the preparation phase, the requirements for the manufacturing system are specified and (ingoing as well as) outgoing materials are defined. Afterwards, manufacturing processes and test cases are derived including intermediate material states. Typically, the manufacturing processes are designed to ensure an efficient transformation from ingoing to outgoing materials through suitable intermediate material and system states. In contrast, the test cases are intended to verify the upcoming implementation with respect to the requirement and the process specifications. Then, in the implementation phase a subset of unimplemented test cases is selected and the implementation is revised/extended until these test cases are passed. In particular, in this step the component architecture, the component behavior and the component parts are modified. Afterwards, one needs to decide whether to accept the system specification (including requirement, process, test, architecture, behavior and part specification) or to revise the specification in a subsequent iteration. Finally, syntactic quality assurance is carried out continuously throughout the conceptual design process, while semantic quality assurance is executed periodically at the end of each iteration and/or increment of the design process. Note that the same procedure is applied also for each component of the internal component architecture. Consequently, in each iteration each subcomponent has to pass the implementation phase before the system can pass the implementation phase.

In the following, the preparation phase is explained in Section 3.1 and the implementation phase in Section 3.2. For each phase the associated activities as well as respective quality assurance measures are described. Furthermore, for each activity its purpose, the relevant background, as well as the specific approach are described.

3.1 Preparation phase

The *preparation phase* is concerned with understanding the customer needs, defining appropriate manufacturing processes, and planning effective test procedures. The customer needs concern the in- and output materials processed by the system as well as the desired energy and data flows. In contrast, the manufacturing processes mostly comprise the admissible sequences of manufacturing operations. Finally, the test procedures embrace the expected reaction of the system under different operative circumstances. Subsequently, the *requirement specification* activity is explained in more detail in Section 3.1.1, the *process specification* activity in Section 3.1.2, and the *test specification* activity in Section 3.1.3. Note that the preparation phase also includes a *material specification* and a *quality assurance* activity (see Figure 3.1). However, these two activities are not described separately, but they are referred to while describing the other activities.

3.1.1 Requirement specification

The *requirement specification* activity is concerned with understanding and documenting the operational context and the purpose of as well as constraints for the manufacturing system from the viewpoint of the different stakeholders [KS98]. Typical stakeholders are the purchaser and the provider of the manufacturing system as well as regulatory institutions. Furthermore, typical sources of requirements are public standards, house regulations of the purchaser and the provider, and country regulations of the purchaser site and the provider site. In the following, first background information on requirement specification is provided before sketching the proposed approach.

Background

The main purpose of the manufacturing system is to produce output materials from potentially predefined input materials. For example, the purpose of a gear manufacturing system is illustrated in Figure 3.2. Both input and output materials can be characterized by a number of different material properties, while each material property can have an impact on the manufacturing process design and the manufacturing system implementation. In particular, in most cases the material geometry needs to be considered to prevent undesired collisions during manufacturing system operation [Hum11]. Moreover, in the case of gear manufacturing the tooth strength needs to be considered to prevent gear failure during usage [KS03]. The material properties considered here are determined usually by the purchaser of the manufacturing system and the operational context of the output material respectively.

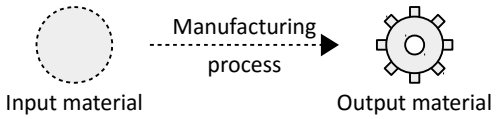


Figure 3.2: Example material specification during requirement specification.

Additional requirements documented in literature concern the performance of the manufacturing process [VN92] and the sustainability of the manufacturing process [GDT06]. The performance can be measured, for example, in terms of turnover, while the sustainability can be measured in terms of waste or emissions. Then, the operational context of the manufacturing system might include human operators and technical systems. Both for human operators and technical systems appropriate interfaces have to be provided that enable desired and disable undesired interactions. For example, human operators might need to adjust parameters during manufacturing system operation, while technical

systems such as the power grid might supply electric energy or a material flow system might supply input materials and fetch output materials.

Approach

The proposed approach envisages first to collect requirements from possible sources (such as standards and regulations) as well as in meetings with stakeholders. Initially, the requirements are documented in natural language and freehand sketches. However, one key problem of natural language and freehand sketches is their inherent ambiguity [Ber08, OSSJ09]. To overcome this problem, in accordance to other work [FKV91] translating the original requirements to more formal representations is proposed. Note that techniques exist that try automating the translation both for natural language [LB02, IO05] and for freehand drawings [SG00, DHT00]. Besides ambiguity there exist numerous other qualities of requirement specifications such as completeness, correctness, as well as internal and external consistency [DOJ+93]. Different approaches have been proposed to address each of these qualities. For example, the completeness of a requirement specification can be assessed by means of reviews and prototypes [And89, Dav92]. In contrast, the internal consistency can be assessed using formal languages and analysis techniques [Alf85, DMRT79]. However, this thesis concentrates on manual quality assurance techniques for simplicity such as reviews. Furthermore, the quality of the requirement specification is expected to improve across iterations and increments.

3.1.2 Process specification

After completing the requirement specification activity (see Section 3.1.1), the *process specification* activity is concerned with designing a suitable manufacturing process. In particular, a suitable manufacturing process must produce the required output material within the given constraints. As mentioned previously, constraints might be defined regarding the process performance or the process sustainability. In the following, again background information on process specification is provided before sketching the proposed approach.

Background

In general, one can distinguish between macro-level process specifications and micro-level process specifications depending on the level of abstraction [EN16]. Macro-level process specifications select appropriate manufacturing operations (e.g. milling or sanding) and

define their sequential order [ElM93]. For example, a possible macro-level process specification for gear manufacturing is illustrated in Figure 3.3. The depicted manufacturing process uses three manufacturing operations for transforming the input material to the desired output material. First, *milling* is used to transform the input material to the first intermediate (or low-precision) material state. Then, *sanding* is used to transform the first intermediate material state to the second intermediate (or high-precision) material state. Finally, *drilling* is used to transform the second intermediate material state to the output (or finished) material. Note that milling is quick and inaccurate, while sanding is slow and accurate. Consequently, both the process speed and the process accuracy benefit from using these manufacturing operations in combination.

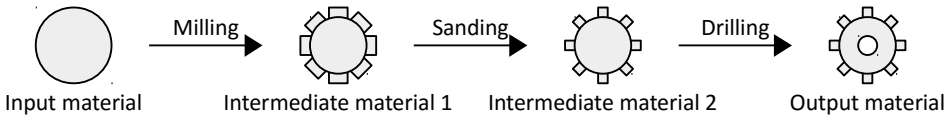


Figure 3.3: Example material specification during process specification.

In contrast, micro-level process specifications define the actual parameters of the manufacturing operations selected in a macro-level process specification. Important parameters are the duration of each manufacturing operation as well as respective tools (e.g. drilling heads or cutting dies) and fixtures (e.g. clamps) [ElM93]. Hereby, in some cases one might be able to reuse existing tools and fixtures developed in previous projects. In many cases, however, new fixtures have to be developed to implement certain manufacturing operations. Typically, the fixtures have to be designed to avoid certain collisions during system operation and to withstand certain forces. Finally, additional parameters under consideration are datum surfaces, production tolerances, and cutting conditions [AZ89]. For example, the cutting conditions include the cutting depth and the cutting speed.

Approach

In accordance to previous work, the proposed approach envisages first to plan the manufacturing process on a macro-level before determining the parameters of each manufacturing operation on a micro-level. Note that computer-aided process planning and computer-aided manufacturing tools exist that try automating this task [AZ89, CWW05]. In particular, planning can be formulated also as an optimization problem over possible manufacturing operations, their parameters, and selected optimization criteria [JSJ98, CT98]. These tools can be used with the proposed method in principle as

well. However, the remainder of this thesis concentrates on manual planning and quality assurance for simplicity, which requires human expertise. Furthermore, the quality (e.g. the optimality) of the process specification is expected to improve across iterations and increments.

3.1.3 Test specification

After completing the requirement specification activity (see Section 3.1.1) and in parallel to the process specification activity (see Section 3.1.2), the *test specification* activity is concerned with deriving suitable test cases and test suites [Gon70, Cho78, FvBK+91]. Hereby, a suitable test suite is able to detect all possible faults in the system implementation. Faults are considered to be unacceptable behaviors of the system implementation as perceived by, e.g., the customer. Note that in the context of the proposed method the requirement specification and the process specification define the set of acceptable behaviors. In the following, background information on test specification is provided before sketching the proposed approach.

Background

Most fundamentally, test cases can be described as relation between selected system inputs and expected system outputs [Tre99, DJK+99]. For example, in the case of the gear manufacturing system described in the previous sections a selected input could be the raw material, while the expected output would be the final product (see Figure 3.4). Note that due to production tolerances the same raw material might lead to different final products respectively product geometries. Then, during test execution the selected input is provided to the system implementation and the actual output of the system implementation is compared to the expected output. If the actual output matches the expected output the system implementation passes the test case. Otherwise, the system implementation fails the test case. Such failures indicate quality defects in the overall system specification, which need to be resolved eventually. Note that both the system implementation and the test specification could be defective in principle. Consequently, one has to evaluate carefully which parts of the specification need to be revised.

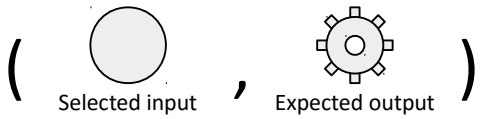


Figure 3.4: Example test specification.

In addition to the previous input-output relationships one might also include timing information into the test specification [KT04]. For example, in the previous case of the gear manufacturing system one can specify that the final product must be provided within a predefined duration (typically in the order of seconds today). Furthermore, one can express that results must not be provided before a second duration has passed. Then, during test execution not only the inputs and outputs have to be considered, but also the duration between events has to be tracked. Consequently, one can compare also the actual duration with the expected minimum and maximum durations. If the actual duration lies in between the expected range the system implementation passes the test case. Otherwise, the system implementation fails the test case. Again, a failure indicates a quality defect which needs to be resolved.

Approach

The proposed approach envisages to derive the test cases from the requirement specification and the process specification respectively. Note that numerous approaches exist that try automating this task. For example, [Gon70, Cho78, FvBK+91] generate test suites from finite state machine specifications of the system under development. More recently, also other specification techniques are supported under the umbrella of model-based testing [DJK+99]. One key quality criterion during test specification is the adequacy of the resulting test suite. The adequacy can be understood as the ability to detect faults in the system implementation [ZHM97]. Different measures have been proposed to describe the adequacy of a test suite. For example, mutation-based measures randomly implant defects into the system implementation and measure the ability of the test suite to uncover them. Note that all previous techniques can be used with the proposed method in principle. The remainder of this thesis concentrates on manual test specification and quality assurance across iterations and increments for simplicity.

3.2 Implementation phase

After finishing the preparation phase, the *implementation phase* is concerned with defining an appropriate technical solution for the original design problem comprising cyber-physical as well as purely mechanical, electrical, and software elements. The implementation phase starts with selecting a subset of test cases (or system functionalities) from the system's test suite. Only the subset of test cases is considered in the following, hence, effectively reducing the scope and complexity of the design problem. Subsequently, the system implementation (i.e. its architecture, behavior, and parts) is revised and extended

with the goal of passing the selected test cases. After revising and extending the system implementation, the test cases are executed to check the overall specification for potential flaws. Hereby, the execution of the individual test cases might *pass* or *fail*.

Passed test cases indicate that the requirement specification, the process specification, the test specification, and the system implementation are *consistent*. A consistent specification suggests that the engineers involved in the design process share a common understanding and that the system was specified and implemented correctly according to this common understanding. However, note that the common understanding of the engineers involved in the design process still might deviate from the expectations of the customers. Therefore, at this point the consistent specification can be presented to the customers to check whether the system specification meets the customer expectations. If the specification meets the expectations the specification is called *valid*. Otherwise the specification is called *invalid*. Furthermore, if all test cases of the system's test suite have been implemented, the specification is called *complete*. Otherwise the specification is called *incomplete*. In case of a valid and complete specification the design process can be terminated. In contrast, in case of a valid and incomplete specification a subsequent walk through the design process is triggered, within which unimplemented test cases are added to the subset of selected test cases. This subsequent walk through the design process is called an *increment* because the scope and complexity of the design problem is increased gradually. Finally, an invalid specification indicates that there might be a misunderstanding between the customers and the engineers. Consequently, the cause of the problem has to be identified before triggering a subsequent walk through the design process and revising the specification accordingly. This subsequent walk through the design process is called an *iteration* because the scope of the design problem remains untouched. Multiple iterations might follow until a valid specification is reached, which can be either complete or incomplete. In case of an incomplete specification the iteration is followed by an increment. Otherwise, the design process can be terminated.

In contrast, failed test cases indicate that the requirement specification, the process specification, the test specification, and the system implementation are *inconsistent*. An inconsistent specification suggests a that there might be a misunderstanding among the engineers involved in the design process and/or that the overall specification might include a bug. Consequently, the engineers involved in the design process have to analyze the cause of the problem. If the cause of the problem is a misunderstanding, the engineers have to discuss their viewpoints and agree upon a common understanding before triggering another walk through the design process and revising their specifications accordingly. Note that sometimes it might be necessary to include the customer into the discussion to resolve the misunderstanding. If the cause of the problem is a bug instead, another walk through the design process can be triggered directly to resolve the specifi-

cation flaw. Note that in both cases changes might be required in any of the specification parts, i.e. the requirement specification, the process specification, the test specification, or the system implementation, which is why the entire design process is triggered from the beginning. The subsequent walk through the design process is called an *iteration* because the scope of the design problem (i.e. the subset of selected test cases) remains untouched. Again, multiple iterations might follow each other until a consistent specification is reached, which can be either valid or invalid and complete or incomplete according to the previous definitions. In case of a valid and complete specification the design process can be terminated directly. In contrast, in case of a valid and incomplete specification an increment follows increasing the scope of the design problem. Finally, in case of an invalid specification an iteration follows including customer feedback.

Subsequently, the execution of test cases is called *verification* and the discussion with customers *validation*. Test-based verification and discussion-based validation, in turn, represent the two final steps of the *quality assurance* activity, which is carried out throughout the entire design process and, hence, the implementation phase (see Figure 3.1). Note that the time to verification and validation is reduced by focusing on a subset of unimplemented test cases (or system functionalities) in each increment and iteration only. Consequently, integration of mechanical, electrical, and software concepts as well as customer feedback can be achieved much earlier than in traditional design processes. As a result, flaws in interdisciplinary designs and misunderstandings in project-related communication can be uncovered and resolved much quicker than before. However, note that focusing on a gradually increasing subset of test cases during the implementation phase also bares risks. In the worst case, the system implementation has to be revised completely after extending the scope of the design problem, hence, causing additional efforts and costs. Consequently, appropriate test selection and implementation strategies are required. Subsequently, the *test selection* activity is described in Section 3.2.1, the *architecture specification* activity in Section 3.2.2, the *behavior specification* activity in Section 3.2.3, and the *part specification* activity in Section 3.2.4. Similar to the previous section, the *quality assurance* activity is not described separately, but it is referred to while describing the other activities.

3.2.1 Test selection

The *test selection* activity is concerned with determining the subset of test cases that is considered in the following implementation activities, namely architecture specification (see Section 3.2.2), behavior specification (see Section 3.2.3), and part specification (see Section 3.2.4). Note that the goal of the subsequent activities is to develop the system design such that it is able to pass the test cases and, hence, to provide the respective

functionalities desired by the customer. As mentioned previously, the test cases have to be selected carefully to avoid negative impacts on the overall design efforts and costs. For example, architectural design decisions taken in early increments might have to be revised in later increments to keep the system design maintainable. However, if the selection order of the test cases would have been reversed, it might have been possible to avoid this problem. Consequently, effective test selection strategies are required, which help identify the most appropriate test cases in each iteration or increment.

Background

Figure 3.5 illustrates an example test selection sequence covering two increments and n iterations. For each increment and iteration the subset of test cases is depicted that has been selected by the engineers involved in the design process. Furthermore, the test selection activities represent the transitions between the individual increments, iterations, and subsets of test cases respectively. Note that when transitioning to the next *iteration*, the subset of test cases remains identical. In contrast, when switching to the next *increment*, the subset of test cases changes. In particular, test cases, which have not been considered previously, are added to the subset based on a test selection *strategy*. Consequently, in subsequent walks through the design process additional test cases are executed and, hence, the system implementation is verified and validated with respect to additional system functionalities. Note that the design process ensures that the system implementation remains correct with respect to system functionalities implemented in previous increments and iterations. Furthermore, note that the test cases might be revised across iterations due to flaws in the test specification. For example, in Figure 3.5 the first test case is revised in iteration 1 and in iteration n.

Figure 3.5: Example test selection sequence covering two increments and n iterations.

Numerous test selection *strategies* are perceivable. For example, one could prefer test cases that are easy to implement in earlier phases of the design process to reduce the time until a working prototype is available. Alternatively, one could concentrate on test cases (or system functionalities), which have been implemented in other projects before to leverage available experience and reuse existing quality-controlled design decisions. Moreover, one could focus on test cases, which have been communicated ambiguously

during project meetings to tackle potential misunderstandings already early in the design process. In general, test selection strategies determine the order, in which the test cases are implemented. This ordering can be achieved, for example, by means of an objective function assigning to each test case a utility value. Then, the test cases with highest utility are selected first and test cases with lower utility are added later. The utility function itself can take various information into account. Most importantly, information about the test cases (e.g. complexity of the individual system functionalities), the project team (e.g. experience of the individual team members), and the customer (e.g. number of projects completed with the same customer) should be included. Note that – assuming appropriate datasets – some information can be computed automatically (e.g. the number of projects completed with the same customer), while other factors have to be determined manually (e.g. the experience of the individual team members).

Approach

The proposed approach envisages to use a test selection strategy that minimizes the time to verification and validation. Consequently, the test cases are prioritized according to the complexity of the underlying system functionalities. Therefore, the approach relies on a method called *planning poker* [Hau06a] for determining an objective measure of complexity for each test case. Hereby, the project manager first explains the test cases that need to be implemented. Then, the engineers can ask question about the test cases to gain a better understanding of the required system functionality. In the next step, each engineer makes an independent estimate of the complexity, e.g., in terms of man hours. Thereafter, the engineers with the lowest and highest estimates explain their viewpoints. Finally, the process is repeated until a consensus is reached. Note that the planning poker method is used commonly in agile development processes such as Scrum [SB02]. Furthermore, note that future research is required to gain more detailed insight into the effect of test selection strategies on the design process and the system quality respectively.

3.2.2 Architecture specification

After the test selection activity (see Section 3.2.1), the *architecture specification* activity is concerned with decomposing the system under development into a suitable combination of less complex elements and their interactions [PW92]. Note that a combination of elements and interactions is considered to be suitable if its behavior conforms to the requirements specification (see Section 3.1.1) and the process specification (see Section 3.1.2) respectively. In this context, the question arises what characterizes the

elements and interactions of an architecture. Several discipline-specific and interdisciplinary interpretations exist. In the following, background information on architecture specification is provided before sketching the proposed approach.

Background

Figure 3.6 illustrates discipline-specific interpretations of the term "architecture". The mechanical architecture can be described by geometric elements [Ulr94]. Then, interactions between elements appear (e.g. upon collision) through exchange of (potential, kinetic, thermal, etc.) energy. Note that collision-based interactions can be forced statically by mechanical connectors (e.g. clamps). Such static interactions can be represented by so-called constraints or joints. In contrast, the electrical architecture can be described by both geometrical and logical elements such as current sources, resistors, or inductors, which provide so-called pins at their interface [FB02]. Then, interactions between elements appear through exchange of electrical energy over connected pins. Furthermore, effects like electromagnetism and thermoelectrics might cause desired/undesired electrical and mechanical interactions. Finally, the software architecture can be described by functional elements, which provide input and output ports with respective data types at their interface [RJB10]. Then, interactions between elements appear through exchange of information over connected ports.

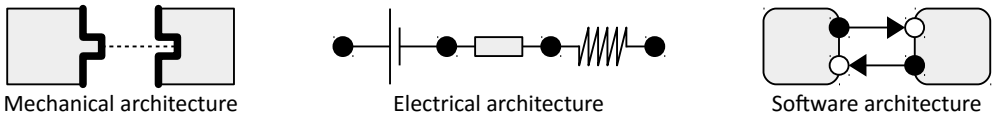


Figure 3.6: Example discipline-specific architecture specifications.

An interdisciplinary interpretation of the term "architecture" requires its elements to combine mechanic, electric, and software characteristics [ME98]. Consequently, the elements of an interdisciplinary architecture might interact via both geometrical and logical interfaces through the exchange of material, energy and information. These characteristics are reflected by the concept of the mechatronic component [Thr05]. In particular, a mechatronic component comprises mechanical ports, electrical ports, and software ports. Mechanical ports represent both mounting ports and material or energy flow ports. Hereby, mounting ports are connected typically using mechanical connectors, material flow ports transport solid, liquid, or gaseous material, and energy flow ports describe energy exchange. In contrast, electrical ports represent only energy flow ports, while software ports represent information flow ports. Furthermore, note that in

principle information might also flow over electrical ports and material ports directly. However, in such cases the information has to be encoded using selected electrical or mechanical properties such as voltage or shape.

Approach

The proposed approach envisages to derive the architecture from the requirement specification and the process specification. First, one has to decide whether a decomposition of the system under development is desired or not. For example, a decomposition might be desired because elements can be reused from previous projects saving development cost or developed in parallel saving development time [Thr05]. In contrast, a decomposition might not be desired because additional elements respectively details would not provide enough benefits during preliminary design. Then, one should prefer using mechatronic components at the upper levels and discipline-specific components at the lower levels of the decomposition hierarchy. Consequently, design decisions can be delayed such as using a robot arm or a conveyor belt for transporting solid material between two positions. Note that approaches exist that try automating the decomposition task. For example, for software systems the task can be formulated as optimization problem over measures such as granularity, cohesion, and coupling [HHP02]. In contrast, for mechatronic systems the task can be formulated as clustering problem over system functions and their interactions [vBET10]. These techniques can be used in principle with the proposed approach as well. However, for simplicity the remainder of this thesis concentrates on manual architecture specification and iteration- and increment-based quality assurance.

3.2.3 Behavior specification

In parallel with the architecture specification activity (see Section 3.2.2), the *behavior specification* activity is concerned with describing the reaction of the architectural elements to stimuli from the environment. Note that typically the stimuli are exchanged over the interfaces of the architectural elements such as mechanical, electrical, and digital ports of mechatronic components [Thr05]. Different formalisms exist for describing this reaction at various levels of detail from high-level logical processes to low-level physical dynamics. In the following, background information on behavior specification is provided before sketching the proposed approach.

Background

The most prominent formalisms probably are differential equations and finite-state machines. Differential equations can be used to model behavior over continuous time and

continuous state variables. Consequently, differential equations are well suited to describe physical behavior such as the relationship between force, acceleration, velocity and position. In practice, differential equations are used for modeling both mechanical and electrical behavior [FB02]. In contrast, finite-state machines can be used to model behavior over discrete time and discrete state variables [LT89]. Finite-state machines are well suited to describe software behavior. Note that software behavior is based inherently on electrical behavior over continuous time and continuous state variables. Consequently, finite-state machine378s represent an abstraction from the underlying physical characteristics. Finally, there also exist hybrid formalisms that integrate differential equations and finite-state machines [LSV03]. Consequently, differential equations can be associated with the finite states such that in each state a different continuous behavior can be observed. The hybrid approaches are well suited for coupling mechanical, electrical, and software behavior.

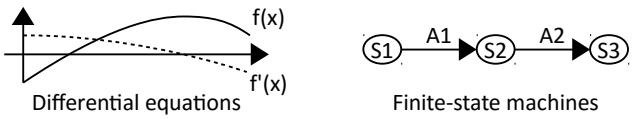


Figure 3.7: Example behavior specifications.

While the software behavior can be designed rather freely, the mechanical and electrical behavior has to obey physical laws. In particular, mechanical behavior comprises multi-body dynamics [Sch97, Sha97] and fluid dynamics [ADD+10]. Multi-body dynamics is concerned with the motion and collision-based interaction of multiple bodies (see Section 2.1.2). Hereby, one can distinguish further between rigid multi-body dynamics [Sch97] and flexible multi-body dynamics [Sha97]. In rigid multi-body dynamics forces only affect the motion of entire bodies, while in flexible multi-body dynamics forces also might affect their shape. Consequently, in flexible multi-body dynamics also the deformation of mechanical elements can be studied. Note that rigid multi-body dynamics represents a simplification of flexible multi-body dynamics. In contrast, fluid dynamics is concerned with the behavior of fluids [ADD+10]. Typically, fluids are modeled using particles with individual physical state and spontaneous interactions. Finally, electrical behavior comprises Maxwell's equations when working with alternating current [Mon03]. Instead, when working with direct current one can use the simplified Kirchhoff's laws.

Approach

For simplicity, the proposed approach envisages to use finite-state machines over discrete time and (possibly) continuous state variables for specifying mechatronic, mechanical, electrical, and software behavior (including spatial motion and its causes explicitly). However, one also can argue that finite-state machines are well-suited for conceptual design. First, they allow one to describe the physical behavior sufficiently precise depending on the discretization. Second, they allow one to abstract from certain physical details such as intermediate material positions during motion. Third, their semantics are easy to explain to potential users (e.g. requirements and process engineers as well as mechanical, electrical, and software engineers) and to implement by potential tool providers. However, a major drawback of the discretization is that critical system states might be neglected or the specified discrete behavior might diverge from the actual physical behavior. Consequently, one has to select carefully the discretization scheme and one has to interpret carefully the model predictions [BR87]. In an advanced version of the approach one might rely on differential equations directly. Additionally, one might include the described laws of physics directly into the model semantics such that they do not have to be modeled explicitly.

3.2.4 Part specification

In parallel with the architecture specification activity (see Section 3.2.2) and the behavior specification activity (see Section 3.2.3), the *part specification* activity is concerned with describing the geometric shape of mechanical and electrical elements. Note that the geometric shape represents an important piece of information during conceptual design of cyber-physical manufacturing systems. In particular, the geometric shape is needed to involve the mechanical engineers in the design process as well as to communicate a clear picture among the different stakeholders and to ensure the collision-free operation of the system. In principle, a variety of techniques exist for describing geometric shapes coming from various fields of research such as civil engineering, mechanical engineering, and computer graphics. In the following, again background information on part specification is provided before sketching the proposed approach.

Background

Fundamentally, one can distinguish between solid representations and boundary representations [RV83], which have their own advantages and disadvantages. Figure 3.8 illustrates the difference between solid representations and boundary representations. Solid representations describe the points in space that belong to a geometric shape,

while boundary/shell representations describe the surface of the geometric shape instead. Probably the most prominent solid representation is provided by constructive solid geometry [RV77]. In constructive solid geometry geometric shapes are represented as sets of points in space. At its core primitive geometric shapes are provided such as cubes, spheres, and cylinders. Then, the classical set operations union, difference, and intersection are provided to compose geometric shapes. Finally, classical geometric transformations such as displacements, isometries, and similarities are provided to change, for example, the position, orientation, and scale of geometric shapes. In contrast, a popular boundary representation is provided by non-uniform rational basis splines [Pie91]. Non-uniform rational basis splines allow one to describe the surface of geometric shapes using control points, their weights, a parameter interval, and knots over the parameter interval. Then, the surface can be defined as a function from the parameter interval to the points in space. Hereby, each parameter value is mapped to a linear combination of the control points. The coefficients of the linear combination are derived from the weights and the knots respectively. In particular, the knots specify the parameter range where certain control points are active.

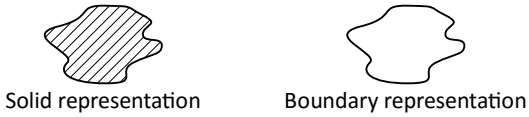


Figure 3.8: Example part specifications.

Besides constructive solid geometry and non-uniform rational basis splines there exist numerous other techniques for describing geometric shapes [Req80]. The most prominent approaches are spatial occupancy enumeration, cell decomposition, and the sweep representation. Spatial occupancy enumeration divides space into cubic cells respectively voxels of equal size. Then geometric shapes are described in terms of the comprised cells/voxels. A downside of the approach is that descriptions of geometric shapes tend to be rather verbose. In contrast, cell decomposition divides space into cells of arbitrary shape and size. Consequently, cell decomposition represents a generalization of spatial occupancy enumeration. Furthermore, cell decomposition is able to produce less verbose descriptions of geometric shapes than spatial occupancy enumeration. However, typically such cell decompositions are difficult to generate/compute. Finally, the sweep representation describes geometric shapes in terms of three-dimensional planes intersecting with the shape and the two-dimensional contours of the shape in these planes. Hereby, one can distinguish between rotational and translational sweeping. In rotational sweeping the planes rotate around one single axis, while in translational sweeping the

panes translate in one single direction. At last, it should be noted that approaches exist to convert between the different representations of geometric shapes [Req80]. However, in some cases only approximate representations can be produced.

Approach

The proposed approach envisages to use constructive solid geometry for describing the geometric shape of mechanical and electrical parts. One can argue that constructive solid geometry is well-suited for describing geometric shapes during conceptual design. In particular, constructive solid geometry provides a solid representation of geometric shapes, from which exact boundary representations can be calculated using boundary evaluation algorithms [RV85]. Furthermore, constructive solid geometry representations can be annotated with features, tolerances, and other attributes to guide the manufacturing process planning (see Section 3.1.2 and [RC86]). One downside is that smooth contours of complex objects can be described using, for example, non-uniform rational basis splines more easily. However, one can argue that during conceptual design the exact geometric shape of mechanical and electrical parts is not important. Rather, the proposed approach suggests working with coarse over-approximations of the geometric shapes to save time, while retaining confidence, for example, on collision-free system operation.

3.3 Summary and outlook

This chapter explained the test-driven design method, by means of which the challenges of mechanical dominance and sequential engineering described in the first chapter are tackled. First, the preparation phase has been described concentrating on the requirements, the input and output materials, the manufacturing processes, and test cases (see Section 3.1). Then, the implementation phase is concerned with elaborating the implementation details incrementally consisting of architectural, behavioral, and geometrical knowledge (see Section 3.2). The next chapter describes the theoretical foundations upon which the modeling technique for capturing the related design information is built.four

4 Theoretical foundation

To realize the proposed test-driven method (see Chapter 3) an existing formalism is exploited and extended for describing the structure and behavior of cyber-physical manufacturing systems. In particular, this thesis bases on two corner stones: Section 4.1 presents a formal theory for the component-based specification of software systems called FOCUS. In particular, the FOCUS theory provides mathematical notions of *components* and their *composition*, which are key concepts for handling the *complexity* of modern (software) systems. Then, Section 4.2 explains an extension of the original FOCUS theory to spatio-temporal systems, i.e. systems having a potentially time variable spatial extent, called STEM. Most notably, the STEM theory includes mathematical notations of *space*, *volume*, *position*, *motion*, and *collision*. Note that the STEM theory is well suited for describing the structure and behavior of cyber-physical manufacturing systems, whose spatial extent and motion are critical properties – besides others – during the conceptual design phase. However, also note that the theory does not capture customer requirements, manufacturing processes, and test cases yet as explained in Sections 2.1.3 and 2.2, which is why it cannot be used off the shelve for the given purposes.

4.1 Focus on components and streams (FOCUS)

FOCUS [BS01, Bro07, Bro10] provides a formal theory and engineering methodology for the component-based development of software systems. The core concepts of the theory are *streams* (see Section 4.1.1), *channels* (see Section 4.1.2), *components* and their *composition* (see Section 4.1.3), as well as *state transition systems* and their *computation* (see Section 4.1.4). In the following the concepts and their relations are explained in detail.

4.1.1 Streams

Streams represent the most basic concept of the FOCUS theory. Streams are defined as sequences of arbitrary *messages*. These sequences can be used for describing various aspects of component-based software systems. For example, the reaction of components

to stimuli from the environment or the interaction between components can be described concisely in terms of input and output message streams.

Definition 4.1 (Streams) *Let M be an arbitrary (message) set. We define the sets M^n with $n \in \mathbb{N}$, M^*, M^∞, and M^ω such that*

$$M^n = \{[0, n] \to M\},\ M^* = \bigcup_{n \in \mathbb{N}} M^n,\ M^\infty = \mathbb{N} \to M,\ \text{and}\ M^\omega = M^* \cup M^\infty.$$

We call M^* the set of finite streams over M, M^∞ the set of infinite streams over M, and M^ω the set of streams over M.

To work with streams one particular operator is needed, which is called the *prefix operator*. The prefix operator allows one to extract a finite prefix stream of a predefined length from a given infinite original stream. Note that various other operators exist for stream such as the *concatenation operator* [BS01], which are not required for the given purposes.

Definition 4.2 (Prefix operator) *Let M be an arbitrary (message) set. We define the binary operator $\downarrow\colon M^\infty \times \mathbb{N} \to M^\omega$ such that*

$$\forall m \in M^\infty, n \in \mathbb{N} : (m \downarrow n : [0, n] \to M \wedge \forall i \in [0, n] : (m \downarrow n)(i) = m(i)).$$

We call $\downarrow$ the (stream) prefix operator.

Note that – in contrast to other source [BS01] – the prefix operator is defined only for infinite streams here. This limitation is sufficient for the given needs. In other cases the prefix operator has to be defined for finite streams as well, which complicates the previous definition and requires the stream *length operator* to be introduced as well.

4.1.2 Channels

Based on the concept of streams (see Section 4.1.1) the concept of channels is derived. Channels essentially are *labeled* and *typed* message streams, which can be used – besides others – for communication between components. While the channel labels are given by a basic set of channel labels C, the channel types are defined by means of a *type mapping*. The type mapping, in turn, is based on a basic set of type labels T.

Definition 4.3 (Type mapping) *Let C be a set of channel labels and T be a set of type labels. We call the function $M' : C \to T$ a type mapping.*

Note that until now types are defined only by their label, but the relation to messages remains unclear. In the following, each type is associated with a subset of messages from an arbitrary, but predefined set of messages M by means of a *message mapping*. The mapping defines for each type label the subset of messages that belong to the type. Note that a very basic type system is used here, which is sufficient for the given needs. More advanced type systems exist comprising, for example, type *inheritance* and *polymorphism*, which are not required here.

Definition 4.4 (Message mapping) *Let T be a set of type labels and M be an arbitrary (message) set. We call the function $M'' : T \to \mathbb{P}(M)$ a message mapping.*

Based on the concept of type and message mappings the concept of *channel assignments* is introduced. A channel assignment maps each channel label to a type-compliant message. To be type-compliant, the message has to belong to the subset of messages associated to the respective channel type by means of the message mapping. Note that channel assignments do not use the concept of streams yet, but represent time instants of communication between components instead.

Definition 4.5 (Channel assignments) *Let C be a set of channel labels, T be a set of type labels, M be an arbitrary (message) set, $\bot \notin M$ be the empty message, $M' : C \to T$ be a type mapping, and $M'' : T \to \mathbb{P}(M)$ be a message mapping. We define the set $\bar{C} \subseteq \{C \to M \cup \{\bot\}\}$ such that*

$$\bar{C} = \{a : C \to M \cup \{\bot\} \mid \forall \kappa \in C : a(\kappa) \in M''(M'(\kappa)) \cup \{\bot\}\}.$$

We call $\bar{C}$ the set of channel assignments over C.

While channel assignments allow one to describe time instants of communication, they cannot be used to describe communication over a longer range of time. To allow one to describe communication over a longer range of time, the concept of *channel histories* is introduced. Channel histories assign to each channel label a stream of type-compliant messages instead of single type-compliant messages only. Then, to be type-compliant each message of the stream has to belong to the subset of messages associated to the respective channel type by means of the message mapping.

Definition 4.6 (Channel histories) *Let C be a set of channel labels, T be a set of type labels, M be an arbitrary (message) set, $\bot \notin M$ be the empty message, $M' : C \to T$ be a type mapping, and $M'' : T \to \mathbb{P}(M)$ be a message mapping. We define the set $\vec{C} \subseteq \{C \to (M \cup \{\bot\})^\infty\}$ such that*

$$\vec{C} = \{h : C \to (M \cup \{\bot\})^\infty \mid \forall \kappa \in C : h(\kappa) \in (M''(M'(\kappa)) \cup \{\bot\})^\infty\}.$$

We call $\vec{C}$ the set of channel histories over C.

Finally, to work with channel histories one particular operator is needed, which is called the *limitation operator*. The limitation operator allows one the reduce a channel history to only a subset of channels from the set of all possible channels. In particular, the limitation operator is needed to define the composition operator for components in Section 4.1.3. Note that many more operators exist for working with channel histories [BS01], which are not required here.

Definition 4.7 (Limitation operator) *Let C be a set of channel labels and $C' \subseteq C$ be a subset of channel labels. We define the unary operator $|_{C'} : \vec{C} \to \vec{C'}$ such that*

$$\forall h \in \vec{C}, \kappa \in C' : (h|_{C'}(\kappa) = h(\kappa)).$$

We call $|_{C'}$ the (channel history) limitation operator over C'.

Note that in this work channel histories are defined over infinite message streams only. Again, this limitation is sufficient for the given needs. In other cases channel histories have to be defined over finite streams as well, which complicates the previous definitions [BS01].

4.1.3 Components

Based on the concept of streams (see Section 4.1.1) and channels (see Section 4.1.2) the concept of components is derived. Components are the structural building blocks of software systems in FOCUS. Components provide a syntactic interface consisting of *input* and *output* channels including their type information. Furthermore, components provide a semantic interface describing the mapping from input to output channel histories. Consequently, the semantic interface describes the reactions of components to different stimuli. These reactions also represent the behavior of components.

Definition 4.8 (Components) *Let I be a set of input channel labels and O be a set of output channel labels. We call the function $C : \vec{I} \to \mathbb{P}(\vec{O})$ a component over I and O.*

In the previous formalism multiple reactions $o \in C(i)$ to the same stimulus $i \in \vec{I}$ are possible. Specifying multiple reactions is useful, for example, in the case of uncertainty. However, in many cases one would like to specify only one possible reaction of the component to the same stimulus, such that the component behaves predictably in every situation. We call such components *deterministic* components.

Definition 4.9 (Deterministic components) *Let C be a component over I and O such that $\forall i \in \vec{I} : |C(i) = 1|$. We call C a deterministic component.*

Then, the previous definitions do not state anything about the messages contained in the stimuli $i \in \vec{I}$ and the reactions $o \in C(i)$ of component C. Consequently, reactions $o(\omega)(n)$ at output channel $\omega \in O$ and time point $n \in \mathbb{N}$ might depend on stimuli $i(\iota)(m)$ at input channel $\iota \in I$ and some future time point $m \in \mathbb{N}$ with $m > n$, which is hard to realize in practice. To support the realizability of FOCUS specifications the concept of *weakly* and *strongly causal* components has been introduced [Bro07].

Definition 4.10 (Weakly causal components) *Let C be a component over I and O such that for each $i_1, i_2 \in \vec{I}$ with $C(i_1) \neq \emptyset$ and $C(i_2) \neq \emptyset$:*

$$\forall n \in \mathbb{N} : (i_1 \downarrow n = i_2 \downarrow n \Rightarrow \{o \downarrow n \in C(i_1)\} = \{o \downarrow n \in C(i_2)\}).$$

We call C a weakly causal component.

Weakly causal components require that the reactions until some time point $n \in \mathbb{N}$ depend only on the stimuli until the same time point. This property still supports the immediate reaction $o(\omega)(n)$ at output channel $\omega \in O$ to stimulus $i(\iota)(n)$ at input channel $\iota \in I$ without time delay. Specifying immediate reactions might be useful in some cases, but unrealistic in other cases, e.g. due to computation time needed. An even stronger restriction of the reaction time is given by the notion of strongly causal components, which require a reaction delay of at least one time unit.

Definition 4.11 (Strongly causal components) *Let C be a component over I and O such that for each $i_1, i_2 \in \vec{I}$ with $C(i_1) \neq \emptyset$ and $C(i_2) \neq \emptyset$:*

$$\forall n \in \mathbb{N} : (i_1 \downarrow n = i_2 \downarrow n \Rightarrow \{o \downarrow (n+1) \in C(i_1)\} = \{o \downarrow (n+1) \in C(i_2)\}).$$

We call C a strongly causal component.

Based on the previous formulations the *composition operator* is derived. The composition operator allows one to construct a new component C over I, O from two given components C_1 over I_1, O_1 and C_2 over I_2, O_2. In particular, the construction describes how the original syntactic and semantic interfaces of components C_1 and C_2 are combined to form a new syntactic and semantic interface of component C. Note that the composition operations also define the internal structure of components and entire software systems.

Definition 4.12 (Composition operator) *Let $\mathcal{C}$ be the set of all components. We define the binary operator $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ such that for each component $C_1 \in \mathcal{C}$ over I_1, O_1 and component $C_2 \in \mathcal{C}$ over I_2, O_2 with disjoint output channels $O_1 \cap O_2 = \emptyset$ as well as combined input channels $I = (I_1 \backslash O_2) \cup (I_2 \backslash O_1)$ and output channels $O = O_1 \cup O_2$:*

$$C_1 \otimes C_2 : \vec{I} \to \mathbb{P}(\vec{O})$$

and for each combined input history $i \in \vec{I}$ and output history $o \in (C_1 \otimes C_2)(i)$ there exists an extended input history $i' \in \vec{Z}$ with $Z = I_1 \cup I_2$ and $i'|_I = i$ such that

$$o|_{O_1} \in C_1(i'|_{I_1}) \wedge o|_{O_2} \in C_2(i'|_{I_2}).$$

We call $\otimes$ the (component) composition operator.

In summary, note that the component formalism, which has been introduced here, uses infinite input and output channel histories only. As mentioned previously, this limitation is sufficient for the given needs. However, in other cases components have to be defined over finite channel histories as well, which complicates the previous definitions [BS01]. In particular, the definition of the composition operator has to be revised to consider channel histories of different lengths.

4.1.4 State transition systems

Then, the question arises how to describe the semantic interface of components (i.e. the mapping from input to output channel histories as introduced in Section 4.1.3) in practical applications. One popular formalism for describing this simulus-reaction mapping – besides input-ouput table specifications [Hum11] – are the so-called state transition systems [Bro07]. State transition systems comprise the same input and output channels as components. Furthermore, state transition systems include *variables* and *states*. The variables and states can be used to store information during execution of the state transition system. Finally, state transition systems define a *state transition function*. The state transition function describes how the states, output channel assignments, and variable assignments evolve during execution based on the previous states, input channel assignments, and variable assignments.

Definition 4.13 (State transition function) *Let I be a set of input channel labels, O be a set of output channel labels, V be a set of variable labels, and S be a set of state labels. We define the function T such that*

$$T : (S \times \bar{I} \times \bar{V}) \to \mathbb{P}(S \times \bar{O} \times \bar{V}).$$

We call T a state transition function over I, O, V, and S.

Note that the state transition function T encodes the transitions from previous states $s \in S$, input channel assignments $i \in \bar{I}$, and previous variable assignments $v \in \bar{V}$ to following states $s' \in S$, output channel assignments $o \in \bar{O}$, and following variable assignments $v' \in \bar{V}$. In particular, the definition of the state transition function is based on channel assignments rather than channel histories. Consequently, the state transition function describes reactions of the state transition system at time instants only. To complete the definition of state transition systems, one has to introduce the *initial state* and the *initial variable assignments*.

Definition 4.14 (State transition system) *Let T be a state transition function over I, O, V, S, $s_0 \in S$ be an initial state label, and $v_0 \in \bar{V}$ be an initial variable assignment. We call the tuple (T, s_0, v_0) a state transition system over I,O,V, and S.*

At last, one has to define the relation between the state transition system (T, s_0, v_0), an arbitrary input channel assignment stream $i \in \bar{I}^\infty$, as well as the generated output channel assignment stream $o \in \bar{O}^\infty$. Note that in contrast to components and their semantic interface (see Section 4.1.3) infinite channel assignment streams are used here rather than channel histories. However, the both concepts can be converted easily into each other. In the following, the stimulus-reaction relationship of state transition systems is described by means of state transition system *computations*. A state transition system computation requires that the output channel assignments are generated from the input channel assignments using the state transition function.

Definition 4.15 (Computation) *Let (T, s_0, v_0) be a state transition system over I, O, V, S and $i \in \bar{I}^\infty$ be an arbitrary input stream. We define the state stream $s \in S^\infty$ with initial assignment $s(0) = s_0$, the variable stream $v \in \bar{V}^\infty$ with initial assignment $v(0) = v_0$, and the output stream $o \in \bar{O}^\infty$ such that*

$$\forall n \in \mathbb{N} : (s(n+1), o(n+1), v(n+1)) = T(s(n), i(n), v(n)).$$

We call the tuple (i, s, v, o) a (state transition system) computation over (T, s_0, v_0).

Finally, note that state transition system computations are strongly causal by definition. Consequently, state transition systems can be used to describe strongly causal components, which are realizable by definition. The realizability is one reason, why state transition systems are used widely in practice for describing stimulus-reaction mappings. Also, techniques exist to generate program code from state transition systems, which can be run on actual computing equipment [RHZ14a].

4.2 Spatio-temporal engineering models (STEM)

While the FOCUS theory (see Section 4.1) is well suited for describing software systems, it provides limited support for spatial aspects such as volumes, motions, and collisions. To overcome this limitation, the STEM theory [HB08, BHHL09, Hum09, BH10, Hum11] defines an extension to the original FOCUS theory tailored for the component-based specification of spatio-temporal systems, i.e. systems comprising a potentially time variable spatial extent. The core concepts of the STEM theory are *transformable collision spaces* (see Section 4.2.1), *spatio-temporal components* (see Section 4.2.2), and *extended spatio-temporal components* (see Section 4.2.3). Note that, in contrast to the FOCUS theory, the STEM theory does not provide a particular engineering method, but concentrates on the specification formalism and its peculiarities. In the following each concept of this formalism is described in detail.

4.2.1 Transformable collision spaces

Before defining the notion of spatio-temporal components one has to introduce a notion of space. In principle, several notions of space can be used such as the two- or three-dimensional Euclidean space with Cartesian coordinates. However, the core STEM theory [Hum11] uses a more abstract notion of space, which is called the transformable collision space. Fundamentally, a transformable collision space consists of *volumes* (e.g. spheres), a *collision relation* between volumes (e.g. two intersecting spheres), a *union operator* for constructing combined volumes (e.g. a cube and a sphere), and *volume transformations* for changing volumes (i.e. mapping two different volumes onto each other). The volume transformation can be used, for example, for translating and rotating volumes. But also more complex transformations can be described.

Definition 4.16 (Transformable collision spaces) *Let V be a set of volumes, $0_V \in V$ be the empty volume, $\bowtie \subseteq V \times V$ be a collision relation, $\sqcup : V \times V \to V$ be a union operator, and $T = \{V \to V\}$ be a set of volume transformations. We call the tuple $(V, 0_V, \bowtie, \sqcup, T)$ a transformable collision space.*

To work with transformable collision spaces two particular operators are needed. The first operator are the so-called *spatial selection predicates*. Spatial selection predicates select specific volumes $v \in V$ of a transformable collision space $(V, 0_V, \bowtie, \sqcup, T)$, while filtering out others. The selection itself is defined as a mapping from volumes v to Boolean values $b \in \mathbb{B} = \{true, false\}$.

Definition 4.17 (Spatial selection predicates) *Let V be a set of volumes. We call the function $\sigma : V \to \mathbb{B}$ a spatial selection predicate over V.*

The second operator, which is required for working with transformable collision spaces, are the so-called *integrated transformation streams*. An integrated transformation stream maps a stream of volume transformations $t \in T^\infty$ to a stream of integrated (i.e. concatenated) volume transformations $t' \in T^\infty$ such that $t'(0) = t(0)$, $t'(1) = t(1) \circ t(0)$, $t'(2) = t(2) \circ t(1) \circ t(0)$ and so on. Consequently, the integrated transformation stream describes the intermediate results of successively applying a stream of transformations to a volume. Note that in the following the operator $\circ$ represents the *function concatenation operator* with $f \circ g(x) = f(g(x))$ rather than the stream concatenation operator of the FOCUS theory.

Definition 4.18 (Integrated transformation streams) *Let T be a set of volume transformations and $t \in T^\infty$ be a stream of volume transformations. We define the function $its : T^\infty \to T^\infty$ such that*

$$its(t)(0) = t(0) \wedge \forall n \in \mathbb{N} : its(t)(n+1) = t(n+1) \circ its(t)(n).$$

We call $its(t)$ the integrated transformation stream over t.

Finally, note that in concrete applications the abstract notion of transformable collision spaces $(V, 0_V, \bowtie, \sqcup, T)$ can be substituted by any concrete notion of space. Probably the most widely used notion of space are the two- and three-dimensional Euclidean spaces with Cartesian coordinate system [Cox61]. The three-dimension Euclidean notion of space is used also in the remainder of this thesis.

4.2.2 Spatio-temporal components

Based on the original concept of components in the FOCUS theory (see Section 4.1.3) and transformable collision spaces (see Section 4.2.2) the STEM theory introduces the concept of spatio-temporal components. Before defining spatio-temporal components, the concept of *spatio-temporal functions* needs to be introduced, which represents a pre-stage of spatio-temporal components. In addition to input and output channels spatio-temporal functions comprise *detectors*, *parts*, and *movers*. Detectors and parts, in turn, are defined by volumes of the underlying transformable collision space (see Section 4.2.1). Detectors can be used for modeling, e.g., light barriers and other kinds of sensors. Hereby, the detector volume defines the portion of space observed by the component. In contrast, parts describe the solid physical bodies of the component and, hence, its spatial extent. Finally, movers can be used to model kinematic chains between components and other motion effects.

Definition 4.19 (Spatio-temporal functions) *Let I be a set of input channel labels, O be a set of output channel labels, D be a set of detector labels, P be a set of part labels, M be a set of mover labels, and $(V, 0_V, \bowtie, \sqcup, T)$ be a transformable collision space. We define the set of detector volumes $\vec{D} = \{D \to V^\infty\}$, the set of part volumes $\vec{P} = \{P \to V^\infty\}$, the set of mover transformations $\vec{M} = \{M \to T^\infty\}$, the set of detector activations $\vec{A}(D) = \{D \to \mathbb{B}^\infty\}$ and the function $C : \vec{I} \times \vec{A}(D) \to \mathbb{P}(\vec{O} \times \vec{D} \times \vec{P} \times \vec{M})$. We call C a spatio-temporal function over I, O, D, P, M and $(V, 0_V, \bowtie, \sqcup, T)$.*

Note that the previous definition of spatio-temporal functions C does not make any assumptions about the mapping from stimuli $(i, a) \in \vec{I} \times \vec{A}(D)$ to reactions $(o, d, p, m) \in C(i, a)$. Similar to components in the FOCUS theory (see Section 4.1.3) one might require the mapping to be, e.g., strongly causal to be realizable. In addition, one important property of spatio-temporal functions is that the detector activations a report collisions between the detector volumes d and the part volumes p correctly. Such spatio-temporal functions are called *collision sensing* spatio-temporal functions.

Definition 4.20 (Collision sensing spatio-temporal functions) *Let $(V, 0_V, \bowtie, \sqcup, T)$ be a transformable collision space and C be a spatio-temporal component over I, O, D, P, M and $(V, 0_V, \bowtie, \sqcup, T)$ such that for each $(i, a) \in \vec{I} \times \vec{A}(D)$ and $(o, d, p, m) \in C(i, a)$:*

$$\forall n \in \mathbb{N}, \delta \in D : (a(\delta)(n) \Leftarrow \bigvee_{\pi \in P} d(\delta)(n) \bowtie p(\pi)(n)).$$

We call C a collision sensing spatio-temporal function.

Note that detector activations $a(\delta)(n)$ with detector $\delta \in D$ and time point $n \in \mathbb{N}$ might also be caused by collisions between the respective detector volume $d(\delta)(n)$ and a part volume from the environment. Consequently, the part activations a of component C follow from collisions between detectors d and parts p of component C, but not vice versa. In other words, a collision sensing spatio-temporal function might also specify the reactions to detector activations caused by part volumes from the environment. From the definition of collision sensing spatio-temporal functions one directly yields the definition of spatio-temporal components.

Definition 4.21 (Spatio-temporal components) *Let C be a collision sensing spatio-temporal function. We call C a spatio-temporal component.*

One important property of spatio-temporal components is their collision free operation. This property is useful particularly in domains where collisions between solid

physical parts might cause severe damage to the system, the business, and / or humans. Such situations can be found typically – besides other industries – in the machine tools industry [LDK03], which is one of the key drivers for this doctoral thesis. Spatio-temporal components that can be operated without collisions are called *collision free* spatio-temporal components.

Definition 4.22 (Collision free spatio-temporal components) *Let $(V, 0_V, \bowtie, \sqcup, T)$ be a transformable collision space and C be a spatio-temporal component over I, O, D, P, M and $(V, 0_V, \bowtie, \sqcup, T)$ such that for each $(i, a) \in \vec{I} \times \vec{A}(D)$ and $(o, d, p, m) \in C(i, a)$:*

$$\forall n \in \mathbb{N}, \pi_1, \pi_2 \in P : (p(\pi_1)(n) \bowtie p(\pi_2)(n) \Rightarrow \pi_1 = \pi_2).$$

We call C a collision free spatio-temporal component.

Then, the question arises how spatio-temporal components C over I, O, D, P, M can be constructed from existing spatio-temporal components C_1 over I_1, O_1, D_1, P_1, M_1 and C_2 over I_2, O_2, D_2, P_2, M_2. This construction is supported by means of the *spatio-temporal composition operator*. Note that in addition to the composition operator of the FOCUS theory (see Section 4.1.3), the spatio-temporal composition operator needs to ensure that the result is a collision sensing spatio-temporal function. Consequently, the detector activations of component C_1 have to report also collisions between detector volumes of component C_1 and part volumes C_2 and vice versa as defined in the following.

Definition 4.23 (Spatio-temporal composition operator) *Let $(V, 0_V, \bowtie, \sqcup, T)$ be a transformable collision space and $\mathcal{C}$ be the set of all spatio-temporal components over $(V, 0_V, \bowtie, \sqcup, T)$. We define the binary operator $\odot : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ such that for each spatio-temporal component $C_1 \in \mathcal{C}$ over I_1, O_1, D_1, P_1, M_1 and spatio-temporal component $C_2 \in \mathcal{C}$ over I_2, O_2, D_2, P_2, M_2 with disjoint outputs $O_1 \cap O_2 = \emptyset$, detectors $D_1 \cap D_2 = \emptyset$, parts $P_1 \cap P_2 = \emptyset$, and movers $M_1 \cap M_2 = \emptyset$ as well as combined inputs $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$, outputs $O = O_1 \cup O_2$, detectors $D = D_1 \cup D_2$, parts $P = P_1 \cup P_2$, and movers $M = M_1 \cup M_2$:*

$$C_1 \odot C_2 : \vec{I} \times \vec{A}(D) \to \mathbb{P}(\vec{O} \times \vec{D} \times \vec{P} \times \vec{M})$$

and for each stimulus $(i, a) \in \vec{I} \times \vec{A}(D)$ and reaction $(o, d, p, m) \in (C_1 \odot C_2)(i, a)$ there exists an extended input history $i' \in \vec{Z}$ with $Z = I_1 \cup I_2$ and $i'|_I = i$ such that

$$(o|_{O_1}, d|_{D_1}, p|_{P_1}, m|_{M_1}) \in C_1(i'|_{I_1}, a|_{D_1}) \wedge (o|_{O_2}, d|_{D_2}, p|_{P_2}, m|_{M_2}) \in C_2(i'|_{I_2}, a|_{D_2})$$

and $C_1 \odot C_2$ is a collision sensing spatio-temporal function. We call $\odot$ the spatio-temporal (component) composition operator over $\mathcal{C}$.

Note that, while the result of the composition operator is required to be a collision sensing spatio-temporal function, the result might not be a collision free spatio-temporal component, e.g., due to uncoordinated motions of the components C_1 and C_2. In other words, the providers of the spatio-temporal components C_1 and C_2 have to ensure that, when integrating C_1 and C_2, no severe damage occurs during system operation. Finally, the effect of movers onto spatio-temporal components needs to be considered, which has been neglected so far. Before describing this effect, *position operator* has to be introduced. The position operator allows one to construct a possibly translated and rotated version of some spatio-temporal component C. Hereby, a transformation stream $t \in T^\infty$ defines the spatial transformations applied to component C throughout its lifetime.

Definition 4.24 (Position operator) *Let $(V, 0_V, \bowtie, \sqcup, T)$ be a transformable collision space and $\mathcal{C}$ be the set of all spatio-temporal components over $(V, 0_V, \bowtie, \sqcup, T)$. We define the binary operator $pos : \mathcal{C} \times T^\infty \to \mathcal{C}$ such that for each spatio-temporal component $C \in \mathcal{C}$ over I, O, D, P, M and transformation stream $t \in T^\infty$:*

$$pos(C, t) : \vec{I} \times \vec{A}(D) \to \vec{O} \times \vec{D} \times \vec{P} \times \vec{M}$$

and for each stimulus $(i, a) \in \vec{I} \times \vec{A}(D)$ and reaction $(o, d, p, m) \in pos(C, t)(i, a)$ there exists a reaction $(o, d', p', m') \in C(i, a)$ such that for each time point $n \in \mathbb{N}$ the detectors are transformed by the integrated transformation stream over t, i.e.

$$\forall \delta \in D : d(\delta)(n) = its(t)(n)(d'(\delta)(n))$$

and the parts are transformed by the integrated transformation stream over t, i.e.

$$\forall \pi \in P : p(\pi)(n) = its(t)(n)(p'(\pi)(n))$$

and the movers are transformed by the original transformation stream t, i.e.

$$\forall \mu \in M : m(\mu)(n) = t(n) \circ m'(\mu)(n).$$

We call pos the (spatio-temporal component) position operator.

Note that the detector volumes $d(\delta)(n)$ with detector label $\delta \in D$ and the part volumes $p(\pi)(n)$ with part label $\pi \in P$ are affected by the elements of the integrated transition stream $its(t)(n)$ at time point $n \in \mathbb{N}$, while the mover transforms $m(\mu)(n)$ with mover label $\mu \in M$ are transformed by the elements of the original transition stream $t(n)$ directly. The reason is that the transformation stream $t \in T^\infty$ itself might be given by a

mover history $m''(\mu') \in T^\infty$ of some positioned spatio-temporal component $pos(C', t')$ over I', O', D', P', M' with mover label $\mu' \in M'$ and transformation stream $t' \in T^\infty$. Therefore, the transformation stream elements $t(n) = m''(\mu')(n)$ at time point $n \in \mathbb{N}$ might include the transformation stream elements $t'(n)$. Now, remember that the position operator *pos* applies the integrated transformation stream elements $its(t)(n)$ to the detector volumes $d(\delta)(n)$ and the part labels $p(\pi)(n)$. Consequently, the position operator applies the integrated transformation stream elements $its(m''(\mu))(n)$ to the detector and part volumes. The integrated transformation stream elements $its(m''(\mu))(n)$, in turn, include the transformation stream elements $t'(m)$ with $m \in \mathbb{N}, m \leq n$. In other words, the position operator integrates the transformations applied to mover transforms later, i.e. when applying the mover transforms to detector and part volumes [Hum11]. Finally, based on the position operator the *motion operator* can be introduced, which allows one to link movers to spatio-temporal components. In particular, the motion operator allows one to transmit some mover transformation stream $m_1(\mu) \in T^\infty$ with $\mu \in M_1$ of component C_1 onto some component C_2.

Definition 4.25 (Motion operator) *Let $(V, 0_V, \bowtie, \sqcup, T)$ be a transformable collision space and $\mathcal{C}$ be the set of all spatio-temporal components over $(V, 0_V, \bowtie, \sqcup, T)$. We define for each spatio-temporal component $C_1 \in \mathcal{C}$ over I_1, O_1, D_1, P_1, M_1 and spatio-temporal component $C_2 \in \mathcal{C}$ over I_2, O_2, D_2, P_2, M_2 with disjoint outputs $O_1 \cap O_2 = \emptyset$, detectors $D_1 \cap D_2 = \emptyset$, parts $P_1 \cap P_2 = \emptyset$, and movers $M_1 \cap M_2 = \emptyset$ as well as combined inputs $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$, outputs $O = O_1 \cup O_2$, detectors $D = D_1 \cup D_2$, parts $P = P_1 \cup P_2$, and movers $M = M_1 \cup M_2$, as well as mover $\mu \in M_1$ the binary operator $\vdash_\mu \colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ such that*

$$C_1 \vdash_\mu C_2 : \vec{I} \times \vec{A}(D) \to \mathbb{P}(\vec{O} \times \vec{D} \times \vec{P} \times \vec{M})$$

and for each stimulus $(i, a) \in \vec{I} \times \vec{A}(D)$ and reaction $(o, d, p, m) \in (C_1 \vdash_\mu C_2)(i, a)$:

$$(o, d, p, m) \in (C_1 \odot pos(C_2, m(\mu)))(i, a)$$

We call $\vdash_\mu$ the (spatio-temporal component) motion operator over μ.

The motion operator allows one to build kinematic chains, e.g., for modeling robot arms and other mechanical mechanisms [Hum11]. Hereby, each segment of the robot arm C can be represented by a spatio-temporal component C_i with $i, n \in \mathbb{N}, i < n$ and $n > 1$ representing the length of the kinematic chain. Then, the individual segments can be composed by means of the motion operator one after the other, i.e. $C = C_0 \vdash_{\mu_0} (C_1 \vdash_{\mu_1} (C_2 \vdash_{\mu_2} (...)))$ with mover labels $\mu_j \in M_j$ and $j \in \mathbb{N}, j < n - 1$. Hereby, the kinematic chain is given by the order of composition. Finally, note that the position and motion operators make sure that the mover transformation streams $m(\mu_i)$ are propagated along the kinematic chain to all follow-up segments C_k with $k \in \mathbb{N}, i < k < n$.

4.2.3 Extended spatio-temporal components

Based on the concepts of transformable collision spaces (see Section 4.2.1) and spatio-temporal components (see Section 4.2.2) the concept of extended spatio-temporal components is derived. Extended spatio-temporal components add means for expressing, e.g., dynamic material handling operations in addition to fixed kinematic mechanisms. Again, before introducing extended spatio-temporal coomponents, extended spatio-temporal functions need to be defined, which represent a pre-stage of the former. In addition to input and output channels as well as detectors, parts, and movers extended spatio-temporal functions introduce *bindings*, *entries*, and *exits*. Entries can be used to generate new spatio-temporal components at runtime, while exits can be used to remove the generated components later. A generated component typically is the material processed by the material handling system. In contrast, bindings can be used to communicate with generated components via input and output channels or to move generated components via movers. Consequently, it also is possible to read information from generated components. Technically, such information exchange can be implemented, for example, using radio frequency identification (RFID) [Rob06] tags attached to the material and RFID readers deployed to the material handling system.

Definition 4.26 (Extended spatio-temporal functions) *Let* $(V, 0_V, \bowtie, \sqcup, T)$ *be a transformable collision space,* $\mathcal{S}$ *be the set of all spatial selection predicates over* V*,* $\mathcal{C}$ *be the set of spatio-temporal components over* $(V, 0_V, \bowtie, \sqcup, T)$*,* I *be a set of input channel labels,* O *be a set of output channel labels,* D *be a set of detector labels,* P *be a set of part labels,* M *be a set of mover labels,* B *be a set of binding labels,* Y *be a set of entry labels, and* X *be a set of exit labels. We define the set of binding conditions* $\vec{B} = \{B \to (\mathcal{S} \times (I \cup O \cup M))^\infty\}$*, the set of generation events* $\vec{Y} = \{Y \to (T \times \mathcal{C})^\infty\}$*, the set of exit conditions* $\vec{X} = \{X \to \mathcal{S}^\infty\}$*, and the function* $C : \vec{I} \times \vec{A}(D) \to \mathbb{P}(\vec{O} \times \vec{D} \times \vec{P} \times \vec{M} \times \vec{B} \times \vec{Y} \times \vec{X})$*. We call* C *an extended spatio-temporal function over* I*,* O*,* D*,* P*,* M*,* B*,* Y*,* X *and* $(V, 0_V, \bowtie, \sqcup, T)$*.*

Before deriving properties of extended spatio-temporal functions C the concept of *generated component streams* needs to be defined. A generated component stream stores for each time step $n \in \mathbb{N}$ of an execution $(o, d, p, m, b, y, x) \in C(i, a)$ with stimulus $(i, a) \in \vec{I} \times \vec{A}(D)$ the spatio-temporal components $C' \in \mathcal{C}$ generated by an extended spatio-temporal function, but not yet removed until that time step. Furthermore, the generated component stream captures the position $t \in T$ of each generated spatio-temporal component C' throughout its lifetime.

Definition 4.27 (Generated component streams) *Let $(V, 0_V, \bowtie, \sqcup, T)$ be a transformable collision space, $\mathcal{C}$ be the set of spatio-temporal components over $(V, 0_V, \bowtie, \sqcup, T)$, C be an extended spatio-temporal function over I, O, D, P, M, B, Y, X and $(V, 0_V, \bowtie, \sqcup, T)$, $(i, a) \in \vec{I} \times \vec{A}(D)$ be an input stream, and $(o, d, p, m, b, y, x) \in C(i, a)$ be an output stream. We define the stream $G \in \mathbb{P}(T \times \mathcal{C})^\infty$ such that the first stream element $G(0)$ is the empty set, i.e. $G(0) = \emptyset$, and for each computation step $n \in \mathbb{N}$ and generated component $(t, C') \in G(n+1)$ with transformation $t \in T$ and spatio-temporal function/component $C' \in \mathcal{C}$ as well as behavior $(o', d', p', m') \in C'(i', a')$ of component C' the component C' existed in the previous computation step n, i.e.*

$$\exists t' \in T : (t', C') \in G(n),$$

or the component C' entered in the previous computation step n, i.e.

$$\exists \upsilon \in Y : y(\upsilon)(n) = (t', C'),$$

and the component C' did not exit in the current computation step $n+1$, i.e.

$$\nexists \chi \in X : \bigvee_{\pi' \in P'} x(\chi)(n+1)(t(p'(\pi')(n+1))).$$

We call G the generated component stream over (i, a) and (o, d, p, m, b, y, x).

Based on the concept of generated component streams the properties of spatio-temporal functions can be translated to extended spatio-temporal functions. The first derived concept are *collision sensing extended spatio-temporal functions*, which require that the detectors $\delta \in D$ of extended spatio-temporal functions C and the detectors $\delta' \in D'$ of generated components C' with $(t, C') \in G(n)$, transformation $t \in T$ and time point $n \in \mathbb{N}$ are activated based on collisions between the respective detector volume and part volumes of the extended spatio-temporal function or some generated component.

Definition 4.28 (Collision sensing extended spatio-temporal functions) *Let $(V, 0_V, \bowtie, \sqcup, T)$ be a transformable collision space and C be an extended spatio-temporal component over I, O, D, P, M, B, Y, X and $(V, 0_V, \bowtie, \sqcup, T)$ such that for each stimulus $(i, a) \in \vec{I} \times \vec{A}(D)$ and reaction $(o, d, p, m, b, y, x) \in C(i, a)$, generated component stream G over (i, a) and (o, d, p, m, b, y, x), as well as time point $n \in \mathbb{N}$ and detector $\delta \in D$:*

$$a(\delta)(n) \Leftrightarrow \bigvee_{\pi \in P} p(\pi)(n) \bowtie d(\delta)(n) \vee \bigvee_{(t, C') \in G(n)} \bigvee_{\pi' \in P'} t(p'(\pi')(n)) \bowtie d(\delta)(n)$$

and for each generated component $(t, C') \in G(n)$ *over* I', O', D', P', M'*, stimulus* $(i', a') \in \vec{I}' \times \vec{A}(D')$*, reaction* $(o', d', p', m') \in C'(i', a')$*, as well as detector* $\delta' \in D'$*:*

$$a'(\delta')(n) \Leftrightarrow \bigvee_{\pi \in P} p(\pi) \bowtie t(d'(\delta')(n)) \vee \bigvee_{(t', C'') \in G(n)} \bigvee_{\pi'' \in P''} t'(p''(\pi'')(n)) \bowtie t(d'(\delta')(n))$$

We call C *a collision sensing extended spatio-temporal function.*

Note that in contrast to collision sensing spatio-temporal functions (see Section 4.2.2), collision sensing extended spatio-temporal functions require detector activations $a(\delta)(n)$ of detectors $\delta \in D$ and detector activations $a'(\delta')(n)$ of detectors $\delta' \in D'$ of generated component $(t, C') \in G(n)$ with transformation $t \in T$ at time point $n \in \mathbb{N}$ to be caused by collisions between the respective detector volumes and part volumes of the extended spatio-temporal function or some generated component *only*. The reason for this limitation is that the STEM theory assumes a closed world for simplicity [Hum11]. However, note that this closed world assumption also limits the composability of extended spatio-temporal functions, which is why the assumption is dropped in the following chapter.

Then, the question arises how generated components $(t, C') \in G(n)$ at computation step $n \in \mathbb{N}$ with transformation $t \in T$ and underlying spatio-temporal function/component $C' \in \mathcal{C}$ as well as behavior $(o', d', p', m') \in C'(i', a')$ of component C' interact with the extended spatio-temporal function C. This interaction is captured using the concept of *channel binding extended spatio-temporal functions*, which requires that the input channels I and output channels O of the extended spatio-temporal function C and the channels I', O' of the generated components (t, C') are connected based on the bindings B of the extended spatio-temporal function C. Consequently, input and output channels of both components can be either *bound* or *not bound* depending on the binding histories of the extended spatio-temporal function C.

Definition 4.29 (Channel binding extended spatio-temporal functions) *Let* $(V, 0_V, \bowtie, \sqcup, T)$ *be a transformable collision space,* $\mathcal{S}$ *be the set of all spatial selection predicates over* V*, and* C *be an extended spatio-temporal function over* I*,* O*,* D*,* P*,* M*,* B*,* Y*,* X *and* $(V, 0_V, \bowtie, \sqcup, T)$ *such that for each stimulus* $(i, a) \in \vec{I} \times \vec{A}(D)$ *and reaction* $(o, d, p, m, b, y, x) \in C(i, a)$ *with generated component stream* G *over* (i, a) *and* (o, d, p, m, b, y, x)*, computation step* $n \in \mathbb{N}$*, as well as input channel* $\iota \in I$ *either the input channel* ι *of function* C *is bound, i.e. there exists a generated component* $(t, C') \in G(n)$ *with,* C' *over* I', O', D', P', M'*, stimulus* $(i', a') \in \vec{I}' \times \vec{A}(D')$*, and reaction* $(o', d', p', m') \in C'(i', a')$ *such that* $i(\iota)(n) = o'(\iota)(n)$ *and*

$$\iota \in O' \wedge \exists \beta \in B, \sigma \in \mathcal{S} : b(\beta)(n) = (\sigma, \iota) \wedge \bigvee_{\pi' \in P'} \sigma(t(p'(\pi')(n))),$$

or the input channel ι *of function* C *is not bound, i.e.* $i(\iota)(n) = \bot$ *and for each generated component* $(t, C') \in G(n)$ *with* C' *over* I', O', D', P', M'*, stimulus* $(i', a') \in \vec{I}' \times \vec{A}(D')$*, and reaction* $(o', d', p', m') \in C'(i', a')$*:*

$$\iota \notin O' \vee \nexists \beta \in B, \sigma \in \mathcal{S} : b(\beta)(n) = (\sigma, \iota) \wedge \bigvee_{\pi' \in P'} \sigma(t(p'(\pi')(n))),$$

and for each generated component $(t, C') \in G(n)$ *with* C' *over* I', O', D', P', M'*, stimulus* $(i', a') \in \vec{I}' \times \vec{A}(D')$*, reaction* $(o', d', p', m') \in C'(i', a')$*, and input channel* $\iota' \in I'$ *either the input channel* ι' *of generated component* (t, C') *is bound, i.e.* $i'(\iota')(n) = o(\iota')(n)$ *and*

$$\iota' \in O \wedge \exists \beta \in B, \sigma \in \mathcal{S} : b(\beta)(n) = (\sigma, \iota') \wedge \bigvee_{\pi' \in P'} \sigma(t(p'(\pi')(n))),$$

or the input channel ι' *of component* (t, C') *is not bound, i.e.* $i'(\iota')(n) = \bot$ *and*

$$\iota' \notin O \vee \nexists \beta \in B, \sigma \in \mathcal{S} : b(\beta)(n) = (\sigma, \iota') \wedge \bigvee_{\pi' \in P'} \sigma(t(p'(\pi')(n))).$$

We call C *a channel binding extended spatio-temporal function.*

Note that, again, channel binding extended spatio-temporal functions obey the closed world assumption because the empty message $\bot \notin M$ is forced to unbound input channels $\iota \in I$ of the extended spatio-temporal function C as well as input channels $\iota' \in I'$ of generated components $(t, C') \in G(n)$. Consequently, the input channels cannot be written by the environment of the extended spatio-temporal function C and the generated components (t, C'). As mentioned previously, in the following chapter the closed world assumption is dropped to support the composability of extended spatio-temporal functions and components.

Finally, the question is answered how generated components $(t, C') \in G(n)$ with time point $n \in \mathbb{N}$ can be moved around the transformable collision space $(V, 0_V, \bowtie, \sqcup, T)$. This effect is achieved through binding movers $\mu \in M$ of extended spatio-temporal functions C to generated components (t, C') yielding the concept of *mover binding generated component streams*. Note that the mover binding property applies to the definition of the generated component stream $G \in \mathbb{P}(T \times \mathcal{C})^\infty$ (see Definition 4.27). Furthermore, similar to channel bindings, the mover bindings are based on spatial selection predicates $\sigma \in \mathcal{S}$ for selectively binding generated components (t, C') based on their spatial extent.

Definition 4.30 (Mover binding generated component streams) *Let* C *be an extended spatio-temporal function over* I*,* O*,* D*,* P*,* M*,* B*,* Y*,* X*,* $(i, a) \in \vec{I} \times \vec{A}(D)$ *be a*

stimulus, $(o, d, p, m, b, y, x) \in C(i, a)$ *be a reaction, and* G *be the generated component stream over* (i, a) *and* (o, d, p, m, b, y, x) *such that for each computation step* $n \in \mathbb{N}$ *and generated component* $(t, C') \in G(n)$*, which also exists in the subsequent computation step* $n + 1$*, i.e.* $(t', C') \in G(n + 1)$*, either the position and orientation of the generated component* (t, C') *has not changed, i.e.* $t = t'$*, because there is no mover binding, i.e.*

$$\nexists \beta \in B, \mu \in M, \sigma \in \mathcal{S} : b(\beta)(n) = (\sigma, \mu) \wedge \bigvee_{\pi' \in P'} \sigma(t(p'(\pi')(n)))$$

or the position and orientation of the generated component (t, C') *has changed, i.e.* $t' = m(\mu)(n) \circ t$*, due to a mover binding, i.e.*

$$\exists \beta \in B, \mu \in M, \sigma \in \mathcal{S} : b(\beta)(n) = (\sigma, \mu) \wedge \bigvee_{\pi' \in P'} \sigma(t(p'(\pi')(n))).$$

We call G *a mover binding generated component stream.*

Consequently, a mover binding generated component stream makes sure that generated components $(t, C') \in G(n)$ with computation step $n \in \mathbb{N}$ remain in the subsequent computation step $n + 1$ where they were in the previous computation step n or mover transformations $m(\mu)(n) \in T$ of movers $\mu \in M$ of the extended spatio-temporal function C are applied to them. Based on the concepts of collision sensing and channel binding extended spatio-temporal functions as well as mover binding generated component streams the concept of extended spatio-temporal components can be introduced.

Definition 4.31 (Extended spatio-temporal components) *Let* C *be a collision sensing and channel binding extended spatio-temporal function such that for each stimulus* $(i, a) \in \vec{I} \times \vec{A}(D)$ *and reaction* $(o, d, p, m, b, y, x) \in C(i, a)$ *the generated component stream* G *over* (i, a) *and* (o, d, p, m, b, y, x) *is mover binding. We call* C *an extended spatio-temporal component.*

Similar to components in FOCUS as well as the original spatio-temporal components in the STEM theory, extended spatio-temporal components can be deterministic or nondeterministic. Furthermore, extended spatio-temporal components can be weakly or strongly causal depending on the reaction delay. The corresponding definitions can be translated easily, which is why they are omitted here. For more information the interested reader can have a look at the original work [Hum11]. However, one interesting property of extended spatio-temporal components is whether they can be operated without collisions or not. As mentioned previously, this property is particularly interesting

for the industrial context of this doctoral thesis because collisions might cause severe damage to the system, the business, and / or humans involved. For dealing with collision free operation, the concept of *collision free extended spatio-temporal components* is introduced. In addition to collision free spatio-temporal components, the extended case has to take into account collisions between part volumes $p(\pi)(n) \in V$ of the extended spatio-temporal component C and part volumes $p'(\pi')(n) \in V$ of generated components $(t, C') \in G(n)$ at time point $n \in \mathbb{N}$ as well as between part volumes of two different generated components.

Definition 4.32 (Collision free extended spatio-temporal components) *Let $(V, 0_V, \bowtie, \sqcup, T)$ be a transformable collision space and C be an extended spatio-temporal component over I, O, D, P, M, B, Y, X and $(V, 0_V, \bowtie, \sqcup, T)$ such that for each stimulus $(i, a) \in \vec{I} \times \vec{A}(D)$ and reaction $(o, d, p, m, b, y, x) \in C(i, a)$ with mover binding generated component stream G over (i, a) and (o, d, p, m, b, y, x), computation step $n \in \mathbb{N}$, and generated component $(t, C') \in G(n)$ with C' over I', O', D', P', M', stimulus $(i', a') \in \vec{I}' \times \vec{A}(D')$ and reaction $(o', d', p', m') \in C'(i', a')$ the component C' is a collision free spatio-temporal component (see Definition 4.22) and the parts of component C and generated component (t, C') do not collide, i.e.*

$$\neg \bigvee_{\pi \in P} \bigvee_{\pi' \in P'} p(\pi)(n) \bowtie t(p'(\pi')(n))$$

and for each other generated component $(t', C'') \in G(n)$ with $(t', C'') \neq (t, C')$ and C'' over I'', O'', D'', P'', M'', stimulus $(i'', a'') \in \vec{I}'' \times \vec{A}(D'')$, and reaction $(o'', d'', p'', m'') \in C''(i'', a'')$ the parts of generated component (t, C') and generated component (t', C'') do not collide, i.e.

$$\neg \bigvee_{\pi' \in P'} \bigvee_{\pi'' \in P''} t(p'(\pi')(n)) \bowtie t'(p''(\pi'')(n)).$$

We call C a collision free extended spatio-temporal component.

At last, note that the original STEM theory [Hum11] does not include a composition operator for extended spatio-temporal components partly because of the assumptions and limitations mentioned previously. Furthermore, the theory leaves some questions open such as what happens when multiple mover transformations $m(\mu_1)(n), m(\mu_2)(n) \in T$ with movers $\mu_1, \mu_2 \in M$ of the extended spatio-temporal component C are bound to the same generated component $(t, C') \in G(n)$ at time point $n \in \mathbb{N}$ or when multiple output channels are bound to the same input channel. Finally, the theory only supports generating spatio-temporal components (see Section 4.2.2) instead of extended spatio-temporal components. Consequently, generated components cannot define their own bindings, entries, and exits, which might be desirable in practical applications.

4.3 Summary and outlook

This chapter introduced the theoretical foundations for the proposed modeling technique. In particular, the FOCUS theory for the design of software systems (see Section 4.1) has been described. Then, the STEM extension for modeling spatio-temporal systems (see Section 4.2) has been explained. The next chapter proposes an extended and revised modeling technique based on FOCUS and STEM, which also captures knowledge about customer requirements, manufacturing processes, test cases, discipline-specific interfaces, and component reuse. Furthermore, the revised theory provides an extended composition operator as well as custom bindings, entries, and exists of generated components. At the same time, a concrete data model is proposed for capturing the conceptual design knowledge. In particular, the concrete data model is required for deriving the *syntactic* quality issues and the revised theory is needed for deriving the *semantic* quality issues in Chapter 6.

5 Modeling technique

In this chapter a novel technique is introduced for describing cyber-physical manufacturing systems during conceptual design. In particular, the technique integrates seamlessly the aspects postulated by the test-driven design method introduced in Chapter 3. These aspects include the customer requirements, manufacturing processes, and test cases during the preparation phase as well as the component architecture, material, energy, and signal flow behaviors, and physical parts of mechanical and electrical components during the implementation phase. To achieve this goal, the FOCUS (see Section 4.1) and STEM (see Section 4.2) *formalisms* presented in the previous chapter are revised and extended. At the same time, a *concrete data model* is introduced for capturing the static design knowledge in a machine-readable format (while a concrete data model for the execution/computation semantics is not provided). In particular, the classes of design objects, their attributes, and their relations to other classes of design objects are described using the UML [BJR+96] class diagram notation. Hereby, *composition* (i.e. parent-child) and *aggregation* relations between design objects are distinguished. Note that composition relations establish a hierarchy among the design objects, while aggregation relations introduce cross-references within this hierarchy. The data model is used for deriving the *syntactic* quality issues of the modeling technique and the revised/extended formalism for deriving the *semantic* quality issues in the following chapter.

Subsequently, three groups of modeling concepts are distinguished, namely *basic concepts* (see Section 5.1), *revised concepts* (see Section 5.2), and *added concepts* (see Section 5.3). Basic concepts exist in the original FOCUS and STEM theories in similar forms, have undergone at most minor revisions, and are required for explaining the remaining concepts. In contrast, revised concepts exist in the original theories, but have undergone a major revision to fit the particular needs. Finally, added concepts do not exist in the original theories yet (particularly in the STEM theory), but have been introduced to cover fully the needs of the test-driven design method proposed in Chapter 3.

5.1 Basic concepts

The most fundamental concepts of the modeling technique are *observations* (see Section 5.1.1), *executables* (see Section 5.1.2), and *expressions* (see Section 5.1.3) as well

as *volumes* (see Section 5.1.4) and *transforms* (see Section 5.1.5). In essence, observations, executables, and expressions provide the underlying model of computation, while volumes and transforms provide the model of shape and motion. In the following, each concept is described in more detail.

5.1.1 Observations

Observations provide information about the system state over time. Formally, observations correspond to the concept of channels with the sets of channel labels L, type labels T, and messages M as well as the type mapping $M' : L \to T$ and message mapping $M'' : T \to \mathbb{P}(M)$ in the FOCUS theory (see Section 4.1.2). Note that the original concepts cannot be reused directly without making modifications. Subsequently, Figure 5.1 describes the concrete data model for representing the observation concept.

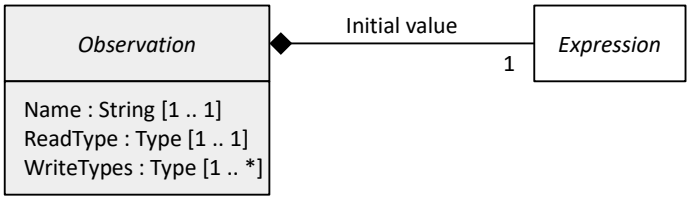


Figure 5.1: Abstract observation class.

The abstract observation class defines a *name*, a *read type*, at least one *write type*, and an *initial value expression*. The name corresponds to the elements $\lambda \in L$ of the set of channel labels L. The read type corresponds to the type mapping M' and specifies the type of messages $M'(\lambda) \in T$, which can be read from corresponding observation streams. Instead, the write types specify the types of messages which can be written to these observation streams. Note that the concrete observation classes convert the messages implicitly in case the read and write types are different, which is supported also in the FOCUS theory by means of the *map operator* [BS01]. In the following, *boolean* (i.e. $\mathbb{B}$), *number* (i.e. $\mathbb{R}$), *matrix* (i.e. $\mathbb{R}^{4\times3}$), and *set* types are used, from which the message mapping M'' can be derived easily. In particular, 4×3 matrices represent volume transformations and, hence, motion in 3-dimensional Euclidean space with Cartesian coordinates. Finally, the initial value expression of observations $\lambda \in L$ determines the first value $l(\lambda)(0) \in M''(M'(\lambda))$ in respective observation streams $l \in \vec{L}$. Note that the initial value expression has to be type-compatible with the write types of the corresponding observation as detailed in Chapter 6. The proposed concept of expressions, in turn, is explained in Section 5.1.3.

5.1.2 Executables

Based on the concept of observations (see Section 5.1.1) the concept of executables is defined. Executables are inspired by the concept of state transition systems (T, s_0, v_0) with input channel labels I, output channel labels O, state labels S, variable labels V, state transition function $T : S \times \bar{I} \times \bar{V} \to \mathbb{P}(S \times \bar{O} \times \bar{V})$, initial state $s_0 \in S$, and initial variable assignment $v_0 \in \bar{V}$ (see Section 4.1.4). However, the original state transition function only takes the current state, variable assignment, and input channel assignment into account. Here, the finite stream of states, variable assignments, and input observation assignments up to the current time point are used instead similar to the test automata described in [KT04]. These modifications allow one to describe, e.g., the duration of states, arbitrary signal delays, or integrals over time more easily. Consequently, the state transition function changes to $T' : (S \times \bar{I} \times \bar{V})^* \to \mathbb{P}(S \times \bar{O} \times \bar{V})$.

Then, the computation of the revised state transition system (T', s_0, v_0) (see original Definition 4.15) is defined for an input channel assignment stream $i \in \bar{I}^\infty$ as the state stream $s \in \bar{S}^\infty$ with initial state $s(0) = s_0$, the variable assignment stream $v \in \bar{V}^\infty$ with initial variable assignment $v(0) = v_0$, and the output channel assignment stream $o \in \bar{O}^\infty$ such that

$$\forall n \in \mathbb{N} : (s(n+1), o(n), v(n+1)) \in T'(s \downarrow n, i \downarrow n, v \downarrow n).$$

Note that the original definition of state transition system computations over T (see Section 4.1.4) is contained by the revised definition of state transition system computations over T'. In particular, the original definition only used the latest states $s(n) \in S$, input channel assignments $i(n) \in \bar{I}$, and variable assignments $v(n) \in \bar{V}$ at computation step $n \in \mathbb{N}$. In contrast, the revised definition uses the entire stream of states $s \downarrow n \in S^n$, input channel assignments $i \downarrow n \in \bar{I}^n$, and variable assignments $v \downarrow n \in \bar{V}^n$ up to the current computation step $n \in \mathbb{N}$ for deriving the follow-up states as well as output channel and variable assignments.

Furthermore, causality can be defined for the revised state machine formalism at different computation steps. The state machine (T', s_0, v_0) is called weakly causal with respect to inputs $I_1 \subseteq I$ and strongly causal with respect to inputs $I_2 \subseteq I$ with $I_1 \cap I_2 = \emptyset$ and $I_1 \cup I_2 = I$ at computation step $k \in \mathbb{N}$, if for each pair of input streams $i_1, i_2 \in \bar{I}^\infty$ with matching input channel assignment stream prefixes

$$\bigwedge_{\iota \in I_1} \bigwedge_{0 \leq k' \leq k} i_1(k')(\iota) = i_2(k')(\iota) \wedge \bigwedge_{\iota \in I_2} \bigwedge_{0 \leq k' < k} i_1(k')(\iota) = i_2(k')(\iota)$$

the possible set of follow-up streams of states, variable assignments, and output channel assignments for the first input channel assignment stream i_1, i.e.

$$\{(s_1 \downarrow (k+1), o_1 \downarrow k, v_1 \downarrow (k+1)) : (i_1, s_1, v_1, o_1) \text{ is a computation of } (T', s_0, v_0)\}$$

equals to the possible set of follow-up streams of states, variables assignments, and output channel assignments for the second input channel assignment stream i_2, i.e.

$$\{(s_2 \downarrow (k+1), o_2 \downarrow k, v_2 \downarrow (k+1)) : (i_2, s_2, v_2, o_2) \text{ is a computation of } (T', s_0, v_0)\}.$$

Note that revised state transition systems (T', s_0, v_0) can be both weakly and strongly causal for the same input channel assignment stream $i \in \bar{I}^{\infty}$ at computation step $k \in \mathbb{N}$, but with respect to different sets of input channel labels I_1 and I_2 with $I_1 \cap I_2 = \emptyset$ and $I_1 \cup I_2 = I$. Furthermore, note that the revised state transition system (T', s_0, v_0) might be weakly and strongly causal with respect to different sets of input channel labels at different computation steps $k' \in \mathbb{N}$ with $k' \neq k$. In contrast, the original state transition systems (T, s_0, v_0) are strongly causal with respect to all input channels $i \in I$ and cannot change their causality at different computation steps $k, k' \in \mathbb{N}$ with $k \neq k'$. Consequently, the revised formalism subsumes the formalism from Section 4.1.4.

Finally, the question arises how to represent revised state transition systems (T', s_0, v_0) in a machine-readable format such that they can be executed and analyzed automatically using computer programs. As noted previously, a representation was developed, which is based on the concept of observations introduced in the previous section. Subsequently, the concrete data model is introduced for representing executables in Figure 5.2.

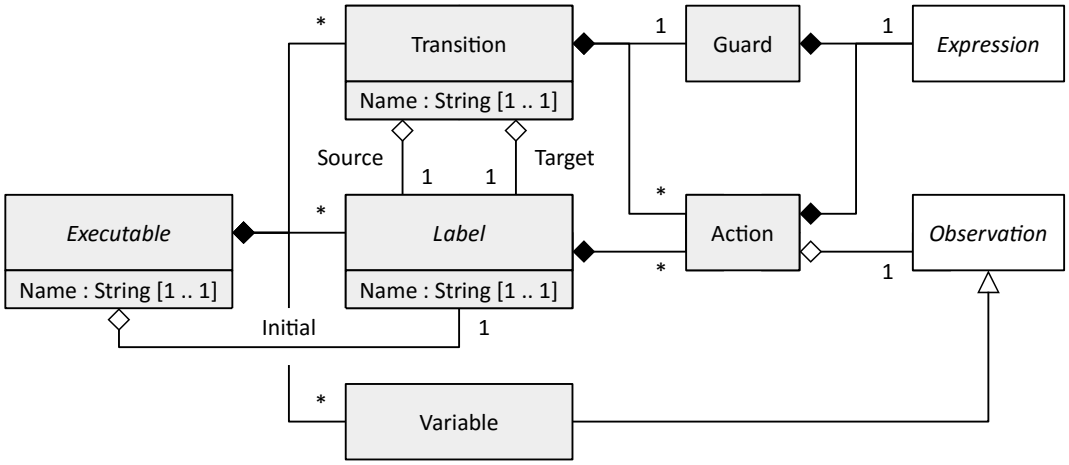


Figure 5.2: Abstract executable class and associated classes.

The abstract executable class comprises a *name*, a set of *(state) labels* with one *initial (state) label*, a set of *transitions*, and a set of *variables*. Variables, in turn, are derived from the observation concept and correspond to the variable labels V. In contrast, (state)

labels define a *name* and correspond to the state labels S. Finally, transitions comprise a *name* as well as a *source* and a *target state*. Note that transitions correspond to the state transition function T'. Furthermore, transitions contain a *guard*, which, in turn, contains one *boolean expression* (see Section 5.1.3). The guard expression tells whether the transition is enabled in each computation step $k \in \mathbb{N}$. Note that multiple transitions might be enabled at the same time, which represents one cause for non-determinism in the model (see Section 6.2.2). Moreover, transitions contain *actions* which, in turn, reference an *output observation* and an *expression*. If the transition is enabled and selected in the a computation step $k \in \mathbb{N}$, its action expressions define the new values $o(\omega)(k) \in M''(M'(\omega))$ of the referenced output observations $\omega \in O$. Finally, (state) labels themselves also may contain *actions*. If no transition is enabled in a computation step, the expressions of these actions define the new values of the referenced output observation instead. Finally, note that action expressions must be type-compatible with the corresponding observations as detailed in Section 6.1.2.

5.1.3 Expressions

Based on the observation concept (see Section 5.1.1) and the executable concept (see Section 5.1.2) the concept of expressions is derived. Expressions map a finite stream of (state) labels, input observation assignments, and variable assignments to one single output value (e.g. a sum over time). Consequently, expressions are functions $E : (S \times \bar{I} \times \bar{V})^* \rightarrow D$ over state labels S, input observation labels I, variable labels V, and target domain $D \subseteq M$, where D can be any of *boolean*, *number*, *matrix*, *string*, and *set*. Figure 5.3 shows the concrete data model for representing expressions in the specification.

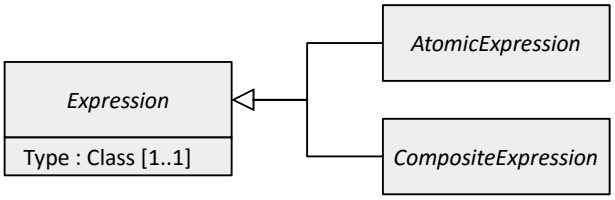


Figure 5.3: Abstract expression class and subclasses.

The abstract expression base class only defines the *return type* of the expression, which corresponds to the target domain D. Then, two abstract subclasses are distinguished, namely *atomic expressions* and *composite expressions*. The difference between the two classes is that composite expression contain *argument expressions*, while atomic expressions are self-contained. In the following, each subclass is explained in detail. Thereby,

the expression semantics is specified for data stream $ds_n \in (S \times \bar{I} \times \bar{V})^n$ with data stream length $n \in \mathbb{N}$ and data stream elements $ds_n(m) = (s_m, i_m, v_m)$ with $m \in \mathbb{N}$ and $m \leq n$, as well as state accessor $s(ds_n, m) = s_m$, input channel assignment accessor $i(ds_n, m) = i_m$, and variable assignment accessor $v(ds_n, m) = v_m$.

Atomic expressions

In principle, a wide variety of atomic expressions are supported. Here, only the most important expressions are presented, which also are used frequently in the industry-close showcase for evaluation purposes (see Chapter 8). Figure 5.4 provides an overview of the data model for the most important atomic expressions.

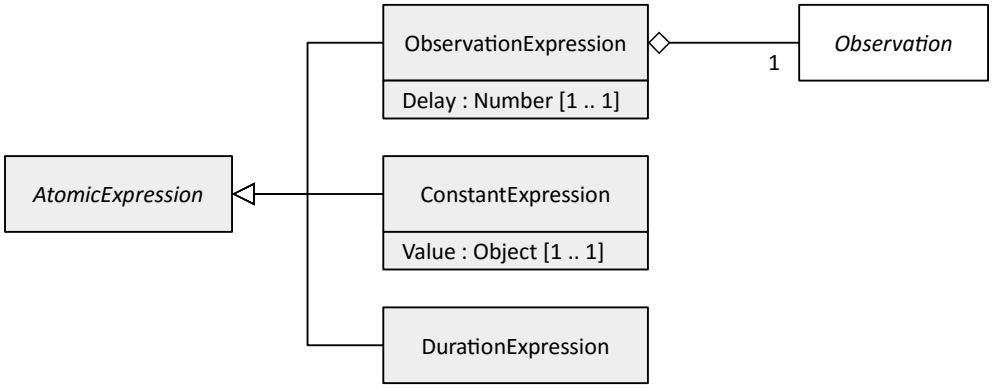


Figure 5.4: Abstract atomic expression class and subclasses.

The most important atomic expressions are the *observation expression*, the *constant expression*, and the *duration expression*. The observation expression allows one to read an observation $\lambda \in I \cup V$ with delay $d \in \mathbb{N}$. The semantics of the observation expression is given by the function

$$E^{\lambda,d}_{Observation}(ds_n) = \begin{cases} i(ds_n, n-d)(\lambda) & \text{if } n-d \geq 0 \wedge \lambda \in I \\ v(ds_n, n-d)(\lambda) & \text{if } n-d \geq 0 \wedge \lambda \in V \\ \bot & \text{if } n-d < 0 \end{cases}.$$

In contrast, the constant expression returns for each data stream ds_n a constant value $c \in D$, where $D \subseteq M$ represents the target domain of the expression. Consequently, the semantics of the constant expression is given by the function

$$E^{c}_{Constant}(ds_n) = c.$$

Finally, the duration expression allows one to determine the number of discrete time steps, for which an executable resides in the same state. The semantics of the duration expression is given by the function

$$E_{Duration}(ds_n) = Count(ds_n, n, n).$$

Hereby, the function $Count : (S \times \bar{I} \times \bar{V})^* \times \mathbb{Z} \times \mathbb{N} \to \mathbb{N}$ counts the repetitive occurrences of the state label $s(ds_n, n) \in S$ at the end of the data stream. Finally, for counting the repetitive occurrences of the state labels the following algorithm is used:

$$Count(ds_n, k, m) = \begin{cases} 1 + Count(ds_n, k-1, m) & \text{if } s(ds_n, k) = s(ds_n, m) \wedge k \geq 0 \\ 0 & \text{if } s(ds_n, k) \neq s(ds_n, m) \vee k < 0 \end{cases}.$$

Note that especially the observation expression and the duration expression require the stream of previous state labels, input channel assignments, and variable assignments to describe their functional semantics.

Composite expressions

Similarly, a wide variety of composite expressions are provided. Again, due to space limitations only the most important expressions are presented, which also are used frequently in the industry-close showcase from Chapter 8. Figure 5.5 provides an overview of the data model for the most important composite expressions.

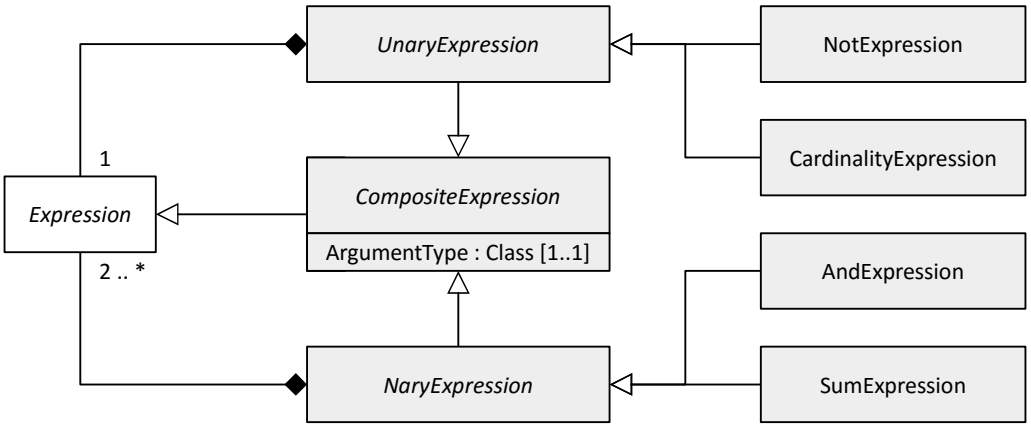


Figure 5.5: Abstract composite expression class and subclasses.

Generally, *unary* and *nary (composite) expressions* are distinguished. Unary expressions contain a single argument expression of a defined argument type, while nary expressions require two or more argument expressions of the defined argument type. The most important unary expressions are the *not expression* and the *cardinality expression*. In contrast, the most important nary expressions are the *and expression* and the *sum expression*. The not expression expects a boolean argument expression and negates its outcome. The semantics of the not expression is given by the function

$$E_{Not}(s) = \neg E_{Argument}(s).$$

In contrast, the cardinality expression expects a set argument expression and returns the respective set cardinality. The semantics is given by the function

$$E_{Cardinality}(s) = |E_{Argument}(s)|.$$

Then, the and expression calculates the logical conjunction of the $m \in \mathbb{N}$ boolean argument expressions. The semantics of the expression is given by the function

$$E_{And}(s) = \bigwedge_{1 \leq k \leq m} E_{Argument_k}(s).$$

Finally, the sum expression calculates the arithmetic sum of the $m \in \mathbb{N}$ number argument expressions. The semantics of the expression is given by the function

$$E_{Sum}(s) = \sum_{1 \leq k \leq m} E_{Argument_k}(s).$$

Similar expressions are provided for other boolean, arithmetic, matrix and set operations such as the logical disjunction, the arithmetic difference, the matrix product, or the set union. Note that the argument expressions might require a stream of states, input observation assignments, and variable assignments respectively.

5.1.4 Volumes

Subsequently, the concept of volumes is introduced. As mentioned in Section 4.2.1, a three-dimensional Euclidean transformable collision space $(V, 0_V, \bowtie, \sqcup, T)$ with Cartesian coordinate system is used. Consequently, volumes $v \in V$ correspond to a set $v \subseteq \mathbb{R}^3$ of elements (i.e. points) in the Euclidean space. Then, the empty volume 0_V corresponds to the empty set $\emptyset \subseteq \mathbb{R}^3$, the volume collision operator $\bowtie$ corresponds to the set intersection operator $\cap$, and the volume union operator $\sqcup$ corresponds to the set union operator $\cup$. Finally, Figure 5.6 shows the concrete data model behind the volume concept.

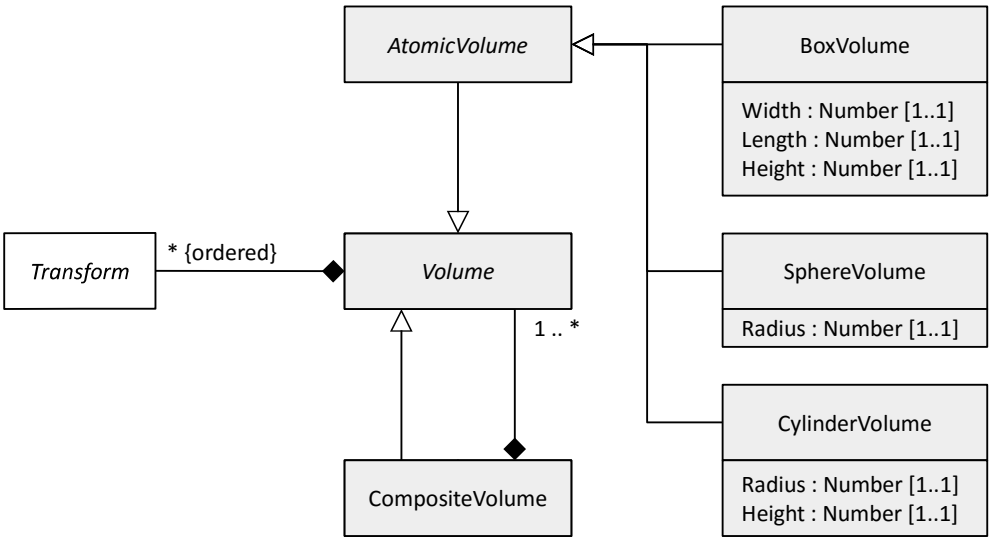


Figure 5.6: Abstract volume class and subclasses.

The abstract volume base class contains an ordered list of transforms $t = t_1 \circ ... \circ t_n \in T$ with length $n \in \mathbb{N}$ (see Section 5.1.5), which represents the volume position and orientation in the initial state. Then, *atomic* and *composite volumes* are distinguished. Atomic volumes can be *box volumes*, *sphere volumes*, or *cylinder volumes*. Box volumes $t(v_b) \in V$ define a width $w \in \mathbb{R}_0^+$, height $h \in \mathbb{R}_0^+$, and length $l \in \mathbb{R}_0^+$ such that

$$v_b = \{(x, y, z) \in \mathbb{R}^3 \mid -w/2 \leq x \leq w/2 \wedge -h/2 \leq y \leq h/2 \wedge -l/2 \leq z \leq l/2\}.$$

In contrast, sphere volumes $t(v_s) \in V$ define a radius $r \in \mathbb{R}_0^+$ such that

$$v_s = \{(x, y, z) \in \mathbb{R}^3 \mid \sqrt{x^2 + y^2 + z^2} \leq r\}.$$

Then, cylinder volumes $t(v_c) \in V$ define a radius $r \in \mathbb{R}_0^+$ and a height $h \in \mathbb{R}_0^+$ such that

$$v_c = \{(x, y, z) \in \mathbb{R}^3 \mid \sqrt{x^2 + z^2} \leq r \wedge -h/2 \leq y \leq h/2\}.$$

Moreover, composite volumes $t(v_\cup) \in \mathbb{R}^3$ define a set $V' \subseteq V$ of child volumes such that

$$v_\cup = \bigcup_{v' \in V'} v'$$

Note that composite volumes only support the unified child volumes. Consequently, the collision of composite volumes can be calculated easily based on the pairwise collision

of the child volumes. However, in practice one might want to work also with volume intersection and difference as known from constructive solid geometry [RV77], which complicates the calculation of the collision relation and is omitted in this doctoral thesis.

5.1.5 Transforms

Based on the volume concept (see Section 5.1.4) the concept of transforms (or motion) is introduced. As mentioned in the previous section, the Euclidean transformable collision space $(V, 0_V, \bowtie, \sqcup, T)$ (see Section 4.2.1) with Cartesian coordinate system is used such that each volume $v \in V$ represents a set of elements (i.e. points) $v \subseteq \mathbb{R}^3$. Consequently, transformations $t \in T$ are functions $t : \mathbb{P}(\mathbb{R}^3) \to \mathbb{P}(\mathbb{R}^3)$ that map point sets onto each other. Figure 5.7 shows the underlying concrete data model.

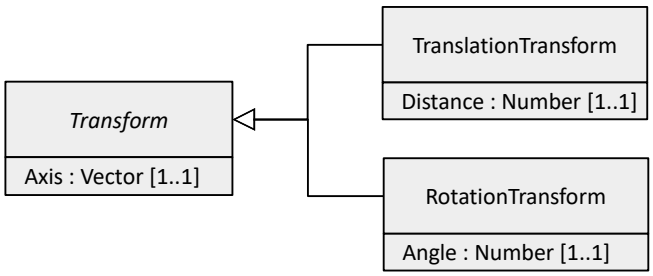


Figure 5.7: Abstract transform class and subclasses.

The abstract transform base class defines a three-dimensional vector *axis* $a \in \mathbb{R}^3$ with $|a| = 1$ and the operator $|\cdot| : \mathbb{R}^3 \to \mathbb{R}$ representing the Euclidean norm. The interpretation of the axis depends on the concrete transform type. Then, for simplicity only two types of concrete transforms are considered, namely *translation* and *rotation transforms*. Translation transforms $t_t \in T$ additionally define a *distance* $d \in \mathbb{R}$ such that

$$t_t(v) = \{p \in \mathbb{R}^3 \mid \exists p' \in v : p' = p - d * a\}$$

with $* : \mathbb{R} \times \mathbb{R}^3 \to \mathbb{R}^3$ representing the scalar product and $- : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$ representing the vector difference operators. In contrast, rotation transforms $t_r \in T$ additionally define an *angle* $\alpha \in \mathbb{R}$ such that

$$t_r(v) = \{p \in \mathbb{R}^3 \mid \exists p' \in v : p' = (a \cdot p) * a + \cos(\alpha) * (a \times p) \times a + \sin(\alpha) * (a \times p)\}$$

with $* : \mathbb{R} \times \mathbb{R}^3 \to \mathbb{R}^3$ representing the scalar product, $\cdot : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ representing the dot product, $\times : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$ representing the cross product, and $+ : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$ representing the vector sum operators.

5.2 Revised concepts

Following the basic concepts, revised concepts are described that already existed in the STEM theory (see Section 4.2), but have been adapted to meet the particular needs. First, *revised spatio-temporal components* are introduced in Section 5.2.1, before turning to *ports* in Section 5.2.2 and *channels* in Section 5.2.3. Finally, solid physical *parts* are explained in Section 5.2.4 and *behaviors* are introduced in Section 5.2.5. Where applicable the extensions and adaptations to the original theory are described.

5.2.1 Components

The revised component concept is inspired by the concept of extended spatio-temporal components $C : \vec{I} \times \vec{A}(D) \to \mathbb{P}(\vec{O} \times \vec{D} \times \vec{P} \times \vec{M} \times \vec{B} \times \vec{Y} \times \vec{X})$ over input channel histories $\vec{I}$, activation histories $\vec{A}(D) = \{D \to \mathbb{B}^\infty\}$, output channel histories $\vec{O}$, detector histories $\vec{D} = \{D \to V^\infty\}$, part histories $\vec{P} = \{P \to V^\infty\}$, mover histories $\vec{M} = \{M \to T^\infty\}$, binding histories $\vec{B} = \{B \to (\mathcal{S} \times (I \cup O \cup M))^\infty\}$, entry histories $\vec{Y} = \{Y \to (T \times \mathcal{C})^\infty\}$, exit histories $\vec{X} = \{X \to \mathcal{S}^\infty\}$, and transformable collision space $(V, 0_V, \bowtie, \sqcup, T)$ as well as the composition operator $\odot$ and the motion operator $\vdash_\mu$ (see Section 4.2). The original theory is modified to support customer requirement, manufacturing process, and test case (see Section 5.3) as well as the remaining specification activities more naturally:

- The original concept is revised such that the transfer of kinetic energy (i.e. volume transforms $t \in T$; see Section 5.1.5) can be achieved via classical FOCUS channels.
- Furthermore, the concept of detectors D is replaced with a more general concept of bindings B, which allows one to observe and identify each bound component.
- Then, means for modeling dynamic interactions between components of the static system architecture are provided through a more general binding concept B.
- Moreover, generated components $(t, C') \in G(n)$ with computation step $n \in \mathbb{N}$ are allowed to define their own bindings B', entries Y', and exits X'.
- Additionally, means are integrated for embedding components dynamically and, hence, forwarding kinetic energy applied to the embedder components.
- Finally, the collision relation $\bowtie$ is used instead of spatial selection predicates $\sigma \in \mathcal{S}$ to simplify the theory.

In the following, first the revised spatio-temporal component formalism is introduced, before the data model for capturing the underlying design information is explained. In

particular, the data model provides mean for reusing existing component specifications in different contexts (i.e. a construction kit with instantiation support).

Revised component formalism

The revised component formalism is based on a revised syntactic and semantic interface of spatio-temporal components. Similar to the introduction of spatio-temporal components (see Section 4.2.2) and extended spatio-temporal components (see Section 4.2.3), first *revised spatio-temporal functions* are introduced. Then, the definitions of *revised spatio-temporal components* as well as the *revised composition operator* are derived. Where necessary, additional properties and concepts are introduced.

Revised spatio-temporal functions Firstly, revised spatio-temporal functions C (see Definition 4.26 for extended spatio-temporal functions) are defined as mappings from input channel histories $\vec{I}$, kinetic energy input channel histories $\vec{I}_M = \{I_M \to T^\infty\}$, kinetic energy forwarding histories $\vec{I}_F = \{\mathcal{C} \to T^\infty\}$ with universe of revised spatio-temporal components $\mathcal{C}$ (definition coming later), and activation histories $\vec{A}(B \cup Y \cup X) = \{B \cup Y \cup X \to \mathbb{P}(\mathcal{C})^\infty\}$ to output channel histories $\vec{O}$, kinetic energy output channel histories $\vec{O}_M = \{O_M \to T^\infty\}$, part histories $\vec{P} = \{P \to V^\infty\}$, binding histories $\vec{B} = \{B \to (V \times \mathbb{P}(\mathcal{O} \times \mathcal{I}) \times \mathbb{B})^\infty\}$ with the universe of (kinetic energy) output channel labels $\mathcal{O}$ and the universe of (kinetic energy) input channel labels $\mathcal{I}$, entry histories $\vec{Y} = \{Y \to (T \times \mathcal{C})^\infty\}$, and exit histories $\vec{X} = \{X \to V^\infty\}$, i.e.

$$C : \vec{I} \times \vec{I}_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X) \to \mathbb{P}(\vec{O} \times \vec{O}_M \times \vec{P} \times \vec{B} \times \vec{Y} \times \vec{X}).$$

Note that the spatial selection predicates $\sigma \in \mathcal{S}$ have been removed from the binding histories $\vec{B}$ and exit histories $\vec{X}$. Instead, collision volumes $v \in V$ are used. Furthermore, note that the original activation histories $\vec{A}(D)$ have been replaced by the activation histories $\vec{A}(B \cup Y \cup X)$. The revised activation histories provide access to the components, which can be observed at each binding $\beta \in B$, entry $\upsilon \in Y$, and exit $\xi \in X$, instead of reporting collisions between parts $\pi \in P$ and detectors $\delta \in D$ only. Consequently, the concept of detectors is blended into a revised concept of bindings. Finally, note that the kinetic energy input channel and forwarding histories $\vec{I}_M$ and $\vec{I}_F$ have been added. The first allows one to transmit kinetic energy over classical FOCUS channels, while the second allows one to describe, e.g., airflow not requiring kinematic chains.

Zero kinetic energy Before describing the reaction of revised spatio-temporal functions C over $I, I_M, O, O_M, P, B, Y, X$ to arbitrary kinetic energy input channel and forwarding

histories $i_M \in \vec{I}_M$ and $i_F \in \vec{I}_F$, first the zero kinetic energy input channel and forwarding histories $i_{M,0} \in \vec{I}_M$ and $i_{F,0} \in \vec{I}_F$ are defined. Both histories are based on the identity transformation stream $t_0 \in T^\infty$ with

$$\forall n \in \mathbb{N}, v \in V : t_0(n)(v) = v.$$

At each computation step $n \in \mathbb{N}$ the identity transformation stream t_0 contains the identity transformation, which maps each volume $v \in V$ onto itself. Based on the identity transformation stream t_0 the zero kinetic energy input channel and forwarding histories $i_{M,0}$ and $i_{F,0}$ can be defined such that

$$\forall \iota \in I_M : i_{M,0}(\iota) = t_0 \text{ and } \forall C \in \mathcal{C} : i_{F,0}(C) = t_0.$$

Note that each kinetic energy input channel $\iota \in I_M$ and each component $C \in \mathcal{C}$ is mapped to the identity transformation stream t_0. Consequently, the zero kinetic energy input channel and forwarding histories $i_{M,0}$ and $i_{F,0}$ can be used to describe the component C at its default position and orientation.

Movable spatio-temporal functions Then, the reaction of revised spatio-temporal functions C to arbitrary kinetic energy stimuli is constrained (replacing the *pos* operator of the STEM theory introduced in Definition 4.24). A revised spatio-temporal function C is called *movable* if for all stimuli $(i, i_M, i_F, a) \in \vec{I} \times \vec{I}_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X)$ and reactions $(o, o_M, p, b, y, x) \in C(i, i_M, i_F, a)$ there exists a stimulus with zero kinetic energy $(i, i_{M,0}, i_{F,0}, a) \in \vec{I} \times \vec{I}_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X)$ and reaction $(o, o_{M,0}, p_0, b_0, y_0, x_0) \in C(i, i_{M,0}, i_{F,0}, a)$, such that for each kinetic energy output channel $\omega \in O_M$ the initial kinetic energy output channel assignment $o_M(\omega)(0)$ equals to the initial energy output channel assignment $o_{M,0}(\omega)(0)$, i.e. $o_M(\omega)(0) = o_{M,0}(\omega)(0)$, and for each computation step $n \in \mathbb{N}$ the follow-up kinetic energy output channel assignment $o_M(\omega)(n+1)$ can be obtained from the follow-up kinetic energy output channel assignment $o_{M,0}(\omega)(n)$ through transformation by kinetic energy input channel and forwarding histories i_M and i_F, i.e.

$$o_M(\omega)(n+1) = its(i_F(C))(n) \circ \bigcirc_{\iota \in I_M} its(i_M(\iota))(n) \circ o_{M,0}(\omega)(n+1) \circ$$

$$\bigcirc_{\iota \in I_M} its^{-1}(i_M(\iota))(n) \circ its^{-1}(i_F(C))(n).$$

Furthermore, for each part $\pi \in P$ the initial part assignment $p(\pi)(0)$ equals to the initial part assignment $p_0(\pi)(0)$, i.e. $p(\pi)(0) = p_0(\pi)(0)$, and for each computation step $n \in \mathbb{N}$ the follow-up part assignment $p(\pi)(n+1)$ can be obtained from the follow-up part

assignment $p_0(\pi)(n)$ through transformation by kinetic input channel and forwarding histories i_M and i_F, i.e.

$$p(\pi)(n+1) = its(i_F(C))(n) \circ \bigcirc_{\iota \in I_M} its(i_M(\iota))(n)(p_0(\pi)(n+1)).$$

Then, for each binding $\beta \in B$ the initial binding assignment $b(\beta)(0)$ equals to the initial binding assignment $b_0(\beta)(0)$, i.e. $b(\beta)(0) = b_0(\beta)(0)$, and for each computation step $n \in \mathbb{N}$ the follow-up binding assignment $b(\beta)(n+1) = (v, c, f)$ can be obtained from the follow-up binding assignment $b_0(\beta)(n+1) = (v_0, c, f)$ through transformation by kinetic energy input channel and forwarding histories i_M and i_F, i.e.

$$v = its(i_F(C))(n) \circ \bigcirc_{\iota \in I_M} its(i_M(\iota))(n)(v_0).$$

Subsequently, for each entry $\upsilon \in Y$ the initial entry assignment $y(\upsilon)(0)$ equals to the initial entry assignment $y_0(\upsilon)(0)$, i.e. $y(\upsilon)(0) = y_0(\upsilon)(0)$, and for each computation step $n \in \mathbb{N}$ the follow-up entry assignment $y(\upsilon)(n+1) = (t', C')$ can be obtained from the follow-up entry assignment $y_0(\upsilon)(n+1) = (t'_0, C')$ through transformation by kinetic energy input channel and forwarding histories i_M and i_F, i.e.

$$t' = its(i_F(C))(n) \circ \bigcirc_{\iota \in I_M} its(i_M(\iota))(n) \circ t'_0.$$

Finally, for each exit $\xi \in X$ the initial exit assignment $x(\xi)(0)$ equals to the initial exit assignment $x_0(\xi)(0)$, i.e. $x(\xi)(0) = x_0(\xi)(0)$, and for each computation step $n \in \mathbb{N}$ the follow-up exit assignment $x(\xi)(n+1)$ can be obtained from the follow-up exit assignment $x_0(\xi)(n+1)$ through transformation by kinetic energy input channel and forwarding histories i_M and i_F, i.e.

$$x(\xi)(n+1) = its(i_F(C))(n) \circ \bigcirc_{\iota \in I_M} its(i_M(\iota))(n)(x_0(\xi)(n+1)).$$

Consequently, the reaction of revised spatio-temporal function C to kinetic energy input channel and forwarding histories i_M and i_F is obtained from the reaction of C to the zero kinetic energy input channel and forwarding histories $i_{M,0}$ and $i_{F,0}$ through transforming the kinetic energy output channel history $o_{M,0}$, the part history p_0, the binding history b_0, the entry history y_0, and the exit history x_0. Only the output channel history o is not affected by the kinetic energy input channel and forwarding histories i_M and i_F. Furthermore, note that the part, binding, entry, and exit histories p, b, y, and x are affected by integrated transformation streams. In contrast, the kinetic energy output history o_M is affected by the inverse integrated transformation stream also. This behavior is quite different from the behavior of the position operator *pos* of the original STEM theory (see Section 4.2.2). The reason for this difference is that in the revised

component formalism the position and orientation of revised spatio-temporal functions is encoded explicitly in the kinetic energy input channel history i_M and the kinetic energy forwarding history i_F. In the original STEM theory, the position and orientation is defined only implicitly in the detector, part, mover, entry, and exit histories as well as the position operator. Consequently, the position and orientation information can be accessed more easily in the formalism, which helps defining certain properties later.

Revised spatio-temporal components Here, movable spatio-temporal functions C over $I, I_M, O, O_M, P, B, Y, X$ also are called revised spatio-temporal components here.

Revised composition operator Then, the question arises how revised spatio-temporal components C_1 and C_2 can be composed, which is not possible in the original STEM theory. Hereby, particularly the exchange of kinetic energy has to be considered. Subsequently, the revised composition operator $\odot : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined such that for each revised spatio-temporal component C_1 over $I_1, I_{M,1}, O_1, O_{M,1}, P_1, B_1, Y_1, X_1$ and revised spatio-temporal component C_2 over $I_2, I_{M,2}, O_2, O_{M,2}, P_2, B_2, Y_2, X_2$ the revised composition operator yields

$$C_1 \odot C_2 : \vec{I} \times \vec{I}_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X) \to \mathbb{P}(\vec{O} \times \vec{O}_M \times \vec{P} \times \vec{B} \times \vec{Y} \times \vec{X})$$

with input channels $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1) \cup (I_{M,1} \setminus O_{M,2}) \cup (I_{M,2} \setminus O_{M,1})$, empty kinetic energy input channels $I_M = \emptyset$, output channels $O = O_1 \cup O_2$, kinetic energy output channels $O_M = O_{M,1} \cup O_{M,2}$, parts $P = P_1 \cup P_2$, bindings $B = B_1 \cup B_2$, entries $Y = Y_1 \cup Y_2$, and exits $X = X_1 \cup X_2$. Furthermore, for each stimulus $(i, i_M, i_{F,0}, a) \in \vec{I} \times \vec{I}_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X)$ with $\forall C \in \mathcal{C} : i_{F,0}(C) = t_0$ and reaction $(o, o_M, p, b, y, x) \in (C_1 \odot C_2)(i, i_M, i_{F,0}, a)$ there must exist an extended (kinetic energy) input channel history $i' \in \vec{Z}$ with $Z = I_1 \cup I_2 \cup I_{M,1} \cup I_{M,2}$ and $i'|_I = i$ such that

$$(o|_{O_1}, o_M|_{O_{M,1}}, p|_{P_1}, b|_{B_1}, y|_{Y_1}, x|_{X_1}) \in C_1(i'|_{I_1}, i'|_{I_{M,1}}, i_{F,0}, a|_{B_1 \cup Y_1 \cup X_1})$$

$$\wedge$$

$$(o|_{O_2}, o_M|_{O_{M,2}}, p|_{P_2}, b|_{B_2}, y|_{Y_2}, x|_{X_2}) \in C_2(i'|_{I_2}, i'|_{I_{M,2}}, i_{F,0}, a|_{B_2 \cup Y_2 \cup X_2}).$$

Note that i' includes the histories of the hidden input channels $I_1 \setminus O_2$ and $I_2 \setminus O_1$ as well as the hidden kinetic energy input channels $I_{M_1} \setminus O_{M,2}$ and $I_{M,2} \setminus O_{M,1}$. Furthermore, the revised spatio-temporal function $C_1 \odot C_2$ must be movable with respect to arbitrary $i_F \in \vec{I}_F$. Consequently, one needs to construct from each stimulus $(i, i_M, i_F, a) \in \vec{I} \times \vec{I}_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X)$ with $i_F(C_1 \odot C_2) = t_0$, already computed reaction $(o, o_{M,0}, p_0, b_0, y_0, x_0) \in$

$(C_1 \odot C_2)(i, i_M, i_F, a)$, and transformation stream $t \in T^\infty$ a transformed stimulus and reaction $(o, o_M, p, b, y, x) \in C_1 \odot C_2(i, i_M, i'_F, a)$ with $i'_F(C_1 \odot C_2) = t$ and $i'_F|_{\mathcal{C}\setminus(C_1 \odot C_2)} = i_F|_{\mathcal{C}\setminus(C_1 \odot C_2)}$. This construction is left to the interested reader.

Motion interface operator Then, the question arises how to augment the composed spatio-temporal component $C : C_1 \odot C_2$ with its own kinetic energy input channels, which only exist implicitly in the STEM theory (see motion operator $\vdash_\mu$ in Definition 4.25). Therefore, the *motion interface operator* $\vdash_{I'_M} : \mathcal{C} \to \mathcal{C}$ with additional kinetic energy input channel labels $I'_M \subseteq \mathcal{I}_M$ and the universe of kinetic energy input channel labels $\mathcal{I}_M$ is introduced such that for each revised spatio-temporal component $C : \vec{I} \times \vec{I}_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X) \to \mathbb{P}(\vec{O} \times \vec{O}_M \times \vec{P} \times \vec{B} \times \vec{Y} \times \vec{X})$ the motion interface operator yields

$$\vdash_{I'_M} (C) = C' : \vec{I} \times \vec{I}''_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X) \to \mathbb{P}(\vec{O} \times \vec{O}_M \times \vec{P} \times \vec{B} \times \vec{Y} \times \vec{X})$$

with augmented kinetic energy input channels $I''_M = I_M \cup I'_M$. Furthermore, the revised spatio-temporal function $\vdash_{I'_M} (C)$ must be movable with respect to $i_M \in \vec{I}''_M$ yielding a revised spatio-temporal component by definition. The reactions of the revised spatio-temporal component $\vdash_{I'_M} (C)$ to stimulus $(i, i_M, i_F, a) \in \vec{I} \times \vec{I}''_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X)$ can be derived easily from the reactions of the revised spatio-temporal component C to the stimulus $(i, i_M|_{I_M}, i_F, a) \in \vec{I} \times \vec{I}_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X)$. Again, the construction is left to the interested reader.

Static component architectures Finally, the revised component formalism entails revised definitions of the collision sensing and the channel binding properties (see Section 4.2.3), while the mover binding property is subsumed by the channel binding property and the definition of movable spatio-temporal functions. Before introducing the revised property definitions, the notion of *static component architectures* is introduced. A static component architecture is defined as a mapping $A : \mathcal{C} \to \mathbb{P}(\mathcal{I}_M) \times \mathbb{P}(\mathcal{C})$ with the universe of kinetic energy input channel labels $\mathcal{I}_M$ such that for each revised spatio-temporal component $C \in \mathcal{C}$ the mapping $A(C) = (I_M, C_C)$ provides the augmented kinetic energy input channels I_M and the child components C_C such that

$$(C_C = I_M = \emptyset) \vee (C = \vdash_{I_M} (\bigodot_{C' \in C_C} C')).$$

Consequently, a component is mapped (1) to the empty set of kinetic energy input channels and child components or (2) to the child components, it is composed of, and the kinetic energy input channels, the composition is augmented by. In the first case C is

called an *atomic component*, while in the second case C is called a *composite component* and $A(C)$ its *decomposition*. Based on the concept of static component architectures A the *descendants mapping* $D : \mathcal{C} \to \mathbb{P}(\mathcal{C})$ is introduced such that

$$D(C) = C_C \cup \bigcup_{C' \in C_C} D(C') \text{ with } A(C) = (I_M, C_C).$$

Note that the descendants mapping D is defined recursively. Therefore, the descendants mapping also can be used to retrieve second and lower level subcomponents, while the static component architecture A only returns the first level subcomponents. Consequently, with mapping D one can argue about all components, both composite and atomic, which are contained in component C. Note that, hereby, the composite components include their augmented kinetic energy input channels.

Generated component streams Furthermore, the revised definition of the collision sensing and channel binding properties requires a revised definition of the generated component stream G (see Definition 4.27). Given stimulus $(i, i_M, i_F, a) \in \vec{I} \times \vec{I}_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X)$ and reaction $(o, o_M, p, b, y, x) \in C(i, i_M, i_F, a)$ the generated component stream $G \in \mathbb{P}(\mathcal{C})^\infty$ is defined such that for each computation step $n \in \mathbb{N}$ it contains in the follow-up computation step $n + 1$ a component $C' \in G(n + 1)$ over $I', I'_M, O', O'_M, P', B', Y', X'$ if the stream G contained the component C' in the previous computation step, i.e. $C' \in G(n)$, and the component C' was not deleted, i.e.

$$\nexists \pi' \in P' : \exists \xi \in X : p'(\pi')(n) \bowtie x(\xi)(n) \vee \bigvee_{C'' \in G(n)} \exists \xi'' \in X'' : p'(\pi')(n) \bowtie x''(\xi'')(n)$$

or the generated component stream G did not contain the component C' in the previous computation step, i.e. $C' \notin G(n)$ and the component C' was created at some position and orientation $t' \in T$, i.e.

$$\exists \upsilon \in Y : y(\upsilon)(n) = (t', C') \vee \bigvee_{C'' \in G(n)} \exists \upsilon'' \in Y'' : y''(\upsilon'')(n) = (t', C').$$

Note that in the previous formulas $(o', o'_M, p', b', y', x') \in C'(i', i'_M, i_F, a')$ refers to the behavior of revised spatio-temporal component $C' \in G(n+1)$, while $(o'', o''_M, p'', b'', y'', x'') \in C''(i'', i''_M, i_F, a'')$ refers to the behavior of revised spatio-temporal component $C'' \in G(n)$. Note that, in contrast to the original STEM theory (see Section 4.2.3), the revised definition of the generated component stream G does not encode the position and orientation $t' \in T$ of the generated revised spatio-temporal components. The reason for this

difference is that in the revised component formalism the position and orientation of components is encoded explicitly in the kinetic energy input channel and forwarding histories $i'_M \in \vec{I}'_M$, $i''_M \in \vec{I}''_M$, and $i_F \in \vec{I}_F$. Furthermore, note that the revised definition of the generated component stream G allows generated components to generate and delete other components (including themselves). This behavior was not supported by the original definition of the generated component stream. Consequently, the revised definition is more powerful than the original definition.

Collision sensing behaviors Then, the collision sensing property (see Definition 4.28) is extended such that only colliding components can be detected, which implement the syntactic interface $c \in \mathbb{P}(\mathcal{O} \times \mathcal{I})$ prescribed by the bindings $b(\beta)(n) = (v, c, f)$ at computation step $n \in \mathbb{N}$ with binding label $\beta \in B$ and binding history $b \in \vec{B} = \{B \to (V \times \mathbb{P}(\mathcal{O} \times \mathcal{I}) \times \mathbb{B})^\infty\}$. Furthermore, the colliding components might include subcomponents $C' \in D(C)$ of component C in addition to generated components $C' \in G(n)$ with computation step $n \in \mathbb{N}$. Hence, the behavior $(o, o_M, p, b, y, x) \in C(i, i_M, i_F, a)$ is called *collision sensing*, when a revised spatio-temporal component $C' \in \mathcal{C}$ over $I', I'_M, O', O'_M, P', B', Y', X'$ with behavior $(o', o'_M, p', b', y', x') \in C'(i', i'_M, i_F, a')$ is bound to binding $b(\beta)(n) = (v, c, f)$ with binding label $\beta \in B$ at computation step $n \in \mathbb{N}$, i.e. $C' \in a(\beta)(n)$, if the component C' is part of the generated component stream G, i.e.

$$C' \in G(n)$$

or the component C' is a descendant of the component C and there exists another revised spatio-temporal component C'' over $I'', I''_M, O'', O''_M, P'', B'', Y'', X''$, which also is a descendant of C, is not related to C', and defines the binding β, i.e.

$$C' \in D(C) \land \exists C'' \in D(C) : \beta \in B'' \land C' \notin D(C'') \land C'' \notin D(C')$$

and the component C', either part of the generated component stream $G(n)$ or part of the component architecture A, implements the syntactic interface prescribed by the dynamic channels $c \in \mathbb{P}(\mathcal{O} \times \mathcal{I})$ of the binding $b(\beta)(n) = (v, c, f)$, i.e.

$$\forall (\omega, \iota) \in c : \omega \in O' \lor \omega \in O'_M \lor \iota \in I' \lor \iota \in I'_M$$

and at least one part $\pi' \in P'$ of component C' must collide with the volume v of the binding $b(\beta)(n) = (v, c, f)$, where the volume v replaces the spatial selection predicate $\sigma \in \mathcal{S}$ of the original STEM theory, i.e.

$$\exists \pi' \in P' : p'(\pi')(n) \bowtie v.$$

Note that the collision sensing property applies to the behaviors $(o', o'_M, p', b', y', x') \in C'(i', i'_M, i_F, a')$ of the generated components $C' \in G(n)$ as well. Furthermore, note that the revised collision sensing property is stronger than the original collision sensing property. In particular, the syntactic interface of each bound component $C' \in G(n)$ at computation step $n \in \mathbb{N}$ can be constrained by means of the dynamic channels $c \in \mathbb{P}(\mathcal{O} \times \mathcal{I})$ of bindings $b(\beta)(n) = (v, c, f)$. In contrast, the original collision sensing property was based on the spatial extent (i.e. the parts) of components only.

Channel binding behaviors Based on the collision sensing property, the channel binding property (see Definition 4.29) is adapted to forward the output channel and kinetic energy output channel assignments along the dynamic channels $c \in \mathbb{P}(\mathcal{O} \times \mathcal{I})$ of bindings $b(\beta)(n) = (v, c, f)$. A stimulus $(i, i_M, i_F, a) \in \vec{I} \times \vec{I}_M \times \vec{I}_F \times \vec{A}(B \cup Y \cup X)$ and reaction $(o, o_M, p, b, y, x) \in C(i, i_M, i_F, a)$ is called *channel binding* if for each binding $b(\beta)(n) = (v, c, f)$ with binding label $\beta \in B$ and computation step $n \in \mathbb{N}$, dynamic channel $(\omega, \iota) \in c$, and bound component $C' \in a(\beta)(n)$ over $I', I'_M, O', O'_M, P', B', Y', X'$ with behavior $(o', o'_M, p', b', y', x') \in C'(i', i'_M, i_F, a')$, the component C' is a generated component, i.e. $C' \in G(n)$, and the (kinetic energy) output channels of bound component C' are forwarded to the (kinetic energy) input channels of component C, i.e.

$$\omega \in O' \wedge \iota \in I \Rightarrow o'(\omega)(n) = i(\iota)(n)$$

$$\omega \in O'_M \wedge \iota \in I \Rightarrow o'_M(\omega)(n) = i(\iota)(n)$$
$$\omega \in O'_M \wedge \iota \in I_M \Rightarrow o'_M(\omega)(n) = i_M(\iota)(n),$$

and the (kinetic energy) output channels of component C are forwarded to the (kinetic energy) input channels of bound component C', i.e.

$$\omega \in O \wedge \iota \in I' \Rightarrow o(\omega)(n) = i'(\iota)(n)$$

$$\omega \in O_M \wedge \iota \in I' \Rightarrow o_M(\omega)(n) = i'(\iota)(n)$$
$$\omega \in O_M \wedge \iota \in I'_M \Rightarrow o_M(\omega)(n) = i'_M(\iota)(n),$$

or the component C' is a descendant of component C, i.e. $C' \in D(C)$, and there exists another descendant component $C'' \in D(C)$ over $I'', I''_M, O'', O''_M, P'', B'', Y'', X''$ of component C, which is not related to C', i.e. $C'' \notin D(C')$ and $C' \notin D(C'')$, and C'' defines the binding β, i.e. $\beta \in B''$, and the (kinetic energy) output channels of C' are forwarded to the (kinetic energy) input channels of C'', i.e.

$$\omega \in O' \wedge \iota \in I'' \Rightarrow o(\omega)(n) = i(\iota)(n)$$

$$\omega \in O'_M \wedge \iota \in I'' \Rightarrow o_M(\omega)(n) = i(\iota)(n)$$

$$\omega \in O'_M \wedge \iota \in I''_M \Rightarrow o_M(\omega)(n) = i_M(\iota)(n),$$

and the (kinetic energy) output channels of component C'' are forwarded to the (kinetic energy) input channels of component C', i.e.

$$\omega \in O'' \wedge \iota \in I' \Rightarrow o(\omega)(n) = i(\iota)(n)$$

$$\omega \in O''_M \wedge \iota \in I' \Rightarrow o_M(\omega)(n) = i(\iota)(n)$$

$$\omega \in O''_M \wedge \iota \in I'_M \Rightarrow o_M(\omega)(n) = i_M(\iota)(n).$$

Note that the channel binding property applies to the behaviors $(o', o'_M, p', b', y', x') \in C'(i', i'_M, i_F, a')$ of the generated components $C' \in G(n)$ as well. Furthermore, note that the revised channel binding property not only enables communication between component C and generated components $C' \in G(n)$, but the property also supports energy and data exchange between independent subcomponents $C', C'' \in D(C)$. Consequently, the formalism allows one to express dynamic interaction within a static component architecture A. Additionally, the dynamic interaction includes exchange of kinetic energy such that component C, some subcomponent $C' \in D(C)$, or some generated component $C' \in G(n)$ can be moved. Consequently, the revised channel binding property in conjunction with the definition of movable spatio-temporal functions also subsume the original mover binding property (see Definition 4.30).

Motion forwarding behaviors Finally, the question arises how the kinetic energy forwarding histories $i_F \in \vec{I}_F$ are computed. In the following, assume an arbitrary behavior $(o, o_M, p, b, y, x) \in C(i, i_M, i_F, a)$ with generated component stream $G \in \mathbb{P}(\mathcal{C})^\infty$, computation step $n \in \mathbb{N}$, and generated component $C' \in G(n)$ over $I', I'_M, O', O'_M, P', B', Y', X'$. First the subset of bindings $B_C^{C'} \subseteq B$ of component C is derived which the component C' is bound to in computation step n, i.e.

$$B_C^{C'} = \{\beta \in B \mid C' \in a(\beta)(n) \wedge \exists v \in V, c \in \mathbb{P}(\mathcal{O} \times \mathcal{I}) : (v, c, true) = b(\beta)(n)\}.$$

Furthermore, for each other generated component $C'' \in G(n), C'' \neq C'$ over label sets $I'', I''_M, O'', O''_M, P'', B'', Y'', X''$ with behavior $(o'', o''_M, p'', b'', y'', x'') \in C''(i'', i''_M, i_F, a'')$ the subset of bindings $B_{C''}^{C'} \subseteq B''$ of component C'' is defined which the component C' is bound to in computation step n, i.e.

$$B_{C''}^{C'} = \{\beta'' \in B'' \mid C' \in a''(\beta'')(n) \wedge \exists v \in V, c \in \mathbb{P}(\mathcal{O} \times \mathcal{I}) : (v, c, true) = b''(\beta'')(n)\}.$$

Note that only bindings $b(\beta)(n) = (v, c, true)$ and $b''(\beta'')(n) = (v'', c'', true)$ are considered where the boolean forwarding flag is *true*. Then, the set of components $C^{C'}_{C^*} \subseteq \mathcal{C}$ with $C^* \in \{C\} \cup G(n) \setminus \{C'\}$ is derived which include any of the selected bindings $\beta \in B^{C'}_{C^*}$ in their syntactic interface, i.e.

$$C^{C'}_{C^*} = \{C''' \in D(C^*) \mid \exists \beta \in B^{C'}_{C^*} : \beta \in B''\}.$$

Subsequently, the kinetic energy input channels is calculated which take effect on the selected components $C''' \in C^{C'}_{C^*}$ over $I''', I'''_M, O''', O'''_M, P''', B''', Y''', X'''$. Note that due to the revised composition operator $\odot$ the kinetic energy input channels can be found in the regular input channels I^* of component C^*, the kinetic energy input channels I^*_M of component C^*, or the kinetic energy output channels O^*_M of component C^*, i.e.

$$I^{C'}_{C^*} = \{\iota \in I^* \mid \exists C''' \in C^{C'}_{C^*} : \iota \in I'''_M\},$$

$$I^{C'}_{M,C^*} = \{\iota \in I^*_M \mid \exists C''' \in C^{C'}_{C^*} : \iota \in I'''_M\} \setminus I^{C'}_{C^*},$$

$$O^{C'}_{M,C^*} = \{\omega \in O^*_M \mid \exists C''' \in C^{C'}_{C^*} : \omega \in I'''_M\} \setminus I^{C'}_{C^*} \setminus I^{C'}_{M,C^*}.$$

Finally, the initial kinetic energy forwarding $i_F(C')(0) \in T$ is set to the identity transformation, i.e. $i_F(C')(0)(v) = v$, and the follow-up kinetic energy forwarding $i_F(C')(n+1)$ is calculated from the kinetic energy forwarding of the components in $C^{C'}_{C^*}$ as well as the kinetic energy input channels in $I^{C'}_{C^*}$, $I^{C'}_{M,C^*}$, and $O^{C'}_{M,C^*}$ such that

$$i_F(C')(n+1) = \bigcirc_{C''' \in C^{C'}_{C}} i_F(C''')(n+1) \circ \bigcirc_{C'' \in G(n)} (\bigcirc_{C'' \in C^{C'}_{C''}} i_F(C''')(n+1)) \circ$$

$$\bigcirc_{\iota \in I^{C'}_{C}} i(\iota)(n+1) \circ \bigcirc_{C'' \in G(n)} (\bigcirc_{\iota \in I^{C'}_{C''}} i''(\iota)(n+1)) \circ$$

$$\bigcirc_{\iota \in I^{C'}_{M,C}} i_M(\iota)(n+1) \circ \bigcirc_{C'' \in G(n)} (\bigcirc_{\iota \in I^{C'}_{M,C''}} i''(\iota)(n+1)) \circ$$

$$\bigcirc_{\omega \in O^{C'}_{M,C}} o_M(\omega)(n+1) \circ \bigcirc_{C'' \in G(n)} (\bigcirc_{\omega \in O^{C'}_{M,C''}} o''(\omega)(n+1)).$$

Consequently, the kinetic energy of each binding component $C''' \in C^{C'}_{C^*}$ is transferred to each bound component $C' \in \mathcal{C}$. Note that the order, in which the transformations are applied, is not prescribed completely in the above equation. However, the kinetic energy of lower level components $C'''_l \in C^{C'}_{C^*}$ should be applied before the kinetic energy of higher level components $C'''_h \in C^{C'}_{C^*}$ with $C'''_l \in D(C'''_h)$. Note that the composition operator $\odot$ and the motion interface operator $\vdash_{I'_M}$ entail the same behavior within static component architectures A. Furthermore, note that two or more components might bind each other mutually. In such cases, the above equation poses a fixed point problem, which has

to be handled with appropriate means (see Section 6.2.2). Moreover, a component C' might not be bound by any binding $\beta \in B$ and $\beta \in B''$ with $C'' \in G(n)$ at computation step $n \in \mathbb{N}$. In such cases, the kinetic energy forwarding $i_F(C')(n)$ is set to the identity transformation, i.e.

$$i_F(C')(n)(v) = v.$$

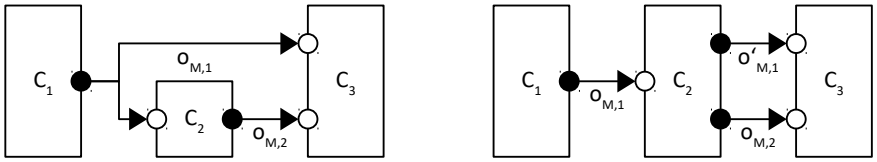
Finally, the behavior $(o, o_M, p, b, y, x) \in C(i, i_M, i_F, a)$ of component C and the behaviors $(o', o'_M, p', b', y', x') \in C'(i', i'_M, i_F, a')$ of generated components $C' \in G(n)$ with $n \in \mathbb{N}$ are called *motion forwarding*, if the kinetic energy forwarding history $i_F \in \vec{I}_F$ satisfies the above equations. Note that in the original STEM theory all bindings are motion forwarding per definition (see Definition 4.30). Since the revised theory also supports non-motion forwarding bindings, kinetic energy exchange, e.g., by means of airflow can be described more naturally. In contrast, in the original STEM theory only fixed serial kinematic chains can be described, which prevent, e.g., airflow modeling.

Revised spatio-temporal computations In the remainder of this thesis, only behaviors $(o, o_M, p, b, y, x) \in C(i, i_M, i_F, a)$ of revised spatio-temporal components C with generated component stream G and behaviors $(o', o'_M, p', b', y', x') \in C'(i', i'_M, i_F, a')$ of generated components $C' \in G(n)$ at computation step $n \in \mathbb{N}$ are considered that are collision sensing, channel binding, and motion forwarding according to the previous definitions. Note that the underlying formalism of revised spatio-temporal component specifications does not ensure these properties intrinsically. Rather, the desired properties have to be encoded, e.g., into some underlying computation engine. In the following, such behaviors are called *revised spatio-temporal computations*.

Modeling kinematic chains

In contrast to the original STEM theory (see Section 4.2), the revised component formalism supports three different ways of modeling kinematic chains. In the following, the different ways are illustrated using a kinematic chain consisting of three elements C_1, C_2, and C_3. Note that in STEM the kinematic chain would be represented by $C_1 \vdash_{\mu_1} (C_2 \vdash_{\mu_2} C_3)$ with mover label $\mu_1 \in M_1$ of element C_1 and mover label $\mu_2 \in M_2$ of element C_2. In contrast, in the revised formalism one way of modeling the chain is to pull through the kinetic energy output channels from C_1 to C_3 as illustrated in Figure 5.8. Hereby, a *direct* and an *indirect* way can be distinguished. In the direct way, the kinetic energy output channel $o_{M,1} \in O_{M,1}$ of element C_1 is connected to both C_2 and C_3. In contrast, in the indirect way the kinetic energy output channel $o_{M,1}$ of element C_1 is connect to C_2 only, while C_2 defines an extra kinetic energy output channel $o'_{M,1}$

forwarding the kinetic energy of $o_{M,1}$ to C_3. Note that one has to ensure the correct order in which the two kinetic energy inputs to component C_3 are applied to achieve the same result than in the original STEM theory.



(a) *Direct* way (b) *Indirect* way

Figure 5.8: *Channel-based* model of kinematic chains.

Alternatively, one can use the revised composition operator to model kinematic chains, which yields similar models than the original STEM theory as illustrated in Figure 5.9. Therefore, one needs to introduce two additional actuator components A_1 and A_2, which are responsible for producing the kinetic energy acting upon the elements C_2 and C_3 of the kinematic chain. Hereby, the revised composition operator ensures that the kinematic energy output channel $o_{M,1}$ of actuator A_1 is transfered also chain element C_3. Furthermore, from the point of view of the first element C_1, the later segments C_2 and C_3 form a single component. Note that this model of kinematic chains is more verbose than the previous due to the need for additional actuator components. However, the more verbose model also resembles reality more closely because actuators are needed to operate, e.g., the segments of a robot arm.

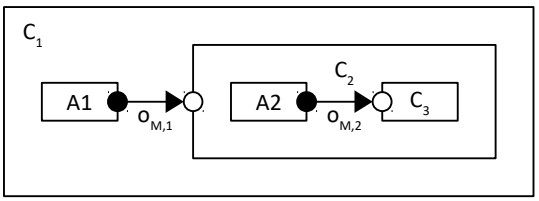


Figure 5.9: *Composition-based* model of kinematic chains.

Finally, note that, in contrast to the original STEM theory, the revised formalism supports modeling kinematic chains with loops. Consequently, more complex interactions can be described such as the last segment of the robot arm (i.e. the *tip*) hitting the first segment of the robot arm (i.e. the *root*) and, hence, causing motion of the entire robot arm. However, also unrealistic behaviors can be produced with the revised component

formalism such as Baron Munchausen pulling himself out of the swamp [Hum11]. At this point, the expressiveness of the modeling technique during conceptual design are preferred over the realism of the models, which can be obtained. Consequently, creative potentials can be exploited more easily. However, on the downside the engineers have to check manually whether the models are realistic or not.

Concrete data model

Subsequently, the concrete data model for representing revised spatio-temporal components in a machine-readable format is depicted in Figure 5.10. Note that in addition to the changes in the underlying formalism, the data model supports multiple instantiations of components. Instantiation has been added to the specification technique to enable reuse. Reuse of existing components is an important factor in manufacturing systems engineering [Thr05].

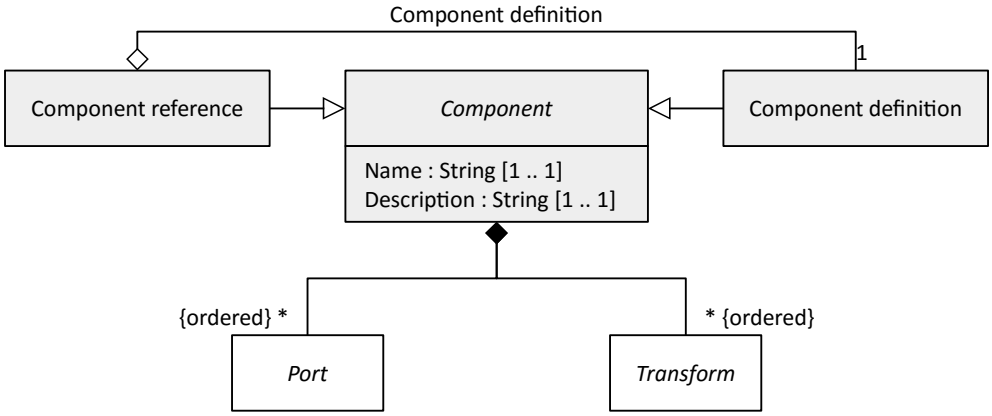


Figure 5.10: Abstract component class and subclasses.

The abstract component base class defines a human-readable *name* and a *description* for documentation purposes. Then, abstract component objects contain an ordered list of *ports* (see Section 5.2.2) and *transforms*, which have been introduced previously in Section 5.1.5. The port objects represent the labels I, I_M, O, O_M, B, Y, X of the revised spatio-temporal component C (note that the part labels P are excluded here), while the transforms $t = t_1 \circ ... \circ t_n \in T$ define the initial position and orientation of the component in its superordinate coordinate system. Hereby, the port order determines uniquely the order, in which the kinetic energy input channels $\iota_M \in I_M$ are applied in the definition of movable spatio-temporal functions and motion forwarding behaviors.

Hence, the degree of determinism can be increased. Furthermore, the transformation t can be interpreted as the first entry $i_F(C)(0) = t$ in the kinetic energy forwarding histories $i_F \in \vec{I}_F$. Note that these two features are not described in the above theory for simplicity. Finally, *component definitions* and *component references* are distinguished. Component definitions allow one to specify new components, while component references allow one to instantiate existing component definitions in a novel context instead. In the following, both types of components are described in more detail.

Component definition A component definition not only consists of its syntactic interface $I, I_M, O, O_M, P, B, Y, X$ and initial transform $t \in T$, but provides much more information. In particular, component definitions cover everything from customer requirements, manufacturing processes, and test cases to subcomponents, behaviors, and parts. The data model behind component definitions is depicted in Figure 5.11.

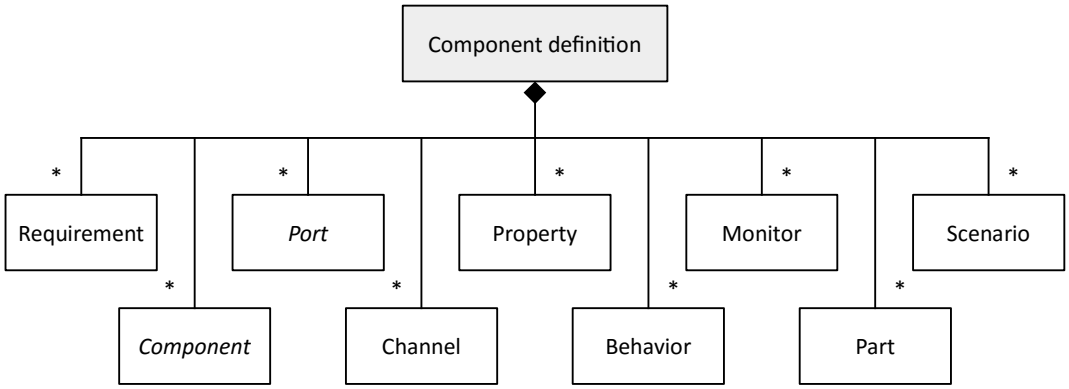


Figure 5.11: Concrete component definition class.

The concrete component definition class contains nine classes of children. First, the expectations of the different stakeholders (e.g. the customer) can be expressed in natural language using the *requirement* concept (see Section 5.3.1). Then, the syntactic interface of the component definition can be described using the *port* concept (see Section 5.2.2). Afterwards, the natural language requirements can be formalized over the ports using the *property* concept (see Section 5.3.2). Additionally, natural language requirements and manufacturing processes can be formalized over the ports using the *monitor* concept (see Section 5.3.3). Note that monitors are more expressive than the properties. Thereafter, test cases can be described for verifying the implementation with respect to properties and monitors using the *scenario* concept (see Section 5.3.4). Subsequently, both compo-

nent definitions and component references and the *channel* concept (see Section 5.2.3) can be used to define the decomposition of the containing component. Note that the subcomponents and their interactions specify the behavior and the spatial extent of the containing component. However, complementary (parts of) the behavior and the spatial extent of the containing component can be described using the *behavior* concept (see Section 5.2.5) and the *part* concept (see Section 5.2.4) directly. In the following each concept is explained in more detail.

Component reference In contrast, in the concrete data model component references only provide a link to an existing component definition. Consequently, component references C' do not define their own syntactic interface $I', I'_M, O', O'_M, P', B', Y', X'$ and semantic interface $C' : \vec{I}' \times \vec{I}'_M \times \vec{I}_F \times \vec{A}(B' \cap Y' \cap X') \to \mathbb{P}(\vec{O}' \times \vec{O}'_M \times \vec{P}' \times \vec{B}' \times \vec{Y}' \times \vec{X}')$. Rather, the syntactic and semantic interfaces of some referenced component definition C is instantiated in an new context. Essentially, component references correspond to renamed versions of the original component definition. In particular, the renaming applies to the inputs I, the kinetic energy inputs I_M, the outputs O, the kinetic energy outputs O_M, the parts P, the bindings B, the entries Y, and the exists X of the original component C.

5.2.2 Ports

Based on the observation concept (see Section 5.1.1) and the component concept (see Section 5.2.1) the port concept is introduced. Ports represent the labels I, I_M, O, O_M, B, Y, X of revised spatio-temporal components. Note that ports are used to model input and output observations as well as movers, bindings, entries and exits at the same time. The corresponding data model for representing ports is depicted in Figure 5.12.

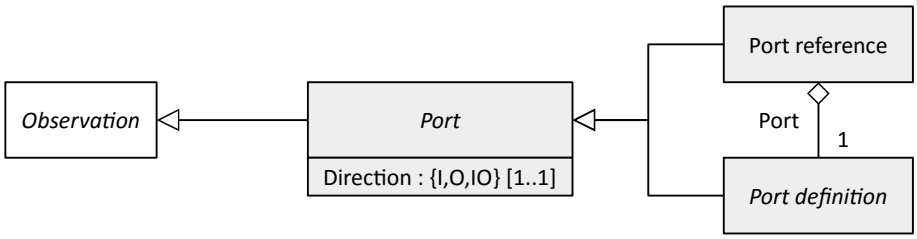


Figure 5.12: Abstract port class and subclasses.

Recall that the abstract observation base class defines a human-readable *name*, a *read type*, and one or more *write types* (see Section 5.1.1). The abstract port base class adds a

direction attribute, which determines whether the port is an input port, an output port, or an input-output port from the perspective of the containing component. Note that input-output ports appear on both sides of the semantic interface $C : \vec{I} \times \vec{I}_M \times \vec{I}_F \times \vec{A}(B \cap Y \cap X) \to \mathbb{P}(\vec{O} \times \vec{O}_M \times \vec{P} \times \vec{B} \times \vec{Y} \times \vec{X})$. Consequently, input-output ports represent the labels B, Y, X, while input ports represent the labels I, I_M, and output ports represent the labels O, O_M. Finally, two subclasses are defined, namely *port definitions* and *port references*. As the names suggests, port definitions are used to define the interface of component definitions (see Section 5.2.1), while port references are used to represent the renamed interface of component references (again see Section 5.2.1).

Port definitions

Then, the proposed modeling technique distinguishes four types of port definitions: *Material port definitions*, *energy port definitions*, *data port definitions*, and *generic port definitions*. Each type of port definition describes a different form of interaction between revised spatio-temporal components. The corresponding extension of the previous data model is illustrated in Figure 5.13.

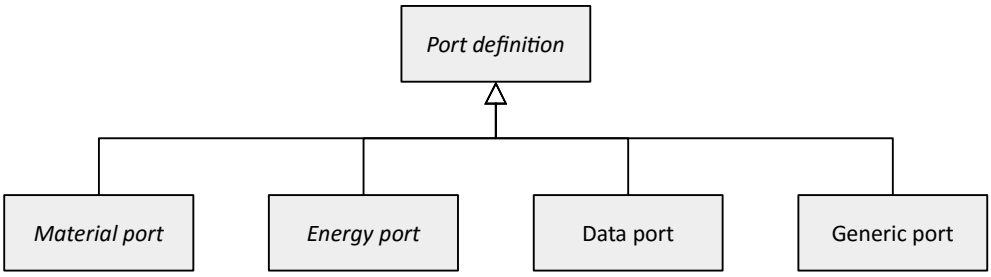


Figure 5.13: Abstract port definition class and subclasses.

Material ports represent the labels B, Y, X and can be used for describing spatial interaction between components (see Section 5.2.1). In contrast, energy ports represent the labels I, I_M, O, O_M and can be employed for specifying energy exchange. Then, data ports represent the labels I, O and are meant for modeling digital information exchange instead. Finally, generic ports represent the labels I, O and are intended for representing any other from of interaction between components not covered by the previous types. Note that generic ports can be used also in case one is unsure about how to realize some interaction between components. For example, it might be possible to implement the interaction by means of data or energy exchange. Choosing either option represents a design decision which can be delayed with the generic port concept. Furthermore,

note that explicit mounting ports are missing in the specification technique [Thr05]. However, parts (see Section 5.2.4) can be considered to represent potential mounting ports implicitly. In the following, each port definition subclass is discussed individually in more detail.

Material ports As stated previously, material ports represent the labels B, Y, X of revised spatio-temporal components C (see Section 5.2.1). Consequently, material ports can be used for sensing material presence, interacting with present material, creating material, or removing material. Recall the revised binding histories $\vec{B} = \{B \to (V \times \mathbb{P}(\mathcal{O} \times \mathcal{I}) \times \mathbb{B})^\infty\}$ with the universe of (kinetic energy) output labels $\mathcal{O}$ and the universe of (kinetic energy) input labels $\mathcal{I}$, as well as the entry histories $\vec{Y} = \{Y \to (T \times \mathcal{C})^\infty\}$, the exit histories $\vec{X} = \{X \to V^\infty\}$, and the activation histories $\vec{A}(B \cup Y \cup X) = \{B \cup Y \cup X \to \mathbb{P}(\mathcal{C})^\infty\}$ with transformable collision space $(V, 0_V, \bowtie, \sqcup, T)$ and the universe of revised spatio-temporal components $\mathcal{C}$. Note that reading the material ports in $a \in \vec{A}(B \cup Y \cup X)$ returns a different result than writing the material ports in $b \in \vec{B}, y \in \vec{Y}, x \in \vec{X}$. In particular, reading material ports $\lambda \in B \cup Y \cup X$ at time step $n \in \mathbb{N}$ returns a set of components $a(\lambda)(n) \in \mathbb{P}(\mathcal{C})$. In contrast, writing, e.g., an exit label $\xi \in X$ at time stamp $n \in \mathbb{N}$ must provide a volume $x(\xi)(n) \in V$. This difference also is the reason why read and write types are distinguished in the general observation concept (see Section 5.1.1), from which ports, port definitions, and material ports are derived. Finally, the data model for representing material ports is shown in Figure 5.14.

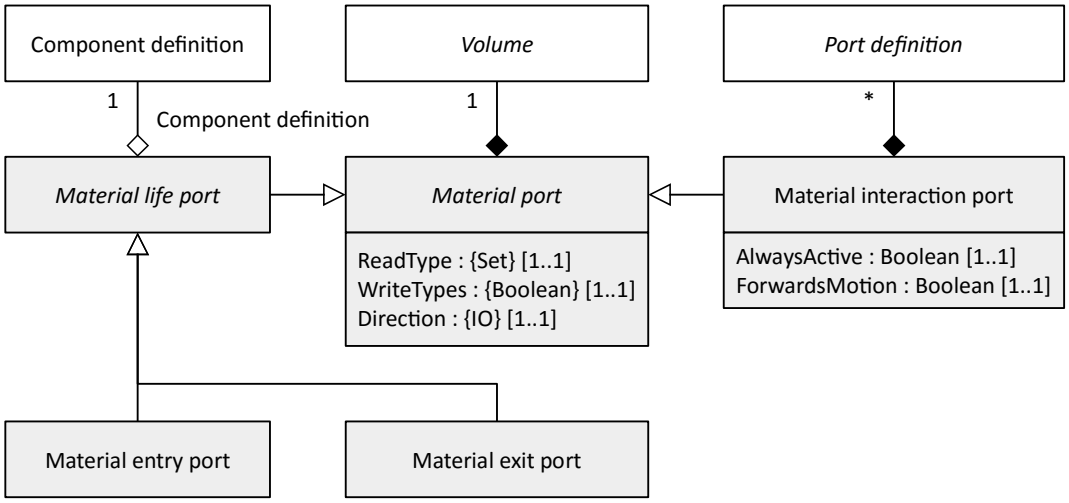


Figure 5.14: Abstract material port class and subclasses.

The abstract material port base class contains one *volume* $t(v) \in V$ (see Section 5.1.4). The volume $t(v)$ defines the position and orientation $t \in T$ as well as the spatial extent $v \in V$ of the material port. Note that the base volume v can be either atomic or composite here. Furthermore, the *read type* of material ports is limited to sets (of revised spatio-temporal components), while the *write types* are limited to boolean, and the *direction* is limited to input-output. Then, two subclasses are defined, namely the *material life ports* and the *material interaction ports*. Material life ports refer to a component definition $C' \in \mathcal{C}$ (see Section 5.2.1) and are subdivided further into *material entry ports* and *material exit ports*. Material entry ports represent the entry labels Y and can be used to create instances of the associated component definition C' with the defined volume transform t. An instance is created if *true* is written to the port. In the same time step, also a set holding this instance can be read from the port. In contrast, material exit ports represent the exit labels X and can be used to remove respective instances C' whose parts collide with $t(v)$. The instances are removed only if *true* is written to the port. At the same time, the removed instances can be read from the port. Finally, material interaction ports define an *always active* flag, a *forwards motion* flag, and (may) contain multiple subordinate *port definitions* I', I'_M, O', O'_M. Material interaction ports represent the binding labels B and can be used to interact with components $C' \in \mathcal{C}$ that provide the required syntactic interface I', I'_M, O', O'_M and whose parts collide with $t(v)$. If the port is always active or *true* is written to the port, upon collision as well as matching interfaces the port assignments are forwarded between the ports I', I'_M, O', O'_M of the material interaction port and the ports I', I'_M, O', O'_M of the component C'. Note that this behavior corresponds to the revised channel binding property (see Section 5.2.1). Furthermore, if the port forwards motion the transforms applied to the parent component C are applied also to the colliding component C' according to the motion forwarding property (again see Section 5.2.1). Consequently, revised spatio-temporal components can be glued together temporarily.

Energy ports Then, the energy ports represent the labels $I_E \subseteq I$, I_M, $O_E \subseteq O$, and O_M of revised spatio-temporal components (see Section 5.2.1). Energy ports can be used for moving components or exchanging other forms of energy between components. Recall the kinetic energy input/output histories $\vec{I}_M/\vec{O}_M = \{I_M/O_M \to T^\infty\}$ (see Section 5.2.1). Consequently, the channels $\lambda \in I_M/O_M$ allow revised spatio-temporal components to receive or produce kinetic energy in the form of transforms $t \in T$. Other forms of energy such as electric energy or thermal energy can be exchanged via the input and output observations I_E, O_E instead. Finally, the data model for representing energy ports as part of system specifications is depicted in Figure 5.15.

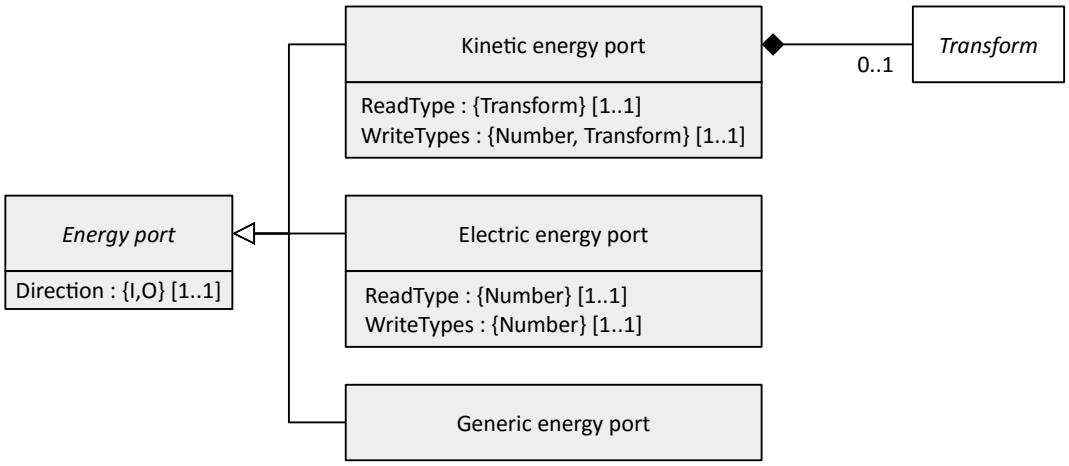


Figure 5.15: Abstract energy port class and subclasses.

The abstract base class for energy ports only permits the input and the output *directions*, while the input-output direction is not allowed. Then, three subclasses of energy ports are defined, namely *kinetic energy ports*, *electric energy ports*, and *generic energy ports*. Kinetic energy port represent the kinetic energy input and output channels I_M, O_M. The *read type* of kinetic energy ports is limited to transforms $t \in T$ (see Section 5.1.5), while the *write types* are limited to *number* and transforms alternatively. Furthermore, kinetic energy ports contain at most one *reference transform*. Writing a transform to a kinetic energy port causes this transform to be forwarded to the receiver components. Writing a number to a kinetic energy port, causes a multiple of the reference transform to be forwarded to the receiver components instead. In contrast, electric energy ports allow one to model the exchange of electric energy between revised spatio-temporal components. Both the *read* and the *write types* of electric energy ports are limited to *number*. A positive number represents energy surplus, while a negative number represents energy demand [HIKB12]. Finally, generic energy ports represent other forms of energy that are not covered by the previous port types. For example, one might want to consider the effect of thermal energy during workpiece processing. Typically, the interaction between workpiece and tool generates thermal energy that needs to be removed by means a cooling medium. At last, generic energy ports can be used also in case the concrete form of energy does not have to be decided during conceptual design. To support this flexibility, the *read* and *write types* of generic energy ports are not limited, but have to be identical.

Data ports Furthermore, data ports subsume the concepts of data input channel labels $I_D \subseteq I$ and data output channel labels $O_D \subseteq O$ of revised spatio-temporal components (see Section 5.2.1). Consequently, data ports can be used to model digital information exchange within the revised component formalism. Note that digital information exchange is implemented typically using software components. However, in principle also electronic components can be used. Again the *direction* of data ports is limited to input or output. Furthermore, the *read type* and the *write types* remain unlimited, but have to be identical. Finally, note that the physical representation of the data flow is not specified as opposed to, e.g., the approach of Thramboulidis [Thr05].

Generic ports Finally, generic ports subsume the concepts of generic input channel labels $I_G \subseteq I$ and generic output channel labels $O_G \subseteq O$ of revised spatio-temporal components (see Section 5.2.1). As stated previously, generic ports can be used to model interactions within the revised component formalism that are not covered by the previous port types. Consequently, one does not have to decide whether the interaction is implemented by means of data, energy, or material exchange. However, note that this decision has to be made at some point during the development process. Again, generic ports are limited to input and output *directions*. Furthermore, the *read* and *write types* are unlimited, but have to be identical.

Port references

At last, the concrete port reference class allows one to instantiate existing port definitions in another context. Hereby, the *read type*, the *write types*, and the *direction* of the port reference equals to the read type, the write types, and the direction of the associated port definition. Port references are used within component references, which have been introduced previously in Section 5.2.1. Both port references and component references allow us to reuse revised spatio-temporal component definitions within conceptual design documents. As stated previously, component reuse is an important factor in mechatronic system development [Thr05]. In particular, component reuse allows one to reduce development effort while increasing system quality.

5.2.3 Channels

Based on the port concept (see Section 5.2.2) the channel concept is introduced. Channels represent one means to specify the interaction between components (in addition to spatial collision and dynamic binding). Technically, channels represent directed connections between ports. Note that the revised channel concept differs a bit from the channel

concept of the FOCUS theory, which is more similar to the proposed observation concept (see Section 4.1.2). The data model for representing channels is depicted in Figure 5.16.

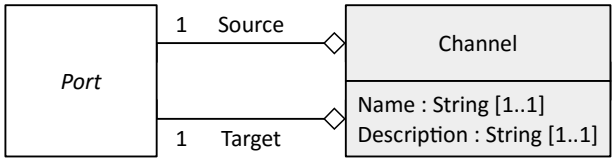


Figure 5.16: Abstract channel class and subclasses.

The concrete channel class defines a human-readable *name* and a *description* for documentation purposes. Then, each channel refers to one *source* and one *target* port. Hereby, both the source and the target port can be port definitions or port reference. Consequently, also the interaction between component definitions and component references (see Section 5.2.1) can be specified. However, source and target ports must not refer to material ports or to port references referencing material ports. The reason for this limitation is that material ports are input-output ports, whose values are determined by the underlying computation engine based on spatial collision. Finally, during execution channels transfer the values from the source to the target port as specified by the revised composition operator $\odot$ (see Section 5.2.1).

5.2.4 Parts

Then, based on the volume concept (see Section 5.1.4) the part concept is introduced. The proposed part concept corresponds to the part labels P of revised spatio-temporal components (see Section 5.2.1). Recall the part histories $\vec{P} : P \to V^{\infty}$. Note that the theory supports specifying parts with changing volumes over time. In contrast, the part concept only supports specifying constant shapes, positions and orientations. The data model for representing parts during conceptual design is shown in Figure 5.17.

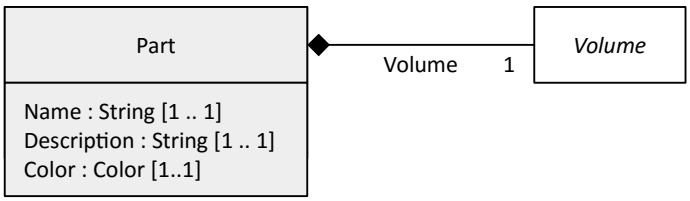


Figure 5.17: Concrete part class.

The concrete part class defines a human-readable *name* and a *description* for documentation purposes. Furthermore, one must specify a *color* for each part representing its material (e.g. concrete or steel). Note that alternatively more complex material specifications could be used [GBC+02]. However, the proposed approach relies on color for simplicity. Finally, each part contains one *volume* $t(v) \in V$ with transform $t \in T$ and base volume $v \in V$, where volume v can be either atomic or composite. Note that the transformed volume $t(v)$ is specified independent of state and time. Consequently, the spatial extent of parts, as proposed in the concrete data model, cannot be changed during execution of revised spatio-temporal components. However, the position and orientation of part volumes can be changed implicitly through transmission of kinetic energy to superordinate components (see Sections 5.2.1 and 5.2.2).

5.2.5 Behaviors

Finally, based on the concept of executables (see Section 5.1.2), the concept of revised spatio-temporal components (see Section 5.2.1), and the concept of parts (see Section 5.2.4) the concept of behaviors is defined in the following. Behaviors extend the concept of executables and are intended to describe the reaction of revised spatio-temporal components C over $I, I_M, O, O_M, P, B, Y, X$ to stimuli from their environment. Now, recall that executables represent state transition systems (T, s_0, v_0) with state transition function $T : (S \times \bar{I} \times \bar{V})^* \rightarrow \mathbb{P}(S \times \bar{O} \times \bar{V})$, state labels S, input channel labels I, variable labels V, output channel labels O, initial state label $s_0 \in S$, and initial variable assignment $v_0 \in \bar{V}$. Behaviors additionally include the kinetic energy input channel assignments $\bar{I}_M$ and the (material port) activation assignments $\bar{A}(B \cup Y \cup X) = \{B \cup Y \cup X \rightarrow \mathbb{P}(\mathcal{C})\}$ into their stimulus as well as the kinetic energy output channel assignments $\bar{O}_M$, the part assignments $\bar{P} = \{P \rightarrow V\}$, the binding assignments $\bar{B} = \{B \rightarrow V \times \mathbb{P}(\mathcal{O} \times \mathcal{I}) \times \mathbb{B}\}$ with the universe of (kinetic energy) output labels $\mathcal{O}$ and the universe of (kinetic energy) input labels $\mathcal{I}$, the entry assignments $\bar{Y} = \{Y \rightarrow T \times \mathcal{C}\}$, and the exit assignments $\bar{X} = \{X \rightarrow V\}$ into their reaction. Consequently, the state transition function T is extended to $T' : (S \times \bar{I} \times \bar{I}_M \times \bar{A}(B \cup Y \cup X) \times \bar{V})^* \rightarrow \mathbb{P}(S \times \bar{O} \times \bar{O}_M \times \bar{P} \times \bar{B} \times \bar{Y} \times \bar{X} \times \bar{V})$. Note that the state transition function T' has to respect the revised collision sensing property during computation (see Section 5.2.1). Consequently, the (material port) activation assignments $a \in \bar{A}(B \cup Y \cup X)$ must reflect correctly the binding assignments $b \in \bar{B}$, entry assignments $y \in \bar{Y}$, and exit assignments $x \in \bar{X}$. Furthermore, note that the part assignments $p \in \bar{P}$ allow one to model a variable spatial extent of revised spatio-temporal components. Finally, the data model for representing the behavior of revised spatio-temporal components is depicted in Figure 5.18.

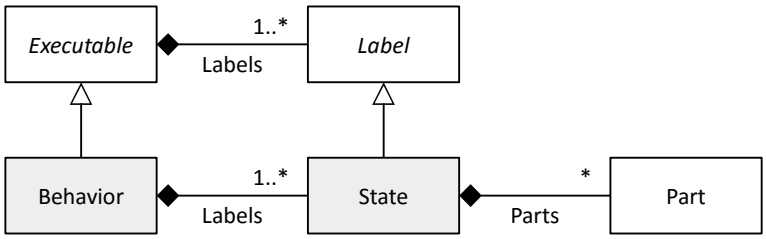


Figure 5.18: Concrete behavior class.

The concrete behavior class extends the abstract executable class and contains concrete *states* instead of abstract *(state) labels*. The concrete state class itself extends the abstract label class and contains *parts*, which were not support in the original STEM approach (see Section 4.2 and [Hum11]). Note that still part objects are specified independent of time as explained in the previous section. Consequently, changing the spatial extent of revised spatio-temporal components has to be modeled using multiple behavioral states $s \in S$ with individual part objects $p \in P$, which – in turn – have their own time-independent volumes $t(v) \in V$ with transformation $t \in T$ and base volume $v \in V$ (see Section 5.1.4). Note that the proposed model is suitable for describing few distinct spatial configurations such as input, output, and intermediate material states (see Section 3.1.1 and Section 3.1.2). In contrast, describing a greater number of distinct spatial configurations can become cumbersome quickly. For that purpose, the models behind flexible multibody dynamics [Sha97], computational fluid dynamics [Roa76], and movable cellular automata [PHO+01] as well as parametric representations [Kul08] might be more suitable. However, for the purpose of conceptual design the selected approach should be sufficient in many cases. In contrast, during the transition to detailed design the other models might prove more useful.

5.3 Added concepts

In the following, concepts are described that did not exist in the original STEM theory (see Section 4.2), but which have been added to cover the needs of the test-driven design method (see Chapter 3). In particular, concepts have been added for capturing requirement, manufacturing process, and test specifications (see Sections 3.1.1, 3.1.2, and 3.1.3). First, the concept of *requirements* is introduced in Section 5.3.1 before turning to the concept of *properties* (see Section 5.3.2) and *monitors* in Section 5.3.3. Finally, the concept of *scenarios* is described in Section 5.3.4.

5.3.1 Requirements

First, means are added for documenting requirements as part of the revised spatio-temporal components (see Section 5.2.1). In principle, a wide variety of approaches exists for capturing and structuring the information of requirement specifications. For example, requirement templates can be used to ensure a certain degree of completeness [TMR13]. However, for simplicity only basic means are provided for representing requirements in terms of natural language statements. The data model for representing requirements is shown in Figure 5.19.

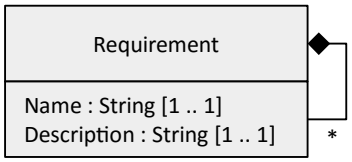


Figure 5.19: Concrete requirement class.

The concrete requirement class only defines a human-readable *name* and a *description*. The name can be used for identifying the requirement. For example, the name can be used in other engineering artifacts to establish cross-references. The description represents the actual content of the requirement in natural language. Furthermore, requirements can be decomposed into multiple *child* requirements. Consequently, larger requirement specifications can be structured hierarchically. Finally, note that the requirement traceability problem is not considered here [GF94].

5.3.2 Properties

Based on the concept of observations (see Section 5.1.1), the concept of expressions (see Section 5.1.3), and the concept of revised spatio-temporal components (see Section 5.2.1) the concept of properties is introduced. Properties essentially represents constraints over the input and output observations of revised spatio-temporal components C over $I, I_M, O, O_M, P, B, Y, X$. Note that properties have been added to the modeling technique to support the formalization of requirements (see Section 5.3.1). Formally, properties are modeled as boolean expressions $E : (S \times \bar{I} \times \bar{I}_M \times \bar{A}(B \cup Y \cup X) \times \bar{O} \times \bar{O}_M \times \bar{V})^* \to \mathbb{B}$ with state labels S, input channel assignments $\bar{I}$, kinetic energy input channel assignments $\bar{I}_M$, (material port) activation assignments $\bar{A}(B \cup Y \cup X) = \{B \cup Y \cup X \to \mathbb{P}(\mathcal{C})\}$, output channel assignments $\bar{O}$, kinetic energy output channel assignments $\bar{O}_M = \{O_M \to T\}$, and variable assignments $\bar{V}$. In particular, property expressions are able to access

the material port activations $a \in \bar{A}(B \cup Y \cup X)$, e.g., for constraining the position of generated components. A finite stream of state labels, channel assignments, activation assignments, and variable assignments is considered to be accepted by the property if the expression evaluates to the boolean value *true* (see Section 6.2.1). Note that properties can be attached to components directly (see Section 5.2.1) or to the *activities* of *monitors* (see Section 5.3.3) as well as the *steps* of *scenarios* (see Section 5.3.4). In the first case the state labels S and the variable assignments $\bar{V}$ can be omitted in the previous definition. In contrast, in the latter two cases monitors and scenarios represent executables and, hence, the state labels S and the variable assignments $\bar{V}$ are defined. Finally, the data model for representing properties in a machine-interpretable format is depicted in Figure 5.20.

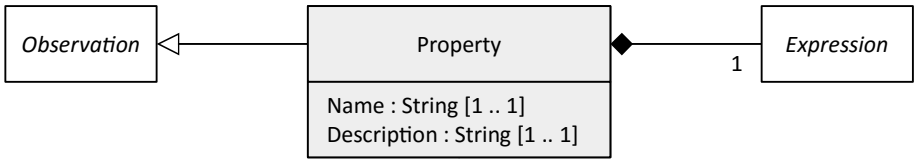


Figure 5.20: Concrete property class.

The concrete property class is derived from the abstract observation class and defines a human-readable *name* and a *description*. The name can be used to reference properties across engineering artifacts, while the description can be used for documentation purposes. Then, each property contains a boolean *expression* (see Section 5.1.3). The expression determines the property value for each data stream $ds \in (S \times \bar{I} \times \bar{I}_M \times \bar{A}(B \cup Y \cup X) \times \bar{O} \times \bar{O}_M \times \bar{V})^n$ with stream length $n \in \mathbb{N}$, entry index $k \in \mathbb{N}$ such that $0 \leq k \leq n$, stream entry $ds_n(k) = (s_k, i_k, i_{M,k}, a_k, o_k, o_{M,k}, v_k)$, state accessor $s(ds_n, k) = s_k$, input channel accessor $i(ds_n, k) = i_k$, kinetic energy input channel accessor $i_M(ds_n, k) = i_{M,k}$, (material port) activation accessor $a(ds_n, k) = a_k$, output channel accesor $o(ds_n, k) = o_k$, kinetic energy output channel accessor $o_M(ds_n, k) = o_{M,k}$, and variable accessor $v(ds_n, k) = v_k$. For example, some property expression E_1 could state that the assignment of some electric energy output port $\omega \in O_E \subseteq O$ must not exceed a predefined limit $l \in \mathbb{R}$, i.e.

$$E_1(ds_n) \Leftrightarrow |o(ds_n, n)(\omega)| \leq l.$$

Similarly, some property expression E_2 might require that material cannot be observed at a particular location $\beta \in B$, i.e.

$$E_2(ds_n) \Leftrightarrow |a(ds_n, n)(\beta)| = 0.$$

Note that the property expressions are evaluated in every computation step $n \in \mathbb{N}$. Consequently, the constraints implied by the property expressions are imposed onto each state of the superordinate revised spatio-temporal component. Essentially, properties allow one to model invariants over certain data streams ds, which include the stimuli and reactions of components as well as information about their internal state. Finally, more information about the evaluation of properties during test execution and the impact of the evaluation on the test result can be found in Section 6.2.1.

5.3.3 Monitors

Then, based on the concept of executables (see Section 5.1.2), the concept of revised spatio-temporal components (see Section 5.2.1), and the concept of properties (see Section 5.3.2) the concept of monitors is defined. Monitors provide the second means for formalizing the requirements a revised spatio-temporal component C over label sets $I, I_M, O, O_M, P, B, Y, X$ must satisfy. In particular, monitors are more powerful than properties, but also yield more verbose models. Furthermore, monitors can be used to model the manufacturing processes, that need to be implemented by component C, in an intuitive manner. Technically, monitors extends the concept of executables. Recall that executables represent state transition systems (T, s_0, v_0) with state transition function $T : (S \times \bar{I} \times \bar{V})^* \rightarrow \mathbb{P}(S \times \bar{O} \times \bar{V})$, state labels S, input channels labels I, output channel labels O, variable labels V, initial state label $s_0 \in S$, and initial variable assignment $v_0 \in \bar{V}$. Monitors additionally define property labels P_M (see Section 5.3.2) and replace the state transition function T through $T' : (S \times \bar{I} \times \bar{I}_M \times \bar{A}(B \cup Y \cup X) \times \bar{O} \times \bar{O}_M \times \bar{V})^* \rightarrow \mathbb{P}(S \times \bar{P}_M \times \bar{V})$ with input channel assignments $\bar{I}$, kinetic energy input channel assignments $\bar{I}_M$, (material port) activation assignments $\bar{A}(B \cup Y \cup X) = \{B \cup Y \cup X \rightarrow \mathbb{P}(\mathcal{C})\}$, output channel assignments $\bar{O}$, kinetic energy output channel assignments $\bar{O}_M$, variable assignments $\bar{V}$, and property assignments $\bar{P}_M = \{P_M \rightarrow \mathbb{B}\}$. Consequently, monitors read both the stimuli and reactions of revised spatio-temporal component C and derive properties P' that must be satisfied in each computation step $n \in \mathbb{N}$. In particular, the monitors are able to access the (material port) activations $a \in \bar{A}(B \cup Y \cup X)$, e.g., for constraining the position of generated components, which is support also by properties. Note that, in contrast to properties, monitors are able to track generated components during a monitor computation by mean of the variable assignments $\bar{V}$. A finite stream of state labels, channel assignments, activation assignments, and variable assignments is considered to be accepted by the monitor if the monitor computation does not cause any property violations, which is formalized in the following chapter (see Section 6.2.1). Finally, the monitor data model is provided in Figure 5.21.

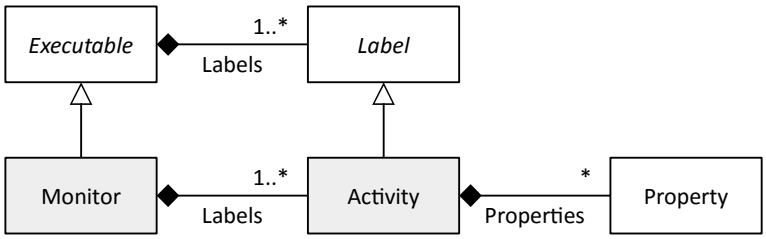


Figure 5.21: Concrete monitor class.

The concrete monitor class extends the abstract executable class and contains concrete *activities* instead of abstract *(state) labels*. The concrete activity class extends the abstract label class and may contain multiple *properties*. During execution, the monitor switches activities based on the stimuli and reactions of the revised spatio-temporal component C as well as the respective variable assignment stream $v \in \bar{V}$. Furthermore, after each computation step the monitor evaluates the expressions associated with the properties of the current activity. If at least one property is violated, the entire monitor is considered to be violated. Consequently, the behavior of the revised spatio-temporal component is not consistent with the monitor specification (see Section 6.2). Note that monitors can be used for requirements or manufacturing process formalization [HCL+14] depending on whether design decisions are included or not. For example, design decisions might include activities, which have not been specified by the customer.

5.3.4 Scenarios

Finally, based on the concept of executables (see Section 5.1.2), the concept of revised spatio-temporal components (see Section 5.2.1), and the concept of properties (see Section 5.3.2) the concept of scenarios is defined. Scenarios can be used to describe the test cases, which are used to *verify* revised spatio-temporal components C over $I, I_M, O, O_M, P, B, Y, X$. Technically, scenarios extend the concept of executables. Again, recall that executables represent state transition systems $(, s_0, v_0)$ with state transition function $T : (S \times \bar{I} \times \bar{V})^* \to \mathbb{P}(S \times \bar{O} \times \bar{V})$, state labels S, input channels labels I, output channel labels O, variable labels V, initial state label $s_0 \in S$, and initial variable assignment $v_0 \in \bar{V}$. Scenarios additionally define custom entry labels Y_S, custom exit labels X_S, property labels P_S, and a final state $s_f \in S$ such that the state transition function T is extended to $T' : (S \times \bar{A}(Y_S \cup X_S) \times \bar{A}(B \cup Y \cup X) \times \bar{O} \times \bar{O}_M \times \bar{V})^* \to \mathbb{P}(S \times \bar{I} \times \bar{I}_M \times \bar{Y}_S \times \bar{X}_S \times \bar{P}_S \times \bar{V})$ with custom (material port) activation assignments $\bar{A}(Y_S \cup X_S) = \{Y_S \cup X_S \to \mathbb{P}(\mathcal{C})\}$, (material port) activation as-

signments $\bar{A}(B \cup Y \cup X) = \{B \cup Y \cup X \to \mathbb{P}(\mathcal{C})\}$, output channel assignments $\bar{O}$, kinetic energy output channel assignments $\bar{O}_M$, input channel assignments $\bar{I}$, kinetic energy input channel assignments $\bar{I}_M$, custom entry assignments $\bar{Y}_S = \{Y_S \to (T \times \mathcal{C})\}$, custom exit assignments $\bar{X}_S = \{X_S \to V\}$, variable assignments $\bar{V}$, and property assignments $\bar{P}_S = \{P_S \to \mathbb{B}\}$. Consequently, scenarios are able to read the output and input-output ports as well as write the input ports of revised spatio-temporal components. Furthermore, scenarios may generate and remove revised spatio-temporal components during execution via the custom entry ports Y_S and custom exit ports X_S. Hereby, the custom entry and exit ports must be activated properly according to the revised collision sensing property (see Section 5.2.1). Finally, during execution scenarios evaluate the all ports of revised spatio-temporal components C, the custom entry and exit ports, and the variables using the properties P_S. A scenario execution is considered to be successful if it reaches the final state s_f without causing any property violations. Consequently, scenarios can be thought of modeling the behavior of the environment of a revised spatio-temporal component including accepting and rejecting states. Finally, the data model for describing scenarios as part of system specifications is provided in Figure 5.22.

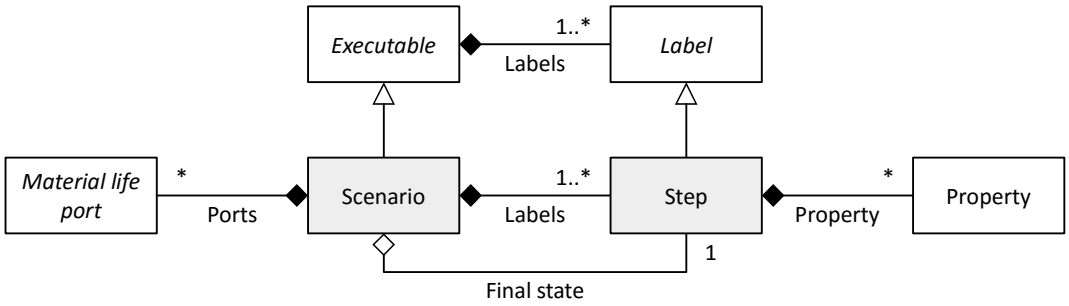


Figure 5.22: Concrete scenario class.

The concrete scenario class extends the abstract executable class and contains *material life ports* (see Section 5.2.2), concrete *steps* instead of abstract *(state) labels*, and one *final step*. The concrete step class extends the abstract label class and may contain multiple *properties* (see Section 5.3.2), which act similar to *assertions* in classical unit testing frameworks [Bec02]. The material life ports represent the custom entry labels Y_S and the custom exit labels X_S, while the final step represents the variable s_f. During execution the scenario switches between the steps depending on the transitions and their guards respectively. Hereby, the guards read the input-output and output ports of the revised spatio-temporal component. In contrast, the actions write the input ports of the revised spatio-temporal component as well as the custom entries $\upsilon_S \in Y_S$, exit

labels $\xi_S \in X^*$, and variables $\lambda \in V$. Consequently, in every time step the scenario may create and remove revised spatio-temporal components via the custom material life ports Y_S and X_S. Additionally, in every time step the properties $\pi_S \in P_S$ associated with the current step object are evaluated. If at least one property is violated, the revised spatio-temporal component behavior is considered to be inconsistent with the scenario specification. A typical test case creates input material at some entry port and waits for the material to appear in a modified state at some exit port within a limited duration. Hereby, the duration limit can be expressed as a property of the respective wait step (see the industry-close showcase in Chapter 8). Finally, more information about the formal semantics of test execution is provided in Section 6.2.

5.4 Summary and outlook

This chapter introduced an integrated modeling technique for capturing knowledge about customer requirements, manufacturing processes, test cases, discipline-specific interfaces, and component reuse. Therefore, basic concepts were explained that are reused across many aspects of the modeling technique (see Section 5.1). Then, concepts were described that have been taken from the original STEM theory (see Section 4.2) and have undergone a major revision to fit the particular needs (see Section 5.2). Finally, concepts were introduced that have been added to the original STEM theory (see Section 5.3) to cover the aspects considered by the test-driven design method proposed in Chapter 3. Subsequently, the next chapter describes quality issues that can be evaluated automatically over the presented modeling technique.

6 Quality issues

After having introduced the revised formalism and the concrete data model for capturing design knowledge in Chapter 5 a systematic view onto the quality issues is developed, which might arise during conceptual design and test-based verification. According to Lindland et al. [LSS94] *syntactic* and *semantic* quality issues are distinguished in the following. The authors additionally define *pragmatic* quality issues, which are neglected here because they are much harder to evaluate automatically. Subsequently, Figure 6.1 provides an overview of the quality issues that are supported by the proposed technique.

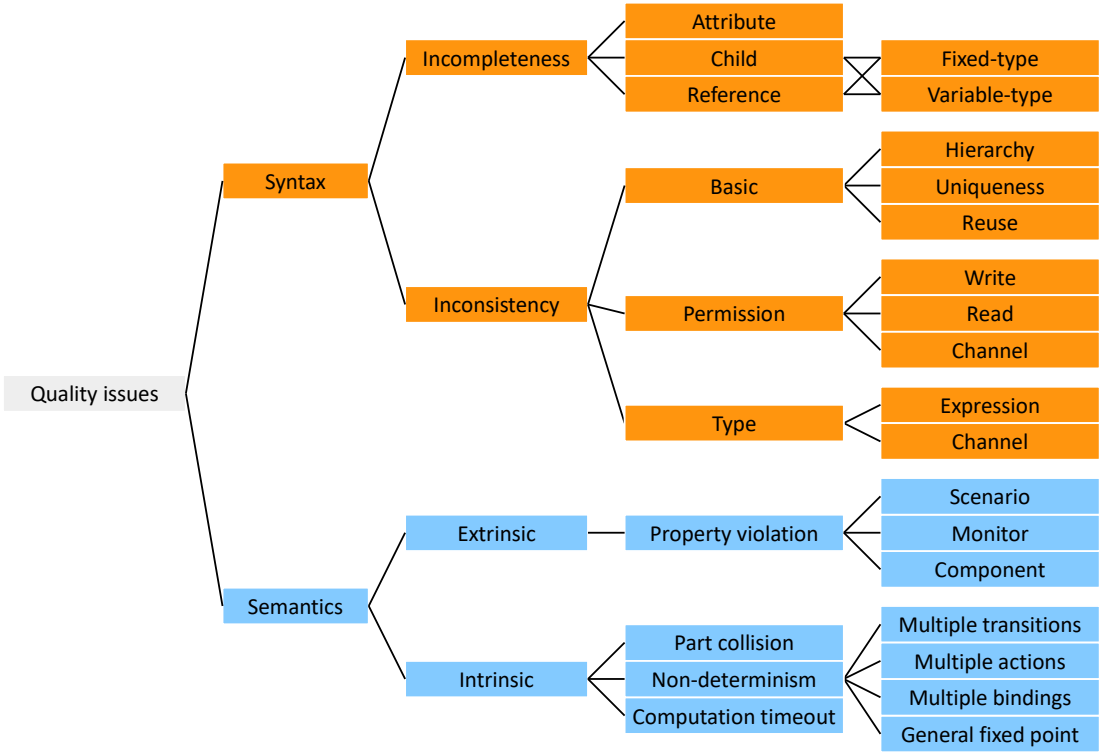


Figure 6.1: Overview of the quality issues that might arise during conceptual design.

In the following, first the syntactic quality issues are described in Section 6.1 before the semantic quality issues are explained in Section 6.2. In particular, the classification schemes are introduced, which have been selected for each category of issues. Furthermore, a precise and unambiguous definition is provided for each issue, such that the issue detection can be implemented by potential tool vendors easily.

6.1 Syntactic issues

According to Lindland et al. [LSS94], syntactic issues represent violations of the language rules. In the given case, the "language" is defined in terms of classes of objects, their attributes, and their relations to other classes of objects as introduced in the previous chapter (see Chapter 5). In particular, in the following attribute and relationship rules are introduced, whose violation leads to syntactic quality issues. Furthermore, according to Mohagheghi et al. [MDN09] *incompleteness* and *inconsistency* issues are distinguished. Note that the authors also define *correctness*, *comprehensibility*, and *changeability* issues. However, syntactic correctness is neglected because this goal is more suitable for grammar-based languages than object-based modeling techniques. Furthermore, comprehensibility and changeability are neglected because these goals are much harder to assess automatically than completeness and consistency.

In the following, first the incompleteness issues are described in Section 6.1.1 that are supported by the proposed approach. Then, the consistency rules and respective inconsistency issues are detailed in Section 6.1.2. Finally, for each of the two categories a specific way of describing the underlying syntactic rules and their relation to the modeling technique from the previous chapter are used.

6.1.1 Incompleteness issues

As mentioned previously, the proposed modeling technique (see Chapter 5) consists of UML (see Section 2.1.1) *classes* and *attributes* as well as *composition* and *aggregation* relations. Note that compositions are relations between classes with filled diamond shape ending, while aggregations are relations between classes with unfilled diamond shape ending. Furthermore, compositions define *parent-child* relationships between objects of the associated classes with the class next to the filled diamond shape ending representing the *parent object* and the other class representing the *child object*. In contrast, aggregations define *general* relationships between objects of the associated classes with the class next to the unfilled diamond shape representing the *referencer object* and the other class representing the *referenced object*. Incompleteness issues appear in case mandatory attributes, compositions, or aggregations are missing. Consequently, *attribute*, *child*,

and *reference* incompleteness issues are distinguished. Note that whether an attribute, composition, or aggregation is mandatory depends on its multiplicity.

In the following, the attribute incompleteness issue are described before explaining the child incompleteness issue and introducing the reference incompleteness issue. Furthermore, UML (see Section 2.1.1 and [BJR+96]) *specification patterns* are used to describe the concepts of the modeling technique from the previous chapter and situations, for which the respective issues appear.

Attribute incompleteness

The attribute incompleteness issue is based on the *required attribute* UML specification pattern. The required attribute specification pattern is depicted in Figure 6.2. The pattern states that every instance of class `Class A` should define exactly one (non-null) value for attribute `Attribute` of type `Type`. Note that the modeling technique from the previous chapter only uses the `Boolean`, `Number`, and `String` attribute types, while in principle much more data types are supported. Instances of the class `Class A` that define an appropriate value for attribute `Attribute` are said to *conform* to the required attribute pattern, while instances of the class `Class A` that do not define an appropriate value for the attribute `Attribute` are said to *violate* the required attribute pattern.

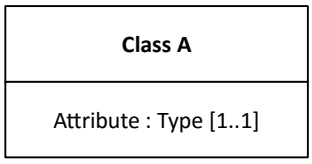


Figure 6.2: UML specification pattern for required attributes.

Examples for this specification pattern are the *name* of components (see Section 5.2.1) or the *value* of constant number expressions (see Section 5.1.3). If required attributes are not defined, the attribute incompleteness issue is raised. As soon as the user defines the value of the attribute, the attribute incompleteness issue disappears. Note that some attributes define a default value such that respective attribute incompleteness issues might never appear during development even though the attribute is mandatory.

Child incompleteness

In contrast, the child incompleteness issue is based on the *required fixed- or variable-type child* UML specification pattern. The required fixed- or variable-type child specification

pattern is shown in Figure 6.3. Note that the parent-child relationship is expressed using a UML composition association with multiplicity `Multiplicity M` (e.g. "between $m \in \mathbb{N}$ and $n \in \mathbb{N}$ with $m \leq n$" or "greater $k \in \mathbb{N}$"). The pattern states that every instance of the class `Class A` should have as many associated child instances of the class `Class B` as prescribed by the multiplicity `Multiplicity M`. In the fixed-type case of the pattern the concrete class `Class B` must be instantiated directly, while in the variable-type case some subclass of the abstract class `Class B` must be used instead. Instances of the class `Class A` with an adequate number of associated child objects are said to *conform* to the pattern, while instances of the class `Class A` with an inadequate number of associated child objects are said to *violate* the pattern.

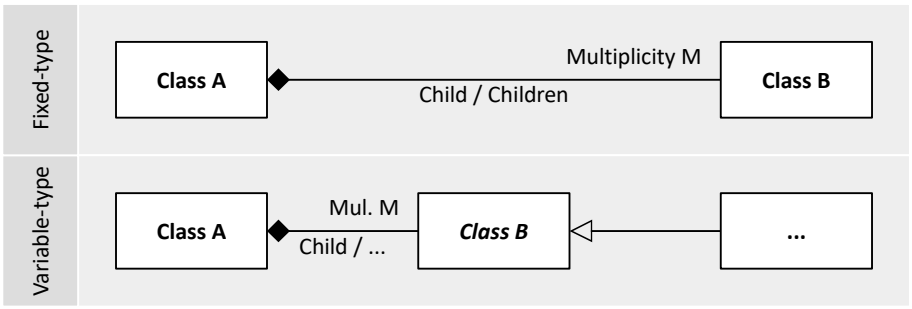


Figure 6.3: UML specification pattern for required fixed- or variable-type children.

Examples for this specification pattern are the *argument expression* of unary expressions (see Section 5.1.3) or the *expression* of transition guards (see Section 5.2.5). The variable- or fixed-type child incompleteness issue is raised when an adequate number of child object is missing and disappears as soon as sufficiently many child objects are added. Note that in principle fixed-type child incompleteness issues can be resolved automatically, while variable-type child missing defects require user intervention. This difference is also the reason why the two cases are distinguished.

Reference incompleteness

Finally, the reference incompleteness issue is based on the *required fixed- or variable-type reference* UML specification pattern. The required fixed- or variable-type reference specification pattern is depicted in Figure 6.4. Note that reference relationships are expressed using UML aggregation associations with multiplicity `Multiplicity M` (e.g. "between $m \in \mathbb{N}$ and $n \in \mathbb{N}$ with $m \leq n$" or "greater $k \in \mathbb{N}$") as opposed to UML composition associations for parent-child relationships. The pattern states that every

instance of the class `Class A` should have as many associated instances of the class `Class B` as prescribed by the multiplicity `Multiplicity M`, while both instances do not have to be in a parent-child relationship. Also, the fixed- and the variable-type cases are distinguished depending on whether class `Class B` represents a concrete or an abstract class. Instances of the class `Class A` with an adequate number of associated referenced objects are said to *conform* to the pattern, while instances of the class `Class A` with and inadequate number of associated referenced objects are said to *violate* the pattern.

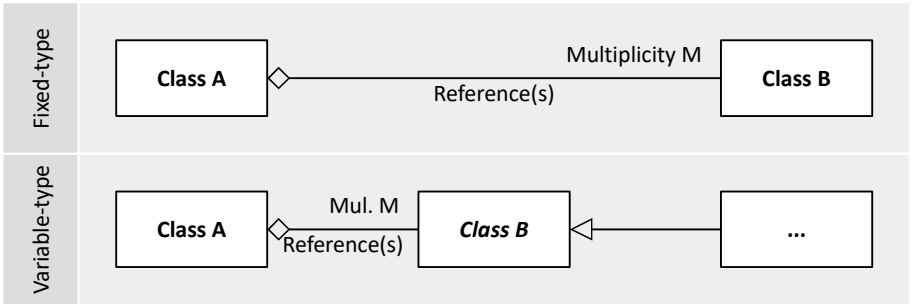


Figure 6.4: UML specification pattern for required fixed- and variable-type references.

Examples for this specification pattern are the *observation* object referenced by observation expressions (see Section 5.1.3) or the *source* and *target port* objects referenced by channels (see Section 5.2.3). The fixed- or variable-type reference incompleteness issue is raised when an adequate number of object references is missing and disappears as soon as an adequate number of object references is added. Note that in the general case both the fixed- and the variable-type reference incompleteness issues cannot be resolved automatically, but require user intervention.

6.1.2 Inconsistency issues

While the incompleteness issues (see Section 6.1.1) only state that certain design information is missing, they do not check the plausibility of the design information. At this point, the inconsistency issues come into play. According to Mohagheghi et al. [MDN09] inconsistencies refer to contradictions in the design knowledge, i.e. the existence of two independent, but contradictory statements. Note that inconsistencies directly reveal implausible design information because at least one of the statements of the design knowledge corpus cannot be valid. Three types of inconsistencies are distinguished, namely *basic*, *permission*, and *type* inconsistencies each addressing a different aspect of the modeling technique from the previous chapter.

Instead of UML (see Section 2.1.1 and [BJR+96]) specification patterns a custom constraint notation is used for specifying the consistency rules. The custom constraint notation is based on the hierarchical structure of the design objects, their class and attribute annotations, as well as their cross relations, which are depicted schematically in Figure 6.5. Note that a number of different cross relations are distinguished in the figure (e.g. *descendant-ancestor* or *nephew-uncle* associations), but the list is not complete.

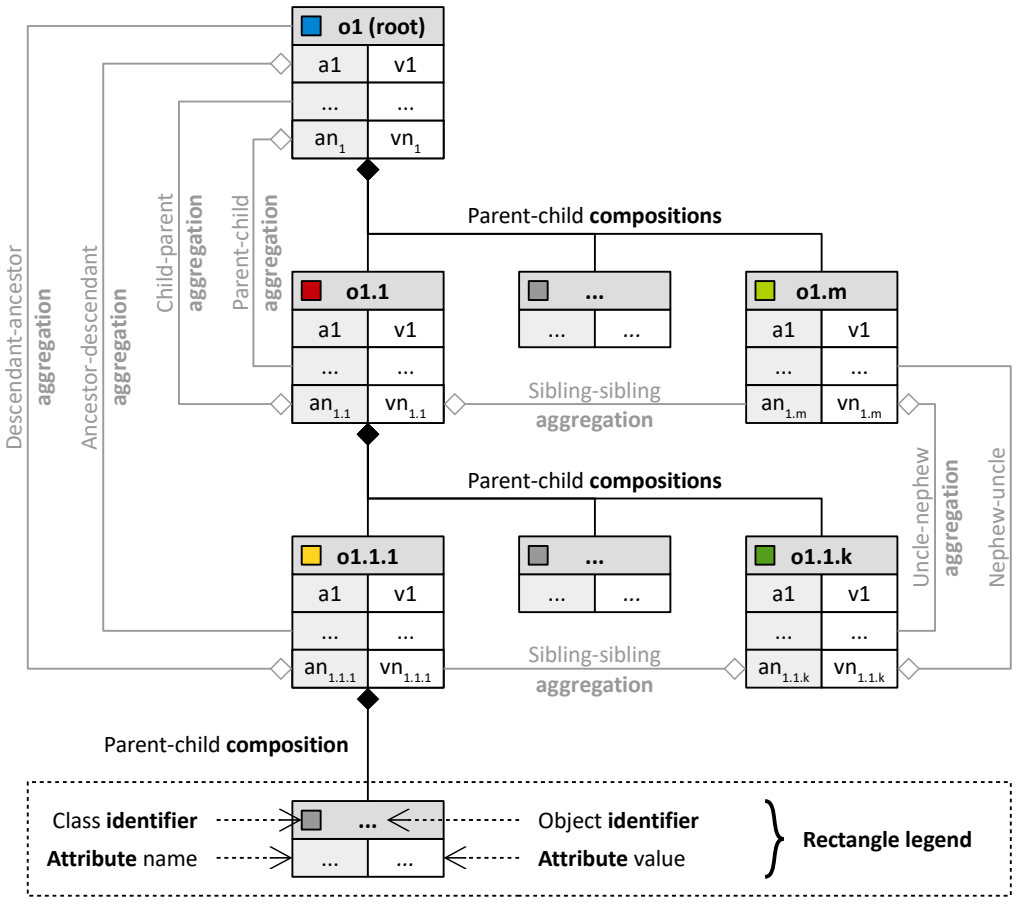


Figure 6.5: Hierarchical structure of the design objects with class and attribute annotations as well as cross references.

In the following, first the mathematical foundation of the custom constraint notation is described, before introducing the constraint notation itself. Then the basic inconsis-

tency issues are explained before elaborating on the permission inconsistency issues and introducing the type inconsistency issues.

Mathematical foundation

The mathematical foundation comprises a hierarchical structure of classes, the instantiation of these classes in terms of design objects, the hierarchical structure of the design objects, the cross references between the design objects, as well as the attributes of the design objects.

The core of the mathematical foundation is given by the set of classes $Classes$, which have been introduced in Chapter 5. These classes are connected by an inheritance hierarchy (e.g. the classes *Port* and *Variable* both inherit from the class *Observation*). For representing this inheritance hierarchy, the mapping $SuperClass : Classes \rightarrow Classes \cup \{\bot\}$ is defined. Given the relation

$$SuperClass(c) = c'$$

the class $c \in Classes$ is called the *sub*class and the class $c' \in Classes$ the *super*class. Furthermore, if $c' = \bot$ holds, class c does not have a superclass and, hence, represents a *base*class. Then, the direct subclasses of a given superclass $c \in Classes$ can be derived using the mapping $DirectSubClasses : Classes \rightarrow \mathbb{P}(Classes)$ such that

$$DirectSubClasses(c) = \{c' \in Classes \mid SuperClass(c') = c\}.$$

Finally, based on the $DirectSubClasses$ mapping the set of all subclasses both direct and indirect of a given superclass $c \in Classes$ can be derived using the mapping $SubClasses : Classes \rightarrow \mathbb{P}(Classes)$ such that

$$SubClasses(c) = \bigcup_{c' \in DirectSubClasses(c)} \{c'\} \cup SubClasses(c').$$

Hereby, it is required that a class $c \in Classes$ cannot be a subclass of itself, i.e. $c \notin SubClasses(c)$. Consequently, the $SuperClass$ mapping defines an acyclic, forest-like, hierarchical structure over the classes $c \in Classes$.

Based on the set of classes $Classes$ the set of design objects $Objects$ can be introduced. The design objects $o \in Objects$ represent instances of the classes $c \in Classes$. The instance relationship between classes and design objects is given by the mapping $Constructor : Objects \rightarrow Classes$. Given the instance relation

$$Constructor(o) = c$$

the class c is called the class of design object o. Then, based on the $Constructor$ mapping all design objects can be derived that belong to a given class. Therefore, the mapping $Instances : Classes \rightarrow \mathbb{P}(Objects)$ is introduced such that

$$Instances(c) = \{o \in Objects : Constructor(o) \in \{c\} \cup SubClasses(c)\}.$$

Note that the $Instances$ mapping uses the transitive $SubClasses$ relationship. Consequently, each design object o may belong either to the class c directly, or to any of the direct or indirect subclasses of class c.

Next, the *parent-child* composition relation and, thus, the hierarchical structure among the design objects is formalized. First, the mapping $Parent : Objects \rightarrow Objects$ is introduced establishing a parent-child relationship between different design objects. Given design objects $o, o' \in Objects$ with

$$Parent(o) = o'$$

design object o is called the *child* and design object o' the *parent*. Furthermore, in case $o = o'$ the object o respectively o' is called the *root* of the hierarchical structure. In practice, the root design object typically is a *workspace* or *project* object. Then, based on the $Parent$ mapping the mapping $Children : Objects \rightarrow \mathbb{P}(Objects)$ is defined returning all child design objects for a given parent design object $o \in Objects$, i.e.

$$Children(o) = \{o' \in Objects \mid Parent(o') = o\}.$$

Furthermore, based on the $Children$ mapping the mapping $Descendants : Objects \rightarrow \mathbb{P}(Objects)$ can be introduced finding all *descendant* design objects for a given *ancestor* design object $o \in Objects$, i.e.

$$Descendants(o) = \bigcup_{o' \in Children(o)} \{o'\} \cup Descendants(o').$$

Consequently, the $Descendants$ mapping calculates the transitive hull over the $Children$ relation and, hence, the $Parent$ relation. Furthermore, a design object $o \in Objects$ is required to not being part of its own descendants relation, i.e. $o \notin Descendants(o)$. As a result, the $Parent$ mapping defines an acyclic, tree-like, hierarchical structure over the design objects.

However, as noted previously, also cross references between design objects spanning the hierarchical structure are supported. In particular, *named* cross references are included in the approach. Therefore, first the set of reference names $ReferenceNames$ needs to

be introduced. Then, cross references themselves are given by the mapping $Reference : Objects \times ReferenceNames \rightarrow Objects$. Given the cross reference

$$Reference(o, rn) = o'$$

object $o \in Objects$ is called the *referrer* design object, rn the name of the reference, and $o' \in Objects$ the *referee* design object. Note that the transitive hull over the $Reference$ relation is not needed here. Instead, only the directly referred design objects are used, which is covered by the $Reference$ relation already.

Finally, the attributes of the design objects need to be formalized. Therefore, first the set of admissible attribute names $AttributeNames$ as well as the set of admissible attribute values $AttributeValues$ are introduced. Then, the mapping $Attribute : Objects \times AttributeNames \rightarrow AttributeValues$ can be defined assigning to each design object $o \in Objects$ and attribute name $an \in AttributeNames$ a corresponding attribute value $av \in AttributeValues$ such that

$$Attribute(o, an) = av.$$

Note that typically the set of admissible attribute names depends on the class $c \in Classes$ of the design object o with $Constructor(o) = c$. Furthermore, note that the admissible attribute values typically depend on the class c and the attribute name an. However, the following syntactic constraints to not depend on these relationships. Therefore, the respective relationships are omitted here to keep the mathematical foundations as simple as possible.

Constraint notation

Finally, the constraint notation used here comprises the a number of operators for accessing the design objects, their hierarchical relationships, their attribute values, and their cross references in a more compact manner. First, the design objects belonging to a particular class are made accessible:

- The set of design objects belonging to a specific class $c \in Classes$ can be obtained using the $\triangle(\cdot)$ operator such that $\triangle(c) = Instances(c)$.

Then, being able to access design objects of different classes different operators for accessing the children, the descendants, the references, and the attribute values of these design objects are provided:

- Children of a design object $o \in Objects$, which belong to a specific subset of design object $s \subseteq Objects$, can be obtained using the $(\cdot)\text{Ⓒ}(\cdot)$ operator such that $o\text{Ⓒ}s = Children(o) \cap s$.

- Descendants of a design object $o \in Objects$, which belong to subset of design objects $s \subseteq Objects$, can be obtained using the $(\cdot)//(\cdot)$ operator such that $o//s = Descendants(o) \cap s$.
- References of a design object $o \in Objects$ with a specific reference name $rn \in ReferenceNames$ can be obtained using the $(\cdot)®(\cdot)$ operator such that $o®rn = Reference(o, rn)$.
- Attributes of a design object $o \in Objects$ with a specific attribute name $an \in AttributeNames$ can be obtained using the $(\cdot)@(\cdot)$ operator such that $o@an = Attribute(o, an)$.

The previous operators only work with single design objects as input. To ease the formulation of the syntactic inconsistency constraints, the previous operators also work with subsets of design objects as input:

- Children of a subset of design objects $s_1 \subseteq Objects$, which belong to another subset of design objects $s_2 \subseteq Objects$, can be obtained using the $(\cdot)©(\cdot)$ operator such that $s_1©s_2 = \bigcup_{o \in s_1} Children(o) \cap s_2$.
- Descendants of subset of design objects $s_1 \subseteq Objects$, which belong to another subset of design objects $s_2 \subseteq Objects$, can be obtained using the $(\cdot)//(\cdot)$ operator such that $s_1//s_2 = \bigcup_{o \in s_1} Descendants(o) \cap s_2$.
- References of subset of design objects $s \subseteq Objects$ with a specific reference name $rn \in ReferenceNames$ can be obtained using the $(\cdot)®(\cdot)$ operator such that $s®rn = \{Reference(o, rn) \mid o \in s\}$.
- Attributes of a subset of design objects $s \subseteq Objects$ with a specific attribute name $an \in AttributeNames$ can be obtained using the $(\cdot)@(\cdot)$ operator such that $s@an = \{Attribute(o, an) \mid o \in s\}$.

Note that essentially single design objects $o \in Objects$ and attribute values $av \in AttributeValues$ are used interchangeable with sets of single design objects $\{o\}$ and attribute values $\{av\}$. Finally, also sets of design objects need to be filtered:

- Arbitrary sets of design objects can be filtered using the $(\cdot)[(\cdot) \mid (\cdot)]$ operator based on, for example,
  - their children from a specific subset of design objects such that $Objects[o \mid |o©s| = 0] \subseteq Objects$ with subset of design objects $s \subseteq Objects$, or

- their descendants from a specific subset of design objects such that $Objects[o \mid |o//s| = 0] \subseteq Objects$ with subset of design objects $s \subseteq Objects$, or
- their references such that $Objects[o \mid o®rn \in \triangle(c)] \subseteq Objects$ with reference name $rn \in ReferenceNames$ and class $c \in Classes$, or
- their attributes such that $Objects[o \mid o@an = av] \subseteq Objects$ with attribute name $an \in AttributeNames$ and attribute value $av \in AttributeValues$.

Note that the constraint notation combines ideas from the object constraint language (OCL) [WK99] for UML models (see Section 2.1.1]) and the XPath language [CFGR02] for extensible markup language (XML) documents. In particular, the OCL provides the means for navigating composition and aggregation associations, while XPath supports the facilities for filtering and navigating the descendants in tree hierarchies.

Basic inconsistencies

The basic inconsistency issues are concerned with basic properties of conceptual design models. In particular, basic *hierarchy* issues, basic *uniqueness* issues, and basic *reuse* issues are distinguished. In the following, each subtype is described in more detail.

Hierarchy basics A hierarchy can be defined over the design objects, which is based on the parent-child relationships and the composition associations respectively. The first rule requires this object hierarchy to be sound:

- $\forall p_1, p_2, c \in Objects : c \in p_1©Objects \wedge c \in p_2©Objects \Rightarrow p_1 = p_2$

The intention of the constraint is that an unambiguous hierarchical relationship exists between the design objects, i.e. each object is associated to at most one parent object. The second basic hierarchy issue concerns the use of the duration expression (see Section 5.1.3):

- $\forall d \in \triangle DurationExpression : \exists e \in \triangle Executable : d \in e//Objects$

A duration expression can be used only inside executables and refers to the number of computation steps, for which the executable has been residing in the same state. Consequently, the duration expression cannot be used inside default expressions of, e.g., ports (see Section 5.2.2), which are defined outside of executables. Furthermore, the duration expression cannot be used inside expressions of properties (see Section 5.3.2), which are attached to components directly.

Uniqueness basics Then, the first basic uniqueness constraint makes sure, that the names of the design objects are unique within their scope. Hereby, the scope is defined by the respective parent object:

- $\forall p \in Objects, c_1, c_2 \in p©Objects[o \mid o@Name \neq null] : c_1@Name = c_2@Name \Rightarrow c_1 = c_2$

Note that the constraint applies to ports (see Section 5.2.2), which are contained inside components (see Section 5.2.1), scenarios (see Section 5.3.4), and material interaction ports. Actually, the unique port naming is essential for the dynamic binding of components during execution (see Section 4.2.3). Furthermore, the constraint applies to properties (see Section 5.3.2), which are attached to components or executable states (see Section 5.1.2), as well as executables (i.e. scenarios, monitors, and behaviors) and parts (see Section 5.2.4), which are contained inside components. Finally, the constraint also applies to states and transitions inside executables. In particular, unique naming is important to identify design objects unambiguously within their context. Furthermore, the names might carry human-interpretable semantics, which is not taken into consideration here. Note that one might require, e.g., unique human-interpreted semantics, which is hard to evaluate automatically. Then, the following two basic uniqueness issues apply to the actions, which are contained inside states and transitions of executables:

- $\forall l \in \triangle Label, a_1, a_2 \in l©\triangle Action : a_1®Observation = a_2®Observation \Rightarrow a_1 = a_2$
- $\forall t \in \triangle Transition, a_1, a_2 \in t©\triangle Action : a_1®Observation = a_2®Observation \Rightarrow a_1 = a_2$

Consequently, at most one (write) action can be provided inside executable (state) labels and transitions for each observation. The constraint makes sure that in each execution step the observation assignment can be determined unambiguously. Note that in case an action has not been provided, the observation assignment defaults to the empty message $\perp \notin M$ (see Section 4.1.1). The last basic uniqueness constraint states that a port might serve only as the target of one channel:

- $\forall c_1, c_2 \in \triangle Channel : c_1®Target = c_2®Target \Rightarrow c_1 = c_2$

The reason for this limitation is that two channels with the same target port introduce a non-determinism (see Section 6.2.2 for more information). Consequently, one cannot decide unambiguously, which value to forward from which channel. Note that in some cases such non-determinisms might be desired, e.g., if both channels are feasible and the design decision, which alternative to prefer, is delayed. However, such cases are not considered in the limited scope of this doctoral thesis for simplicity.

Reuse basics Finally, the reuse inconsistency issues make sure that components and their ports are reused correctly. The first rule states that the port references must reflect correctly the information stored in the associated port definitions (see Section 5.2.2):

- $\forall p_1 \in \triangle PortReference, p_2 = p_1 ® Definition : p_1 @ Name = p_2 @ Name \wedge p_1 @ ReadType = p_2 @ ReadType \wedge p_1 @ WriteTypes = p_2 @ WriteTypes \wedge p_1 @ Direction = p_2 @ Direction$

In particular, the name, the type, and the direction attributes of the port reference and its definition must be equal. Then, the second reuse consistency rule makes sure, that the port references of each component reference (see Section 5.2.1) are valid:

- $\forall c_1 \in \triangle ComponentReference, p_1 \in c_1 © \triangle Port, c_2 = c_1 ® Definition : p_1 ® Definition \in c_2 © \triangle Port$

Consequently, each port reference must point to a port definition of the associated component definition. Finally, one has to make sure that the port references of each component reference are complete:

- $\forall c \in \triangle ComponentReference, p \in c ® Definition © \triangle Port : p \in c © \triangle Port ® Definition$

Consequently, each port definition has to be associated to a port reference. Note that the reuse information is limited to the port-based interface definition of the component. In contrast, "internal" details of the component definition such as behaviors (see Section 5.2.5) and parts (see Section 5.2.4) do not have to be referenced.

Permission inconsistencies

Subsequently, permissions are defined for reading and writing observations (see Section 5.1.1) from within properties (see Section 5.3.2) and executables (see Section 5.1.2) as well as for drawing channels (see Section 5.2.3) between ports (see Section 5.2.2). In the following, first the *write* permission issues are described before explaining the *read* permission issues and detailing the *channel* permission issues.

Write permissions The write permission issues concern the observations (i.e. ports or variables; see Section 5.1.1), which can be written by actions inside states and transitions of executables (i.e. behaviors, monitors, and scenarios; see Section 5.1.2):

- $\forall c \in \triangle Component, b \in c © \triangle Behavior, o \in b // \triangle Action ® Observation : o \in b © \triangle Variable \cup c © \triangle Port[p \mid p @ Direction \neq input] \cup c © \triangle MaterialInteractionPort © \triangle Port[p \mid p @ Direction = input]$

- $\forall c \in \triangle Component, m \in c © \triangle Monitor, o \in m // \triangle Action ® Observation : o \in m © \triangle Variable$

– $\forall c \in \triangle Component, s \in c©\triangle Scenario, o \in s//\triangle Action®Observation : o \in s©\triangle Variable \cup s©\triangle Port \cup c©\triangle Port[p \mid p@Direction = input] \cup c©\triangle MaterialInteractionPort©\triangle Port[p \mid p@Direction = input]$

The constraints say that behaviors may write their variables, the input and input-output ports of the containing component, as well as the input ports of the material interaction ports of the containing component. Consequently, behaviors can interact with dynamically bound components directly, which is not defined in the theory explicitly (see Section 5.2.5), but helps simplifying the models. In contrast, monitors may write only their variables, e.g., for tracking information across their activities. Finally, scenarios may write their variables, their custom material life ports, the input ports of the containing component, as well as the input ports of the material interaction ports of the containing components. Consequently, scenarios may provide inputs to the system under test, which is common practice in test-based approaches [Bec02]. Furthermore, scenarios are able to interact with components bound dynamically to the system under test, e.g., for preparing their state during test setup. Note that, alternatively, scenarios could have defined their own custom material interaction ports and, hence, bind components dynamically themselves instead. Such custom material interaction ports could yield more intuitive models. Therefore, in the future the proposed modeling technique should be revised accordingly. However, for assessing the feasibility of the test-driven design method the current version of the modeling technique is sufficient.

Read permissions Then, read permissions refer to the observations (i.e. ports and variables), which can be read by state and transition actions as well as transition guards inside executables (i.e. behaviors, monitors, and scenarios):

– $\forall c \in \triangle Component, b \in c©\triangle Behavior, o \in b//(\triangle Guard \cup \triangle Action)//\triangle ObservationExpression®Observation : o \in b©\triangle Variable \cup c©\triangle Port[p \mid p@Direction \neq output] \cup c©\triangle MaterialInteractionPort©\triangle Port[p \mid p@Direction = output]$

– $\forall c \in \triangle Component, m \in c©\triangle Monitor, o \in m//(\triangle Guard \cup \triangle Action)//\triangle ObservationExpression®Observation : o \in m©\triangle Variable \cup c©\triangle Port \cup c©\triangle MaterialInteractionPort©\triangle Port$

– $\forall c \in \triangle Component, s \in c©\triangle Scenario, o \in s//(\triangle Guard \cup \triangle Action)//\triangle ObservationExpression®Observation : o \in s©\triangle Variable \cup s©\triangle Port \cup c©\triangle Port[p \mid p@Direction \neq input] \cup c©\triangle MaterialInteractionPort©\triangle Port[p \mid p@Direction = output]$

The constraints say that behaviors may read their variables, the input and input-output ports of the containing component, as well as the output ports of the material interaction ports of the containing component. Consequently, behaviors are able to activate

material ports, observe the components bound dynamically, and read outputs from the dynamically bound components. In contrast, monitors may read their variables, the ports of the containing component, as well as the ports of the material interaction ports of the containing component. Note that monitors may read ports independent of their direction. Finally, scenarios may read their variables, their material life ports, the output and input-output ports of the containing component, as well as the output ports of the material interaction ports of the containing component. Again, direct interaction with dynamically bound components has been added, e.g., to observe their state during test execution. Subsequently, the following two read permission issues concern the observations (i.e. ports and variables), which can be read by property expressions inside executables (i.e. monitors and scenarios):

- $\forall c \in \triangle Component, m \in c©\triangle Monitor, o \in$
  $m//\triangle Property//\triangle ObservationExpression®Observation : o \in$
  $m©\triangle Variable \cup c©\triangle Port \cup c©\triangle MaterialInteractionPort©\triangle Port$

- $\forall c \in \triangle Component, s \in c©\triangle Scenario, o \in$
  $s//\triangle Property//\triangle ObservationExpression®Observation : o \in$
  $s©\triangle Variable \cup s©\triangle Port \cup c©\triangle Port \cup c©\triangle MaterialInteractionPort©\triangle Port$

The constraints say that properties inside monitors may read the variables of the containing monitor, the ports of the containing component, as well as the ports of the material interaction ports of the containing component. In contrast, properties inside scenarios may read the variables of the containing scenario, the ports of the containing scenario, the ports of the containing component, as well as the ports of the material interaction ports of the containing component. Consequently, both monitors and scenarios may constrain also the inputs and outputs of dynamically bound components as well as the identity dynamically bound components themselves. In particular, scenarios are able to track the components generated through their custom material life ports during test execution. Finally, the last read permission issue concerns the observations that can be read by properties, which are attached to component directly:

- $\forall c \in \triangle Component, o \in c©\triangle Property//\triangle ObservationExpression®Observation :$
  $o \in c©\triangle Port \cup c©\triangle MaterialInteractionPort©\triangle Ports$

The rule says that properties attached to components directly may read the ports of the containing component as well as the ports of the material interaction ports of the containing component. Consequently, data and energy inputs or outputs of the containing component can be accessed as well as material port activations and the inputs and outputs of dynamically bound components.

Channel permissions The last group of permission inconsistencies are the so-called channel permission issues, which are concerned with the correct use of channels (see Section 5.2.3) for connecting ports (see Section 5.2.2). The first rule states that channels cannot be used to connect material ports:

- $\forall s \in \triangle Channel, p \in s\circledR Source : p \notin \triangle MaterialPort \cup \triangle PortReference[r \mid r\circledR Definition \in \triangle MaterialPort]$
- $\forall s \in \triangle Channel, p \in s\circledR Target : p \notin \triangle MaterialPort \cup \triangle PortReference[r \mid r\circledR Definition \in \triangle MaterialPort]$

The reason for this limitation is that, in contrast to energy and data, material does not flow through (logical) channels, but it flows through space and it is bound dynamically to material ports based on entry and exit as well as binding conditions (see Sections 4.2.3 and 5.2.2). Then, the next two rules state the circumstances, under which a channel and source or target port might belong to the same parent component:

- $\forall c \in \triangle Component, p \in c\copyright\triangle Channel\circledR Source[s \mid s@Direction = input] : p \in c\copyright\triangle Port$
- $\forall c \in \triangle Component, p \in c\copyright\triangle Channel\circledR Target[t \mid t@Direction = output] : p \in c\copyright\triangle Port$

Consequently, input ports of the same component might serve as source ports and output ports of the same component might serve as target ports. Also, the rule allows one to model channels forwarding values from input ports to output ports of the same component directly. However, channel source and target ports might refer also to ports of some subcomponent. In such cases, the next two rules have to be considered:

- $\forall c \in \triangle Component, p \in c\copyright\triangle Channel\circledR Source[s \mid s@Direction = output] : p \in c\copyright\triangle Component\copyright\triangle Port \cup c\copyright\triangle MaterialInteractionPort\copyright\triangle Port$
- $\forall c \in \triangle Component, p \in c\copyright\triangle Channel\circledR Target[t \mid t@Direction = input] : p \in c\copyright\triangle Component\copyright\triangle Port \cup c\copyright\triangle MaterialInteractionPort\copyright\triangle Port$

Hence, output ports of subcomponents might serve as source ports of channels and input ports of subcomponents might serve as target ports of channels. Hereby, both the source and the target ports might belong to the same subcomponent for modeling feedback cycles. Alternatively, both the source and target ports might belong to material interaction ports and, hence, describe dynamic interactions based on collisions (see Section 5.2.2). Again, feedback cycles are supported in principle as well.

Type inconsistencies

At last, the type inconsistency issues are concerned with the consistency of design objects carrying user-defined type information. Consequently, type inconsistencies can be

caused by ports (see Section 5.1.1), which are connected by channels (see Section 5.2.3). Alternatively, type inconsistencies might arise inside expressions (see Section 5.1.3) including their argument expressions and referenced observations. Consequently, in the following *expression* type issues and *channel* type issues are distinguished, which are explained one after the other.

Expression types The first two expression type issues concern the expressions contained in properties (see Section 5.3.2) and executable guards (see Section 5.1.2), which both have to return boolean values:

- $\forall p \in \triangle Property, e \in p©\triangle Expression : e@Type = Boolean$
- $\forall g \in \triangle Guard, e \in g©\triangle Expression : e@Type = Boolean$

Then, the default expressions contained in observations (i.e. the expressions for computing the observation assignments of the initial state as described in Section 5.1.1) and the expressions contained in executable actions (see Section 5.1.2) must return a variable type, which is prescribed by the associated observation:

- $\forall o \in \triangle Observation, e = o©\triangle DefaultExpression : e@Type \in o@WriteTypes$
- $\forall a \in \triangle Action, o = a®Observation, e = a©\triangle Expression : e@Type \in o@WriteTypes$

However, one needs to consider an exception here: Kinetic energy ports might have an associated transform object or not (see Section 5.2.2). When a kinetic energy port has an associated transform object, the respective default expressions must return numbers, otherwise they must return transform objects directly:

- $\forall k \in \triangle KineticEnergyPort : k©\triangle Transform \neq null \Rightarrow k©\triangle DefaultExpression@Type = Number$
- $\forall k \in \triangle KineticEnergyPort : k©\triangle Transform = null \Rightarrow k©\triangle DefaultExpression@Type = Transform$

Note that in the first case the number is converted into a multiple of the respective reference transform. In the same manner, expressions of actions inside behaviors (see Section 5.2.5) writing kinetic energy output ports must be aware of the transform object. If the transform object exists, the action expressions must return numbers, otherwise they must return transform objects directly:

- $\forall a \in \triangle Action, o = a®Observation, e = a©\triangle Expression : o \in KineticEnergyPort \wedge o©\triangle Transform \neq null \Rightarrow e@Type = Number$
- $\forall a \in \triangle Action, o = a®Observation, e = a©\triangle Expression : o \in KineticEnergyPort \wedge o©\triangle Transform = null \Rightarrow e@Type = Transform$

Subsequently, the next two expression type issues refer to the type of the argument expressions, which are contained inside unary expressions and nary expressions, i.e. expressions that contain one or more argument expressions (e.g. the set cardinality expression or the set union expression):

- $\forall e_1 \in \triangle UnaryExpression, e_2 \in e_1 \copyright \triangle Expression : e_1@ArgumentType = e_2@Type$
- $\forall e_1 \in \triangle NaryExpression, e_2 \in e_1 \copyright \triangle Expression : e_1@ArgumentsType = e_2@Type$

The two constraints say that the argument expression of a unary expression must return the argument type of the unary expression, while the nary argument expressions must return the argument type of the nary expression. For example, both the unary cardinality and the nary union expressions require argument expressions of type *set*. Finally, the last expression type inconsistency refers to the type of observation expressions:

- $\forall o \in \triangle ObservationExpression : o@Type = o \circledR Observation@ReadType$

Consequently, the observation expression must return the read type of the referenced observation. Note that the observation expression allows one to read the value of variables and ports during system execution.

Channel types Finally, the first channel type issue concerns the types of ports (see Section 5.2.2), which are connected by channels (see Section 5.2.3) inside components (see Section 5.2.1):

- $\forall s \in \triangle Channel, p_1 \in s \circledR Source, p_2 \in s \circledR Target : p_1@ReadType \in p_2@WriteTypes$

The constraint says that for each channel the read type of the source port must be included in the write types of the respective target port. Note that the constraint only refers to the data type of the ports and not to the different port classes (i.e. energy or data, while material as been excluded before). Additional connectivity constraints of energy ports are provided in the following:

- $\forall s \in \triangle Channel, p_1 = s \circledR Source, p_2 = s \circledR Target : p_1 \in \triangle GenericEnergyPort \cup \triangle PortReference[r \mid r \circledR Definition \in \triangle GenericEnergyPort] \Rightarrow p_2 \in \triangle GenericEnergyPort \cup \triangle PortReference[r \mid r \circledR Definition \in \triangle GenericEnergyPort] \cup \triangle GenericPort \cup \triangle PortReference[r \mid r \circledR Definition \in \triangle GenericPort]$
- $\forall s \in \triangle Channel, p_1 = s \circledR Source, p_2 = s \circledR Target : p_1 \in \triangle KineticEnergyPort \cup \triangle PortReference[r \mid r \circledR Definition \in \triangle KineticEnergyPort] \Rightarrow p_2 \in \triangle KineticEnergyPort \cup \triangle PortReference[r \mid r \circledR Definition \in \triangle KineticEnergyPort] \cup \triangle GenericEnergyPort \cup \triangle PortReference[r \mid r \circledR Definition \in \triangle GenericEnergyPort] \cup \triangle GenericPort \cup \triangle PortReference[r \mid r \circledR Definition \in \triangle GenericPort]$

– $\forall s \in \triangle Channel, p_1 = s\circledR Source, p_2 = s\circledR Target : p_1 \in \triangle ElectricEnergyPort \cup \triangle PortReference[r \mid r\circledR Definition \in \triangle ElectricEnergyPort] \Rightarrow p_2 \in \triangle ElectricEnergyPort \cup \triangle PortReference[r \mid r\circledR Definition \in \triangle ElectricEnergyPort] \cup \triangle GenericEnergyPort \cup \triangle PortReference[r \mid r\circledR Definition \in \triangle GenericEnergyPort] \cup \triangle GenericPort \cup \triangle PortReference[r \mid r\circledR Definition \in \triangle GenericPort]$

Consequently, generic energy ports can be connected to generic energy ports or generic ports only. Then, kinetic energy ports can be connected to kinetic energy ports, generic energy ports, or generic ports. Furthermore, electric energy ports can be connect to electric energy ports, generic energy ports, or generic ports. Finally, the connectivity of data ports has to be constrained:

– $\forall s \in \triangle Channel, p_1 \in s\circledR Source, p_2 \in s\circledR Target : p_1 \in \triangle DataPort \cup \triangle PortReference[r \mid r\circledR Definition \in \triangle DataPort] \Rightarrow p_2 \in \triangle DataPort \cup \triangle PortReference[r \mid r\circledR Definition \in \triangle DataPort] \cup \triangle GenericPort \cup \triangle PortReference[r \mid r\circledR Definition \in \triangle GenericPort]$

It follows that data ports can be connected to data port or generic ports. Note that the port connectivity rules prescribe that the target port can be more generic than the source port. In contrast, the source port cannot be more generic than the target port. Similar mechanisms can be found also in modern type-safe programming languages such as Java [vON99] or C++ [WNST06]. Finally, note that the last four rules could also indicate incompleteness issues rather than inconsistency issues, e.g., when connecting a data port to an electric energy port. In this case, a mechanism might be missing, which is responsible for the conversion between data and energy. However, such incompleteness might be acceptable during conceptual design, but critical during design refinement [HRZ15a].

6.2 Semantic issues

Having introduced the syntactic issues in the previous section, now the semantics of the design objects is focused. According to Lindland et al. [LSS94], semantic issues are concerned with the meaning of the design information. Here, the meaning of the design information is given by the execution semantics, which has been defined in the theoretical foundations (see Chapter 4) and revised as well as extended while introducing the modeling technique (see Chapter 5). In particular, revised spatio-temporal components C over $I, I_M, O, O_M, P, B, Y, X$ (see Section 5.2.1) are executed in the context of scenarios (T, s_0, s_f, v_0) with state transition function $T : (S \times \bar{A}(Y_S \cup X_S) \times \bar{A}(B \cup Y \cup X) \times \bar{O} \times \bar{O}_M \times \bar{V})^* \rightarrow \mathbb{P}(S \times \bar{I} \times \bar{I}_M \times \bar{Y}_S \times \bar{X}_S \times \bar{P}_S \times \bar{V})$, state labels S, custom entry labels Y_S, custom exit labels X_S, custom property labels P_S, variable labels V, initial

state label $s_0 \in S$, final state label $s_f \in S$, and initial variable assignment $v_0 \in \bar{V}$ (see Section 5.3.4) to discover semantic issues fully automatically.

Formally, a test execution is defined as collision sensing state transition system computation $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*, p_S^*)$ over (T, s_0, s_f, v_0) such that there exists a collision sensing, channel binding, and motion forwarding behavior $(o, o_M, p, b, y, x) \in C(i, i_M, i_F, a)$ of component C with matching (energy and data) input channel assignment streams $i^* \in \bar{I}^\infty$ of the scenario computation and (energy and data) input channel histories $i \in \vec{I}$ of the revised spatio-temporal computation, i.e.

$$\forall n \in \mathbb{N}, \iota \in I : i(\iota)(n) = i^*(n)(\iota),$$

and matching kinetic energy input channel assignment streams $i_M^* \in \bar{I}_M^\infty$ of the scenario computation and kinetic energy input channel histories $i_M \in \vec{I}_M$ of the revised spatio-temporal computation, i.e.

$$\forall n \in \mathbb{N}, \iota \in I_M : i_M(\iota)(n) = i_M^*(n)(\iota),$$

and matching (material port) activation assignment streams $a^* \in \bar{A}(B \cup Y \cup X)^\infty$ of the scenario computation and (material port) activation histories $a \in \vec{A}(B \cup Y \cup X)$ of the revised spatio-temporal computation, i.e.

$$\forall n \in \mathbb{N}, \lambda \in B \cup Y \cup X : a(\lambda)(n) = a^*(n)(\lambda),$$

and matching (energy and data) output channel assignment streams $o^* \in \bar{O}^\infty$ of the scenario computation and (energy and data) output channel histories $o \in \vec{O}$ of the revised spatio-temporal computation, i.e.

$$\forall n \in \mathbb{N}, \omega \in O : o(\omega)(n) = o^*(n)(\omega),$$

and matching kinetic energy output channel assignment streams $o_M^* \in \bar{O}_M^\infty$ of the scenario computation and kinetic energy output channel histories $o_M \in \vec{O}_M$ of the revised spatio-temporal computation, i.e.

$$\forall n \in \mathbb{N}, \omega \in O_M : o_M(\omega)(n) = o_M^*(n)(\omega).$$

The previous equations require the behavior $(o, o_M, p, b, y, x) \in C(i, i_M, i_F, a)$ of component C to coincide with the computation $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*, p_S^*)$ of scenario (T, s_0, s_f, v_0). Note that the equations do not make any assumptions about the custom entry and exit assignments $y_S^* \in \bar{Y}_S^\infty$ and $x_S^* \in \bar{X}_S^\infty$ of the scenario as well as the respective activation assignments $a_S^* \in \bar{A}(Y_S \cup X_S)^\infty$. To describe their behavior, the definition of the generated component stream G (see Section 5.2.1) needs to be revised.

The generated component stream $G \in \mathbb{P}(\mathcal{C})^\infty$ is redefined over the custom entries Y_S and exists X_S of the scenario (T, s_0, s_f, v_0), the entries Y and exits X of the revised spatio-temporal component C, and the entries and exits of the generated components themselves. Consequently, for each computation step $n \in \mathbb{N}$ and generated component $C' \in G(n+1)$ with behavior $(o', o'_M, p', b', y', x') \in C'(i', i'_M, i_F, a')$ the component C' must have existed in the previous computation step, i.e.

$$C' \in G(n)$$

or the generated component C' must have entered in the current time step $n+1 \in \mathbb{N}$ through the entries Y of component C, i.e.

$$\exists t' \in T, \upsilon \in Y : y(\upsilon)(n+1) = (t', C'),$$

or through the entries Y'' of some other generated component $C'' \in G(n+1)$ with $C'' \neq C'$ and behavior $(o'', o''_M, p'', b'', y'', x'') \in C''(i'', i''_M, i_F, a'')$, i.e.

$$\exists t' \in T, \upsilon'' \in Y'' : y''(\upsilon'')(n+1) = (t', C'),$$

or through the custom entries Y_S of scenario (T, s_0, s_f, v_0), i.e.

$$\exists t' \in T, \upsilon_S \in Y_S : y^*_S(n+1)(\upsilon_S) = (t', C'),$$

and the generated component C' must not have been deleted in the next time step $n+1 \in \mathbb{N}$ through the exits X of component C, i.e.

$$\nexists \xi \in X : \bigvee_{\pi' \in P'} x(\xi)(n+1) \bowtie p'(\pi')(n+1),$$

or through the exists X'' of another generated component $C'' \in G(n+1)$ with $C'' \neq C'$ and behavior $(o'', o''_M, p'', b'', y'', x'') \in C''(i'', i''_M, i_F, a'')$, i.e.

$$\nexists \xi'' \in X'' : \bigvee_{\pi'' \in P'} x''(\xi'')(n+1) \bowtie p'(\pi')(n+1),$$

or through the custom exists X_S of scenario (T, s_0, s_f, v_0), i.e.

$$\nexists \xi_S \in X_S : \bigvee_{\pi' \in P'} x^*_S(n+1)(\xi_S) \bowtie p'(\pi')(n+1).$$

Consequently, the collision sensing property, the channel binding property, and the motion forwarding property (see Section 5.2.1) can be redefined over the revised definition

of the generated component stream G. In particular, the revised definitions have to respect the entries and exits caused by the component C, some generated component $C'' \in G(n+1)$, and the scenario (T, s_0, s_f, v_0). Note that mostly the definition of the generated components stream G needs to be substituted, which is why the revised definitions of the properties mentioned above is omitted. Finally, the semantic issues over test executions can be defined. Subsequently, *intrinsic* and *extrinsic* semantic issues are distinguished, depending on whether the execution constraints are predefined (i.e. intrinsic) or user-defined (i.e. extrinsic).

In the following, first the extrinsic semantic issues are introduced in Section 6.2.1 before explaining the intrinsic semantic issues in Section 6.2.2. Furthermore, precise and unambiguous explanations and definitions of the execution constraints underlying each semantic issue are provided, such that the proposed constraint checks can be implemented by potential tool vendors easily.

6.2.1 Extrinsic issues

As mentioned previously, extrinsic semantic issues represent user-defined (or custom) execution constraints. In particular, one type of extrinsic semantic issue is supported, namely the so-called property violations. In the following, property violations are explained in more detail.

Property violations

Properties (see Section 5.3.2) allow one to encode custom execution constraints. Hereby, properties can be attached to components directly (see Section 5.2.1) or the the state labels of monitors (see Section 5.3.3) and scenarios (see Section 5.3.4). These three cases are distinguished in the following, while commonly referring to the test execution $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*, p_S^*)$ over scenario (T, s_0, s_f, v_0) and component C.

Scenario properties Recall the definition of scenarios (T, s_0, s_f, v_0) with transition function $T : (S \times \bar{A}(Y_S \cup X_S) \times \bar{A}(B \cup Y \cup X) \times \bar{O} \times \bar{O}_M \times \bar{V})^* \to \mathbb{P}(S \times \bar{I} \times \bar{I}_M \times \bar{Y}_S \times \bar{X}_S \times \bar{P}_S \times \bar{V})$ and property assignments $\bar{P}_S = \{P_S \to \mathbb{B}\}$. Furthermore, note that $p_S^* \in \bar{P}_S^\infty$ represents the property assignments of the test execution $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*, p_S^*)$ over scenario (T, s_0, s_f, v_0) and revised spatio-temporal component C. The given test execution is said to violate some property $\pi_S \in P_S$ at computation step $k \in \mathbb{N}$ if

$$p_S^*(k)(\pi_S) = false.$$

Note that in case the property π_S is contained in the state $s^*(k)$ the value $p_S^*(k)(\pi_S)$ is computed using a boolean expression $E_{\pi_S} : (S \times \bar{A}(Y_S \cup X_S) \times \bar{A}(B \cup Y \cup X) \times \bar{O} \times \bar{O}_M \times \bar{I} \times \bar{I}_M \times \bar{V})^* \to \mathbb{B}$ such that

$$p_S^*(k)(\pi_S) = E_{\pi_S}(s^* \downarrow k, a_S^* \downarrow k, a^* \downarrow k, o^* \downarrow k, o_M^* \downarrow k, i^* \downarrow k, i_M^* \downarrow k, v^* \downarrow k).$$

Otherwise, the value $p_S^*(k)(\pi_S)$ defaults to $true$, i.e. the property is not violated. Note that properties inside scenarios represent user-defined execution constraints of test engineers rather than requirement engineers or manufacturing process engineers. For example, a scenario might require the system to provide a predefined response to some input within a maximum duration. Hereby, the duration constraint can be encoded as a property of some wait step of the respective scenario (see the industry-close showcase in Chapter 8 for more details).

Monitor properties Subsequently, recall the definition of monitors (T', s_0', v_0') with transition function $T' : (S' \times \bar{I} \times \bar{I}_M \times \bar{A}(B \cup Y \cup X) \times \bar{O} \times \bar{O}_M \times \bar{V}')^* \to \mathbb{P}(S' \times \bar{P}_M' \times \bar{V}')$ and property assignments $\bar{P}_M' = \{P_M' \to \mathbb{B}\}$. Given the test execution $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*, p_S^*)$ over scenario (T, s_0, s_f, v_0) and component C, the monitor execution $(i^*, i_M^*, a^*, o^*, o_M^*, s'^*, v'^*, p_M'^*)$ over the monitor (T', s_0', v_0') is defined. Note that the input channel assignment streams $i^* \in \bar{I}^\infty$, kinetic energy input channel assignment streams $i_M^* \in \bar{I}_M^\infty$, (material port) activation assignment streams $a^* \in \bar{A}(B \cup Y \cup X)^\infty$, output channel assignment streams $o^* \in \bar{O}^\infty$, and kinetic energy output channel assignment streams $o_M^* \in \bar{O}_M^\infty$ are shared between the test execution and the monitor execution. Only, the state streams $s'^* \in S'^\infty$, variable assignment streams $v'^* \in \bar{V}'^\infty$, and property assignment streams $p_M'^* \in \bar{P}_M'^\infty$ are computed individually. Then, the monitor execution is said to violate property $\pi_M' \in P_M'$ at computation step $k \in \mathbb{N}$ if

$$p_M'^*(k)(\pi_M') = false.$$

Again, note that in the case the property π_M' is contained in the state $s'^*(k)$ the value $p_M'^*(k)(\pi_M')$ is computed using a boolean expression $E_{\pi_M'} : (S \times \bar{I} \times \bar{I}_M \times \bar{A}(B \cup Y \cup X) \times \bar{O} \times \bar{O}_M \times \bar{V})^* \to \mathbb{B}$ such that

$$p_M'^*(k)(\pi_M') = E_{\pi_M'}(s'^* \downarrow k, i^* \downarrow k, i_M^* \downarrow k, a^* \downarrow k, o^* \downarrow k, o_M^* \downarrow k, v'^* \downarrow k).$$

Otherwise, the value $p_M'^*(k)(\pi_M')$ defaults to $true$, i.e. the property is not violated. Note that monitor properties represent user-defined execution constraints of requirement and manufacturing process engineers rather than test engineers depending on whether the monitor includes design decisions (e.g. states / activities not prescribed by the customer).

Finally, note that the monitor property violation issue can be caused also by the monitors of generated components $C' \in G(k)$. For example, a manufacturing process monitor might require the system to finish a manufacturing operation (e.g. milling or grinding the workpiece) within a predefined maximum duration. Again, the duration constraint can be encoded as a property of the corresponding monitor activity (see the industry-close showcase in Chapter 8 for more details).

Component properties Finally, recall the definition of properties π_C, which are attached to components directly. Such properties are represented by some boolean expression $E_{\pi_C} : (\bar{I} \times \bar{I}_M \times \bar{A}(B \cup Y \cup X) \times \bar{O} \times \bar{O}_M)^* \rightarrow \mathbb{B}$. The test execution $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*, p_S^*)$ over scenario (T, s_0, s_f, v_0) and component C is said to violate property π_C at computation step $k \in \mathbb{N}$ if

$$E_{\pi_C}(i^* \downarrow k, i_M^* \downarrow k, a^* \downarrow k, o^* \downarrow k, o_M^* \downarrow k) = false.$$

Otherwise, the test execution does not violate property π_C. Note that properties, which are attached to components directly, represent user-defined execution constraints of requirement engineers rather than manufacturing process engineers and test engineers. Finally, note that the component property violation issue can be caused also by the properties of generated components $C' \in G(k)$. For example, a component property might require the demand for electric energy in each computation step not to exceed a predefined limit. If the implementation satisfies such property, the electric energy supply can be dimensioned accordingly.

6.2.2 Intrinsic issues

As noted previously, the intrinsic semantic issues are based on predefined execution constraints, which hold for each design case equally, as opposed to the user-defined extrinsic execution constraints introduced previously. In particular, three types of intrinsic semantic issues are distinguished, namely *part collisions*, *non-determinisms*, and *computation timeouts*. In the following, each type of intrinsic semantic issue is explained individually, while introducing further subtypes where necessary.

Part collisions

Part collision issues indicate that the manufacturing system cannot be operated without the danger of severe damages. Therefore, in the following the original collision free property (see Definition 4.32) is adapted to the revised formalism. Given test execution $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*)$ and the respective component behavior

$(o, o_M, p, b, y, x) \in C(i, i_M, i_F, a)$ a part collision issue is said to appear at computation step $k \in \mathbb{N}$ if two different parts of component C collide, i.e.

$$\exists \pi_1, \pi_2 \in P, \pi_1 \neq \pi_2 : p(\pi_1)(k) \bowtie p(\pi_2)(k)$$

or a part of component C and a part of some generated component $C' \in G(k)$ with behavior $(o', o'_M, p', b', y', x') \in C'(i', i'_M, i_F, a')$ collide, i.e.

$$\exists \pi \in P, \pi' \in P' : p(\pi)(k) \bowtie p'(\pi')(k)$$

or two different parts of the same generated component $C' \in G(k)$ with behavior $(o', o'_M, p', b', y', x') \in C'(i', i'_M, i_F, a')$ collide, i.e.

$$\exists \pi'_1, \pi'_2 \in P', \pi'_1 \neq \pi'_2 : p'(\pi'_1)(k) \bowtie p'(\pi'_2)(k)$$

or a part of some generated component $C' \in G(k)$ with behavior $(o', o'_M, p', b', y', x') \in C'(i', i'_M, i_F, a')$ and a part of some different generated component $C'' \in G(k)$ with $C' \neq C''$ and behavior $(o'', o''_M, p'', b'', y'', x'') \in C''(i'', i''_M, i_F, a'')$ collide, i.e.

$$\exists \pi' \in P', \pi'' \in P'' : p'(\pi')(k) \bowtie p''(\pi'')(k).$$

Note that the collision relation $\bowtie \subseteq V \times V$ is expected to report penetration between the two volumes rather than zero-dimensional point, one-dimensional line, or two-dimensional plane touching. Obviously, touching volumes might be desired, e.g., while manipulating a workpiece with a tool. In contrast, real penetration between different part volumes does not appear in practice due to physical laws.

Non-determinisms

Subsequently, non-determinism issues indicate that multiple possible test executions $(a^*_S, a^*, o^*, o^*_M, s^*, v^*, i^*, i^*_M, y^*_S, x^*_S, p^*_S)$ exist over the same scenario (T, s_0, s_f, v_0) and component C and / or that multiple monitor executions $(i^*, i^*_M, a^*, o^*, o^*_M, s'^*, v'^*, p'^*_M)$ exist over some monitor (T', s'_0, v'_0) for a given test execution. In principle, several reasons might exist for a non-determinism, namely *multiple* simultaneously enabled *transitions*, *multiple* simultaneously writing *behaviors*, *multiple* simultaneously active *bindings*, and *general fixed points*. In the following, each case is explained in more detail.

Multiple transitions Multiple simultaneously enabled transitions can be found in all types of executables (see Section 5.1.2), i.e. scenarios (see Section 5.3.4), monitors (see Section 5.3.3), and behaviors (see Section 5.2.5). In general, an executable (T, s_0, v_0)

with transition function $T : (S \times \bar{I} \times \bar{V})^* \rightarrow \mathbb{P}(S \times \bar{O} \times \bar{V})$, initial state $s_0 \in S$, and initial variable assignments $v_0 \in \bar{V}$ includes a non-determinism, if for a given input state multiple target states exist, i.e.

$$\exists (s, i, v) \in (S \times \bar{I} \times \bar{V})^*, s(0) = s_0, v(0) = v_0 : |T(s, i, v)| > 1.$$

Hence, given some test execution $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*, p_S^*)$ over scenario (T, s_0, s_f, v_0) and component C the scenario might include a multiple transitions non-determinism at computation step $k \in \mathbb{N}$ if

$$|T(s^* \downarrow k, a_S^* \downarrow k, a^* \downarrow k, o^* \downarrow k, o_M^* \downarrow k, v^* \downarrow k)| > 1.$$

In contrast, a monitor computation $(i^*, i_M^*, a^*, o^*, o_M^*, s'^*, v'^*, p_M'^*)$ over monitor (T', s_0', v_0') for the same test execution might cause a multiple transitions non-determinism at computation step $k \in \mathbb{N}$ if

$$|T'(s'^* \downarrow k, i^* \downarrow k, i_M^* \downarrow k, a^* \downarrow k, o^* \downarrow k, o_M^* \downarrow k, v'^* \downarrow k)| > 1.$$

Finally, a behavior computation $(i^*, i_M^*, a^*, s''^*, v''^*, o^*, o_M^*, p^*, b^*, y^*, x^*)$ over behavior (T'', s_0'', v_0'') for the same test execution might have raised a multiple transitions non-determinism issue at computation step $k \in \mathbb{N}$ if

$$|T''(s''^* \downarrow k, i^* \downarrow k, i_M^* \downarrow k, a^* \downarrow k, v''^* \downarrow k)| > 1.$$

Note that multiple transitions are enabled simultaneously, if their source states are equal and their guard expressions return *true* for the same stimulus. However, such stimuli might not occur during test execution such that possible non-determinism issues remain undetected. To overcome this problem, advanced reasoning tools are required such as SAT solvers [MMZ+01] or SMT solvers [dMB08]. Finally, note that the multiple transitions non-determinism issue can be caused also by the monitors and behaviors of generated components $C' \in G(k)$.

Multiple behaviors Then, the second cause for a non-determinism issue might be multiple simultaneously writing behaviors (see Section 5.2.5). Assume that component C over $I, I_M, O, O_M, P, B, Y, X$ is composed of two behaviors (T', s_0', v_0') and (T'', s_0'', v_0'') with transition functions $T' : (S' \times \bar{I} \times \bar{I}_M \times \bar{A}(B \cup Y \cup X) \times \bar{V}')^* \rightarrow \mathbb{P}(S' \times \bar{O} \times \bar{O}_M \times \bar{P}' \times \bar{B} \times \bar{Y} \times \bar{X} \times \bar{V}')$ and $T'' : (S'' \times \bar{I} \times \bar{I}_M \times \bar{A}(B \cup Y \cup X) \times \bar{V}'')^* \rightarrow \mathbb{P}(S'' \times \bar{O} \times \bar{O}_M \times \bar{P}'' \times \bar{B} \times \bar{Y} \times \bar{X} \times \bar{V}'')$, initial labels $s_0' \in S'$ and $s_0'' \in S''$, as well as initial variable assignments $v_0' \in \bar{V}'$ and $v_0'' \in \bar{V}''$ such that the part labels of the behaviors are disjoint, i.e. $P' \cap P'' = \emptyset$, and complete with respect to component C, i.e. $P' \cup P'' = P$. Furthermore, assume that given test

execution $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*, p_S^*)$ over scenario (T, s_0, s_f, v_0) and component C the related behavior computations are $(i^*, i_M^*, a^*, s'^*, v'^*, o'^*, o_M'^*, p'^*, b'^*, y'^*, x'^*)$ over behavior (T', s_0', v_0') and $(i^*, i_M^*, a^*, s''^*, v''^*, o''^*, o_M''^*, p''^*, b''^*, y''^*, x''^*)$ over behavior (T'', s_0'', v_0''). A multiple behaviors non-determinism is said to occur at computation step $k \in \mathbb{N}$ if the behaviors (T', s_0', v_0') and (T'', s_0'', v_0'') write the same output channel $\omega \in O$ of component C simultaneously, i.e.

$$\exists \omega \in O : o'^*(k)(\omega) \neq \bot \wedge o''^*(k)(\omega) \neq \bot$$

or the behaviors (T', s_0', v_0') and (T'', s_0'', v_0'') write the same kinetic energy output channel $\omega_M \in O_M$ of component C simultaneously, i.e.

$$\exists \omega_M \in O_M : o_M'^*(k)(\omega_M) \neq \bot \wedge o_M''^*(k)(\omega_M) \neq \bot$$

or the behaviors (T', s_0', v_0') and (T'', s_0'', v_0'') write the same material interaction port $\beta \in B$ of component C simultaneously, i.e.

$$\exists \beta \in B : b'^*(k)(\beta) \neq \bot \wedge b''^*(k)(\beta) \neq \bot$$

or the behaviors (T', s_0', v_0') and (T'', s_0'', v_0'') write the same material entry port $\upsilon \in Y$ of component C simultaneously, i.e.

$$\exists \upsilon \in Y : y'^*(k)(\upsilon) \neq \bot \wedge y''^*(k)(\upsilon) \neq \bot$$

or the behaviors (T', s_0', v_0') and (T'', s_0'', v_0'') write the same material exit port $\xi \in X$ of component C simultaneously, i.e.

$$\exists \xi \in X : x'^*(k)(\xi) \neq \bot \wedge x''^*(k)(\xi) \neq \bot.$$

If both behaviors (T', s_0', v_0') and (T'', s_0'', v_0'') write the same port simultaneously, one of the values has to be selected randomly leading to a multiple behaviors non-determinism issue. On the other hand, if one behavior writes the empty message $\bot$ the value of the other behavior can be used. Consequently, if both behaviors write the empty message $\bot$ the component output also is the empty message $\bot$. In the latter two cases, no intrinsic semantic issue is raised. Finally, note that the multiple actions non-determinism issues can be caused also by the behaviors of generated components $C' \in G(k)$.

Multiple bindings Subsequently, the third cause for a non-determinism issue might be multiple simultaneously active bindings to different material interaction ports (see Sections 4.2.2 and 4.2.3). Hereby, two cases are distinguished, namely *conflicting dynamic channels* and *multiple kinetic energy forwarding*. As previously, given the test

execution $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*, p_S^*)$ over scenario (T, s_0, s_f, v_0) and component C over $I, I_M, O, O_M, P, B, Y, X$ a multiple bindings non-determinism issue occurs at computation step $k \in \mathbb{N}$ due to conflicting dynamic channels if two (possibly identical) binding labels $\beta_1, \beta_2 \in B$ of component C exist, which bind component $C' \in \mathcal{C}$ over $I', I'_M, O', O'_M, P', B', Y', X'$, i.e.

$$C' \in a(\beta_1)(k) \wedge C' \in a(\beta_2)(k)$$

and the binding assignments $b(\beta_1)(k) = (s_1, c_1, f_1)$ and $b(\beta_2)(k) = (s_2, c_2, f_2)$ bind different (kinetic energy) output channels $\omega_1, \omega_2 \in O \cup O_M$ of component C with $\omega_1 \neq \omega_2$ to the same (kinetic energy) input channel $\iota' \in I' \cup I'_M$ of component C', i.e.

$$\exists \omega_1, \omega_2 \in O \cup O_M, \omega_1 \neq \omega_2, \iota' \in I' \cup I'_M : (\omega_1, \iota') \in c_1 \wedge (\omega_2, \iota') \in c_2$$

or the binding assignments $b(\beta_1)(k) = (s_1, c_1, f_1)$ and $b(\beta_2) = (s_2, c_2, f_2)(k)$ bind different (kinetic energy) output channels $\omega'_1, \omega'_2 \in O' \cup O'_M$ of component C' with $\omega'_1 \neq \omega'_2$ to the same (kinetic energy) input channel $\iota \in I \cup I_M$ of component C, i.e.

$$\exists \omega'_1, \omega'_2 \in O' \cup O'_M, \omega'_1 \neq \omega'_2, \iota \in I \cup I_M : (\omega'_1, \iota) \in c_1 \wedge (\omega'_2, \iota) \in c_2.$$

Alternatively, a multiple bindings non-determinism issue occurs at computation step $k \in \mathbb{N}$ due to conflicting dynamic channels if there exist two components $C'_1 \in \mathcal{C}$ over $I'_1, I'_{M,1}, O'_1, O'_{M,1}, P'_1, B'_1, Y'_1, X'_1$ and $C_2 \in \mathcal{C}$ over $I'_2, I'_{M,2}, O'_2, O'_{M,2}, P'_2, B'_2, Y'_2, X'_2$ with $C'_1 \neq C'_2$, which are bound to two (possibly identical) binding labels $\beta_1, \beta_2 \in B$, i.e.

$$C'_1 \in a(\beta_1)(k) \wedge C'_2 \in a(\beta_2)(k)$$

and the binding assignments $b(\beta_1)(k) = (s_1, c_1, f_1)$ and $b(\beta_2)(k) = (s_2, c_2, f_2)$ bind different (kinetic energy) output channels $\omega'_1 \in O'_1 \cup O'_{M,1}$ and $\omega'_2 \in O'_2 \cup O'_{M,2}$ of the components C'_1 and C'_2 to the same (kinetic energy) input channel $\iota \in I \cup I_M$ of component C, i.e.

$$\exists \omega'_1 \in O'_1 \cup O'_{M,1}, \omega'_2 \in O'_2 \cup O'_{M,2}, \iota \in I \cup I_M : (\omega'_1, \iota) \in c_1 \wedge (\omega'_2, \iota) \in c_2.$$

Finally, a multiple bindings non-determinism issue is raised at computation step $k \in \mathbb{N}$ due to multiple kinetic energy forwarding if a revised spatio-temporal component $C' \in \mathcal{C}$ over $I', I'_M, O', O'_M, P', B', Y', X'$ is bound to two different binding labels $\beta_1, \beta_2 \in B_1$ with $\beta_1 \neq \beta_2$, i.e.

$$C' \in a(\beta_1)(k) \wedge C' \in a(\beta_2)(k)$$

and the binding labels β_1 and β_2 belong to two different subcomponents $C_1 \in D(C)$ over $I_1, I_{M,1}, O_1, O_{M,1}, P_1, B_1, Y_1, X_1$ and $C_2 \in D(C)$ over $I_2, I_{M,2}, O_2, O_{M,2}, P_2, B_2, Y_2, X_2$ of

component C with $C_1 \neq C_2$, $\beta_1 \in B_1$, and $\beta_2 \in B_2$, and the components C_1 and C_2 are not related to each other, i.e.

$$C_1 \notin D(C_2) \wedge C_2 \notin D(C_1)$$

and the binding assignments $b(\beta_1)(k) = (s_1, c_1, f_1)$ and $b(\beta_2)(k) = (s_2, c_2, f_2)$ both forward kinetic energy from component C_1 and C_2 respectively to the bound component C', i.e.

$$f_1 = f_2 = true.$$

Note that the original channel binding and mover binding properties (see Section 4.2.3) do not consider such cases. In the proposed approach conflicting dynamic channels and kinetic energy forwarding are considered as non-determinisms. Consequently, choosing either of the dynamic channels and forwarding the port assignments as well as choosing either order of kinetic energy transforms yields a valid execution trace of the system. During test execution, the selection can be done randomly and a non-determinism issue can be reported. Alternatively, one could generate all possible traces during test execution, which can become a computationally intensive task. Then, also the question arises how the test results are aggregated across all possible execution traces and presented to the user. Such advanced considerations are omitted here. Finally, note that the multiple bindings non-determinism issue can be caused also by the binding assignments $b'(\beta')(k) = (v', c', f')$ with $\beta' \in B'$ of generated components $C' \in G(k)$.

General fixed points The last cause of a non-determinism issue might be a general fixed point problem. Again, the general fixed point problem can have two possible causes: (1) The composition of *weakly causal executables* (see Section 5.1.2) or (2) the *mutual binding* of components with kinetic energy forwarding (see Section 5.2.1).

In the case of weakly causal executables, the composition might include (a) the scenario and the component behavior or (b) the behaviors of two different subcomponents, while monitors can be neglected. In the first case, it is assumed that component C is tested with scenario (T, s_0, s_f, v_0) and the reaction of component C is given by the behavior (T', s'_0, v'_0). Then, the general fixed point problem arises at computation step $k \in \mathbb{N}$ if there exists a (kinetic energy) input channel $\iota \in I \cup I_M$ and a (kinetic energy) output channel $\omega \in O \cup O_M$ of component C such that the scenario (T, s_0, s_f, v_0) is weakly causal with respect to ω and the behavior (T', s'_0, v'_0) is weakly causal with respect to ι at computation step k (see Section 5.1.2 for the definition of weak causality). In the second case, it is assumed that component C is composed of two subcomponents C_1 and C_2 with $C = \vdash_{I_M} (C_1 \odot C_2)$. Furthermore, it is assumed that the reaction of component C_1 is given by the behavior $(T'_1, s'_{0,1}, v'_{0,1})$ and the reaction of component C_2 is given by

the behavior $(T_2', s_{0,2}', v_{0,2}')$. Then, the general fixed point problem arises at computation step $k \in \mathbb{N}$ if there exists a (kinetic energy) input channel $\iota_1 \in I_1 \cup I_{M,1}$ of component C_1, which is a (kinetic energy) output channel of component C_2, i.e. $\iota_1 \in O_2 \cup O_{M,2}$, and there exists an (kinetic energy) input channel $\iota_2 \in I_2 \cup I_{M,2}$ of component C_2, which is a (kinetic energy) output channel of component C_1, i.e. $\iota_2 \in O_1 \cup O_{M,1}$, and the behavior $(T_1', s_{0,1}', v_{0,1}')$ is weakly causal with respect to ι_1 and the behavior $(T_2', s_{0,2}', v_{0,2}')$ is weakly causal with respect to ι_2 at computation step k. Note that the second cause might also appear inside the static component architecture of generated components $C' \in G(k)$.

Instead, in the case of mutual bindings with kinetic energy forwarding assume a test execution $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*, p_S^*)$ over scenario (T, s_0, s_f, v_0) and component C with behavior $(o, o_M, p, b, y, x) \in C(i, i_M, i_F, a)$ and generated component stream G. Then, a general fixed point issue appears due to mutual bindings with kinetic energy forwarding at computation step $k \in \mathbb{N}$ if there exist two components $C' \in \mathcal{C}$ over $I', I_M', O', O_M', P', B', Y', X'$ and $C'' \in \mathcal{C}$ over $I'', I_M'', O'', O_M'', P'', B'', Y'', X''$ with $C' \neq C''$, behavior $(o', o_M', p', b', y', x') \in C'(i', i_M', i_F, a')$, and behavior $(o'', o_M'', p'', b'', y'', x'') \in C''(i'', i_M'', i_F, a'')$ such that component C' participates in the test execution, i.e.

$$C' = C \vee C' \in D(C) \vee \exists C''' \in G(k) : C' = C''' \vee C' \in D(C''')$$

and component C'' participates in the test execution, i.e.

$$C'' = C \vee C'' \in D(C) \vee \exists C''' \in G(k) : C'' = C''' \vee C'' \in D(C''')$$

and C'' is bound to C' and the binding forwards kinetic energy, i.e.

$$\exists \beta' \in B' : C'' \in a'(\beta')(k) \wedge \exists v \in V, c \in \mathbb{P}(\mathcal{O} \times \mathcal{I}) : b'(\beta')(k) = (v, c, true)$$

and C' is bound to C'' and the binding also forwards kinetic energy, i.e.

$$\exists \beta'' \in B'' : C' \in a''(\beta'')(k) \wedge \exists v \in V, c \in \mathbb{P}(\mathcal{O} \times \mathcal{I}) : b''(\beta'')(k) = (v, c, true).$$

Consequently, the kinetic energy forwarding $i_F(C')(k)$ depends on the kinetic energy forwarding $i_F(C'')(k)$ and vice versa (see Section 5.2.1 for more details on how to compute the kinetic energy forwarding). Consequently, the fixed point of the kinetic energy forwarding equation has to be found. Note that an appropriate solver is not provided for simplicity, but a semantic issue is raised and the test execution is terminated instead. Consequently, the general fixed point problem has to be resolved manually by the engineers. Finally, note that the general fixed point problem might appear also due to mutual binding caused by the binding assignments $b'''(\beta''')(k)$ with $\beta''' \in B'''$ of generated components $C''' \in G(k)$.

Computation timeouts

Finally, computation timeout issues indicate that the computation of a test execution $(a_S^*, a^*, o^*, o_M^*, s^*, v^*, i^*, i_M^*, y_S^*, x_S^*, p_S^*)$ over scenario (T, s_0, s_f, v_0) and component C could not be completed within a predefined time window $w \in \mathbb{R}_0^+$ with arbitrary time unit. The underlying assumption is that computing the successor state $(s^*(n+1), i^*(n), i_M^*(n), y_S^*(n), x_S^*(n), p_S^*(n), v^*(n+1)) \in T(s^* \downarrow n, a_S^* \downarrow n, a^* \downarrow n, o^* \downarrow n, o_M^* \downarrow n, v^* \downarrow n)$ takes some time $t_n \in \mathbb{R}^+$ at computation step $n \in \mathbb{N}$. Then, after $k \in \mathbb{N}$ computation steps a timeout issues appears if the time window w is exceeded, i.e.

$$\sum_{0 \leq i < k} t_i > w$$

and the final scenario state $s_f \in S$ has not been reached, i.e.

$$\forall 0 \leq i < k : s(i) \neq s_f$$

and no severe semantic issue (i.e. a property violation, a part collision, or a nondeterminism due to multiple behaviors, multiple bindings with conflicting dynamic channels, or a general fixed point) has been detected in the first k computation steps. The computation timeout issue is particularly useful in case the final state s_f can never be reached. However, the issue does not indicate such state reachability problems. To uncover state reachability problems, advanced model-checking techniques are necessary [ACJT96], which are not considered in this doctoral thesis. In particular, the technique has to take into account the collision-based binding mechanism.

6.3 Summary and outlook

This chapter introduced a number of quality issues, which can be evaluated fully automatically over the conceptual design models expressed with the modeling technique from the previous chapter (see Chapter 5). In particular, syntactic issues concerning the correct use of the modeling language (see Section 6.1) and semantic issues concerning the meaning of the design knowledge (see Section 6.2) were distinguished. Hereby, syntactic issues have been described using UML [BJR+96] specification patterns and a custom mathematical notation, while semantic issues have been described based on the revised formalism (see Chapters 4 and 5). The following chapter answers the question how prototypical tool support could look like for the presented modeling and quality assurance technique.

7 Prototypical tooling

In the following, the question is answered how the modeling technique from Chapter 5 and the quality issues from Chapter 6 can be implemented into a software tool. In particular, the question arises how an integrated user interface could look like that covers the modeling technique and the quality issues described previously. The proposed software design consists of a *modeling interface* (see Section 7.1) and a *testing interface* (see Section 7.2). In the following each interface is explained in more detail.

7.1 Modeling interface

The modeling interface integrates modeling and continuous syntactic quality checking features. A screenshot of the modeling interface is provided in Figure 7.1.

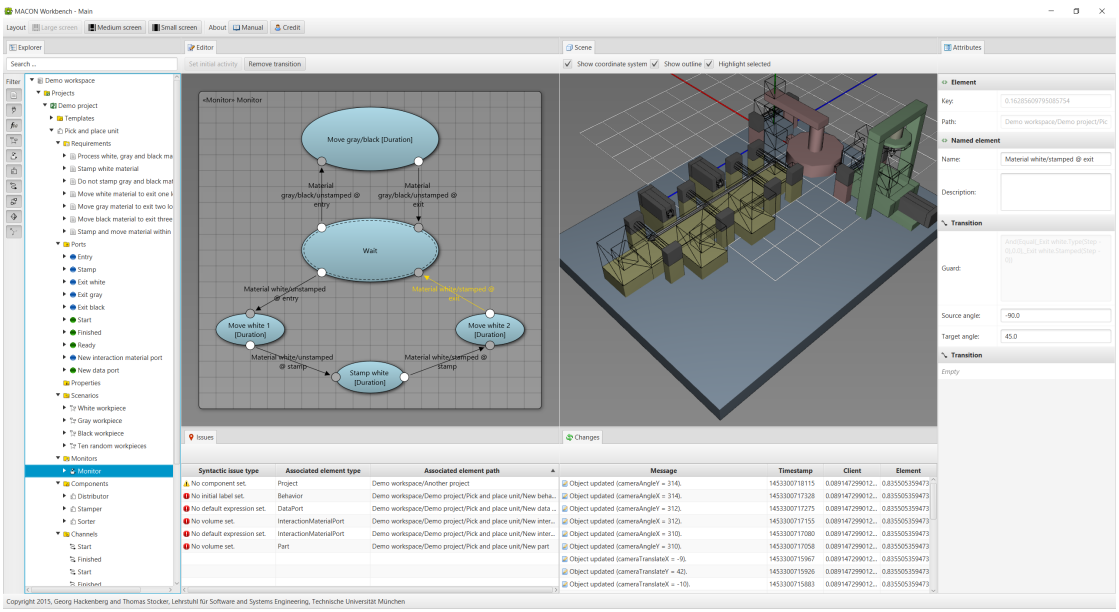


Figure 7.1: Modeling interface integrating modeling and syntactic quality checking.

The modeling interface consists of seven views each serving a different purpose during manufacturing systems modeling using the presented approach (see Chapters 3 to 6): The *toolbar view* (see Section 7.1.1), the *explorer view* (see Section 7.1.2), the *editor view* (see Section 7.1.3), the *scene view* (see Section 7.1.4), the *issues view* (see Section 7.1.5), the *changes view* (see Section 7.1.6), and the *attributes view* (see Section 7.1.7). In the following each view is described in detail including their purpose, their contents, supported user interactions, as well as events published and consumed. Note that the views interact with each other by means of events published to and received from a common event bus. This mechanism allows one to decouple the view implementations.

7.1.1 Toolbar view

Firstly, the toolbar view can be found at the top of the modeling interface. The toolbar view provides two functionalities, namely (1) changing the layout of the modeling interface and (2) providing access to information about the software tool and related resources. An enlarged screenshot of the toolbar view is provided in Figure 7.2.

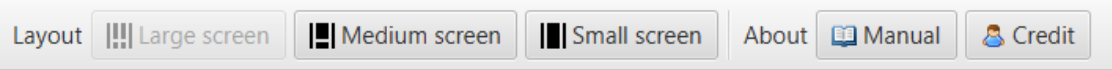


Figure 7.2: Toolbar view of the modeling interface.

The toolbar view is structured into the sections *layout* and *about*. The layout section provides buttons for three different layouts: A layout for large screens, a layout for medium screens, and a layout for small screens. The large screen layout defines six view areas (black boxes), while the medium screen only provides four view areas, and the small screen only contains three view areas. The button of the active layout is disabled (e.g. in the screenshot the large screen layout is active). In contrast, the about section contains two buttons, one for accessing the software manual, the other for viewing the credits (i.e. information about the developers as well as associated projects and organizations). The remaining space of the toolbar view is left empty.

7.1.2 Explorer view

Then, the explorer view can be found at the left of the modeling interface. The explorer view allows the user to browse and filter the model of the manufacturing system. Furthermore, the explorer view provides means to create new model elements, select existing model elements and delete previously created model elements. A screenshot of the explorer view is given in Figure 7.3.

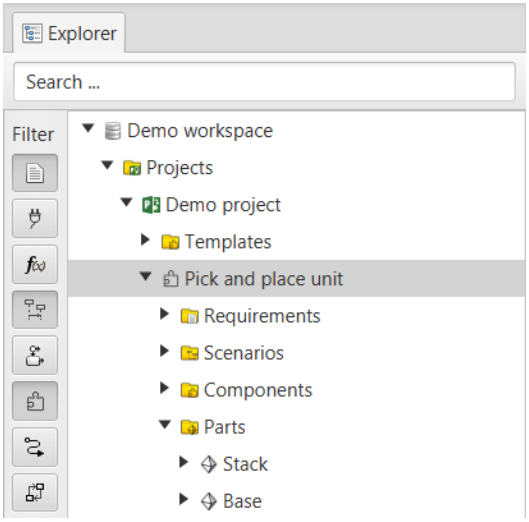


Figure 7.3: Explorer view of the modeling interface.

The explorer view is structured into three sections: A *top toolbar*, a *left toolbar*, and a center *tree view*. The top toolbar provides a text field for keyword-based searching and filtering the tree view contents. Hereby, the tree nodes remain visible in case their own or any descendant's name matches the keyword. The left toolbar provides toggle buttons for filtering the tree view contents according to the features of the modeling technique (i.e. requirements, ports, properties, scenarios, monitors, components, channels, behaviors, and parts; see Chapter 5). If some toggle button is active, the respective feature is visible. Otherwise, the feature is hidden from the display. Finally, the tree view shows the entire model of the manufacturing system starting with the workspace, its projects, templates (i.e. reusable component definitions), and main components. Templates and components contain folder nodes for each feature of the modeling technique. Then, each folder node contains the instances of the respective feature (e.g. the requirements node contains requirements for the superordinate component/template). At last, model elements can be created and deleted via the context menu of the tree nodes. In particular, folder nodes allow one to add subordinate instances of the respective feature.

7.1.3 Editor view

The editor view can be found in the middle top left of the modeling interface. The editor view contents depend on the element selected in the tree view of the explorer view (see

Section 7.1.2). Depending on the selection three different cases can be distinguished: (1) *Workspace, project, and component nodes*, (2) *scenario, monitor, and behavior* nodes, as well as (3) *component folder nodes*. In the following the editor view contents are described for each of these cases in more detail.

Workspace, project, and component nodes

For workspace, project, and component nodes the editor view allows one to generate a test report covering all subordinate components and scenarios. To generate the test report all scenarios are simulated and the results (i.e. success, failure, and timeout) are collected. The test report itself aggregates the results hierarchically. A screenshot of the editor view for workspace, project, and component nodes is provided in Figure 7.4.

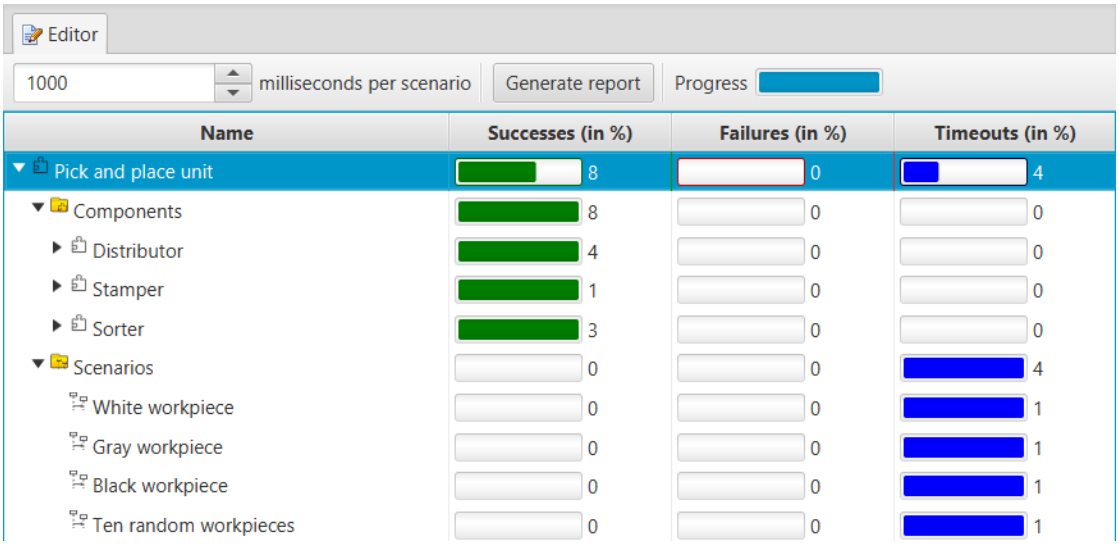


Figure 7.4: Editor view for workspace, project, and component nodes.

The editor view is structured into a top *toolbar* and a center *tree table view*. The top toolbar contains a number field for defining the maximum time in milliseconds, which is provided for each scenario simulation until a timeout issue is raised (see Section 6.2.2). Then, the top toolbar includes a button for starting the test report generation and a progress bar for tracking the progress of report generation. The progress is calculated from the total number of scenarios and the number of scenarios already processed. Finally, the tree table view shows the actual test report. The test report is structured hierarchically starting with the workspace, project, or component selected in the explorer

view (see Section 7.1.2). For each component node the tree table view also includes subordinate component and scenario folder nodes. Finally, for each tree table node also the aggregate number of subsidiary successful, failed, and timed-out test executions is visualized. The aggregate numbers are intended as a measure of implementation progress (i.e. the more scenarios can be completed successfully, the larger is the progress is).

Scenario, monitor, and behavior nodes

For scenario, monitor, and behavior nodes the editor view provides different variants of a graphical state machine editor showing the executables' labels and transitions (see Section 5.1.2). The graphical editors allow one to select states and transitions. Furthermore, the position of states as well as the out- and ingoing positions of transitions can be changed. Finally, transitions between states can be added and removed. A screenshot of the editor view for scenario, monitor and behavior nodes is depicted in Figure 7.5.

Figure 7.5: Editor view for scenario, monitor, and behavior nodes.

The editor contains a top *toolbar* and a center *diagram area*. The toolbar provides buttons for setting initial (and potentially final) states and removing transitions. The buttons are enabled depending on and act upon the states and transitions currently selected in the diagram area. The diagram area itself provides a graphical representation of the executable. States are shown as labeled ellipses and transitions are depicted as labeled lines with arrow endings. Furthermore, for each transition small circular markers are added to the source and the target states. Selected states can be moved around the

diagram area and selected transition markers can be moved around the ellipse borders of the respective state. Both the diagram area and the ellipse borders define a grid to which the moved objects snap unless the `Alt` key is pressed. Finally, new transitions can be created by pressing the `Ctrl` key, pressing the left mouse button on the desired source state, dragging the mouse to the desired target state, and releasing the left mouse button. More advanced interactions are not supported currently.

Component folder nodes

For component folder nodes the editor view provides a graphical component diagram editor showing the subcomponents (see Section 5.2.1), ports (see Section 5.2.2), and channels (see Section 5.2.3) of the superordinate component as well as the ports of subcomponents and material interaction ports. Furthermore, the editor supports creating and removing channels as well as rearranging the subcomponents, ports, and channels. A screenshot of the editor view for component feature nodes is provided in Figure 7.6.

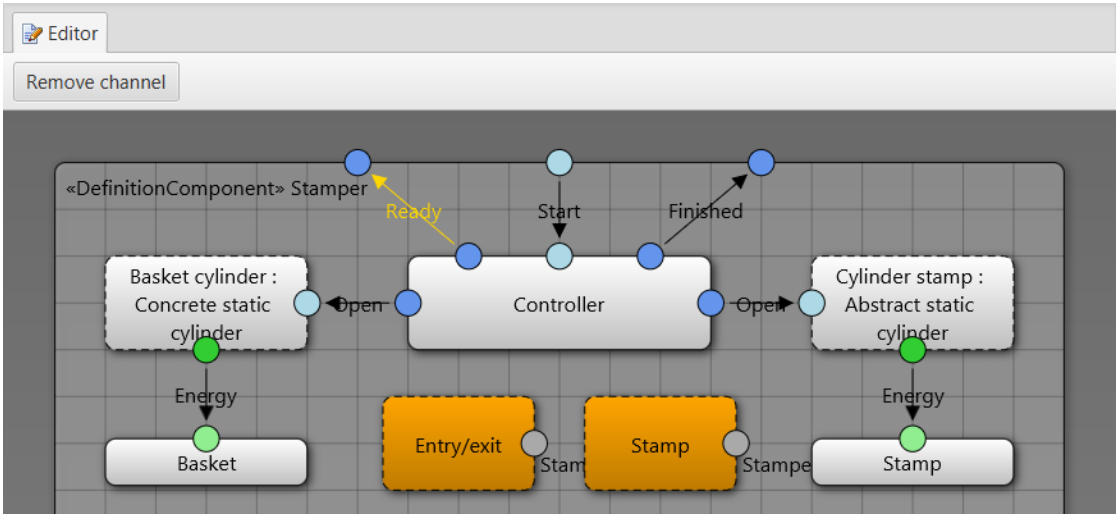


Figure 7.6: Editor view for component feature nodes.

Again, the editor view is divided into a top *toolbar* and a center *diagram area*. The toolbar provides one button for removing channels. The button is enabled based on the selection in the diagram area. In contrast, the diagram area shows the system boundary of the superordinate component including its ports. Then, the subcomponents and material interaction ports are displayed as labeled gray and orange rectangles. Note that template instances (such as the basket cylinder or the stamp cylinder in Figure 7.6) are

distinguished by dashed borders. Finally, the diagram area represents the channels between the ports of the superordinate component, the subcomponents, and the material interaction ports with labeled lines and arrow endings. All, components, ports and channels can be selected by clicking with the left mouse button. Then, components and ports also can be moved by pressing the left mouse button and dragging the object around. Again, the `Alt` key can be used to disable grid snapping on the diagram background and at rectangle borders. Finally, new channels can be created by pressing the `Crtl` key, pressing the left mouse button on the source port, dragging the mouse to the target port, and release the left mouse button.

7.1.4 Scene view

Then, the scene view can be found in the middle top right of the modeling interface. The scene view is responsible for displaying the volumes of material ports and (physical) parts as well as respective transforms, which can be found underneath the component selected in the explorer view (see Section 7.1.2). Furthermore, ports, parts, and specific volumes can be highlighted. A screenshot of the scene view is provided in Figure 7.7.

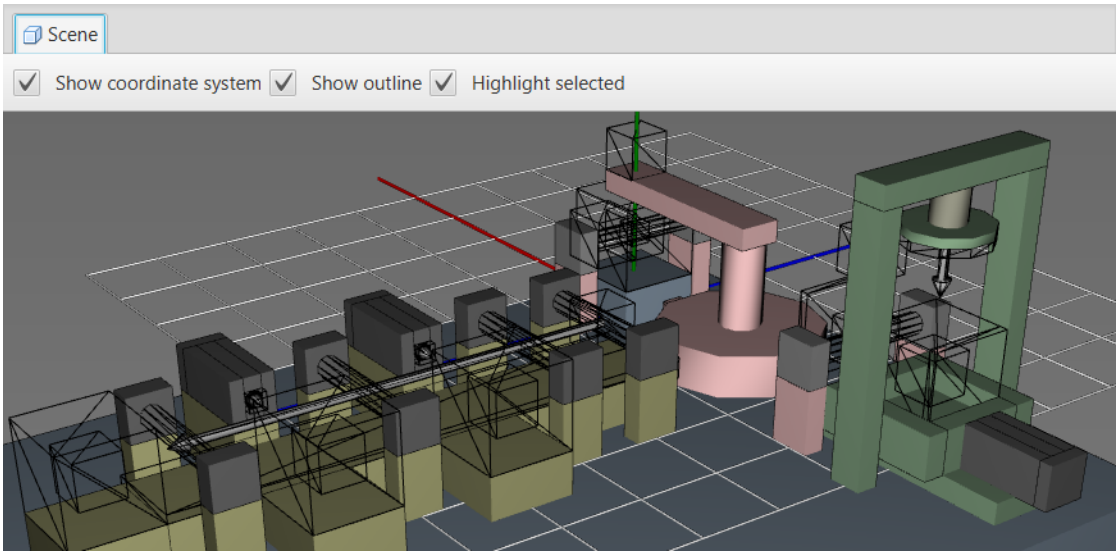


Figure 7.7: Scene view of the modeling interface.

The scene view consist of a top *toolbar* and a center *canvas*. The toolbar provides three check boxes for enabling or disabling respective options. The first option concerns the display of the coordinate system. The second option allows one to show and hide

the outline of part volumes. And the third option can be used to enable or disable the highlighting. In contrast, the canvas displays the part volumes as solid shapes and material port volumes as wire frame shapes respecting associated transforms. If the coordinate system option is enabled, the canvas also displays the x-, y-, and z- axes as well as a grid in the x-z-plane. Then, if the outline option is enabled, the canvas displays black borders around each (phyiscal) part volume. Finally, if the highlight option is enabled, the canvas contents are shaded or lightened based on the selection in the explorer view (see Section 7.1.2) and the editor view (see Section 7.1.3). For example, in Figure 7.7 all part volumes of the pink crane component are highlighted. Alternatively, single part and material port volumes can be highlighted as well. Consequently, the spatial elements of the system model can be identified more easily. At last, the mouse can be used for changing the camera perspective. A separate perspective is stored for each component. The left mouse button can be used for rotating, the right mouse button for translating, and the mouse wheel can be used for zooming the perspective.

7.1.5 Issues view

The issues view can be found in the middle bottom left of the modeling interface. The issue view is responsible for displaying the active syntactic issues (see Section 6.1) to the user. Furthermore, the user is able to sort the active issues by different aspects such as the issue message, the type of the associated modeling element, and the path to the associated element. A screenshot of the issue view is given in Figure 7.8.

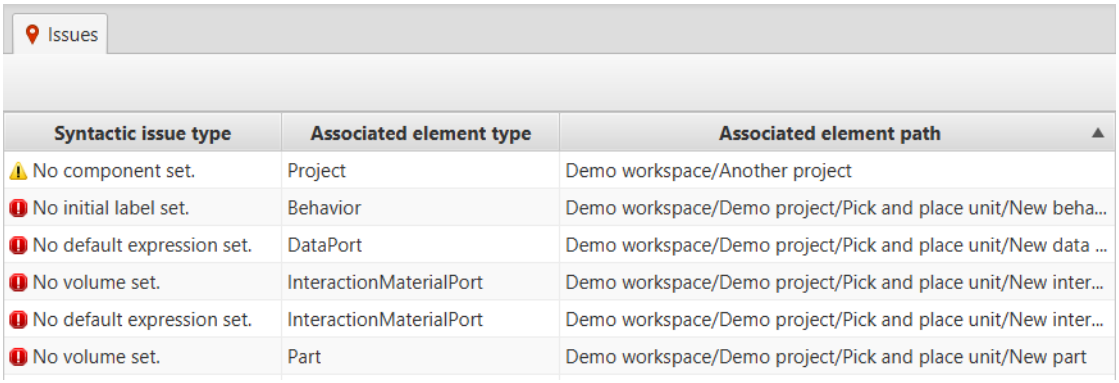


Figure 7.8: Issue view of the modeling interface.

The issues view consists of a *table view* with three columns: The *syntactic issue type*, the *associated element type*, and the *associated element path*. The rows of the table repre-

sent the active syntactic issues. Issues are added to the table as soon as they are detected in the model and removed from the table as soon as they are resolved. Technically, each (potential) syntactic issue is represented as an object, which acts as an observer over the model and reevaluates itself every time the underlying data changes. Finally, in the table syntactic defects (i.e. severe issues) and syntactic deficiencies are distinguished using icons. Note that defects have to be resolved by the user, while deficiencies can be left untouched, which is why this visual distinction has been introduced.

7.1.6 Changes view

The changes view can be found in the middle bottom right of the modeling interface. The view displays a chronological stream of changes (i.e. change events), which have been made on the model by any connected software client. Possible events are the creation of new model elements, the modification of element attributes, and the deletion of model elements. A screenshot of the change view is provided in Figure 7.9.

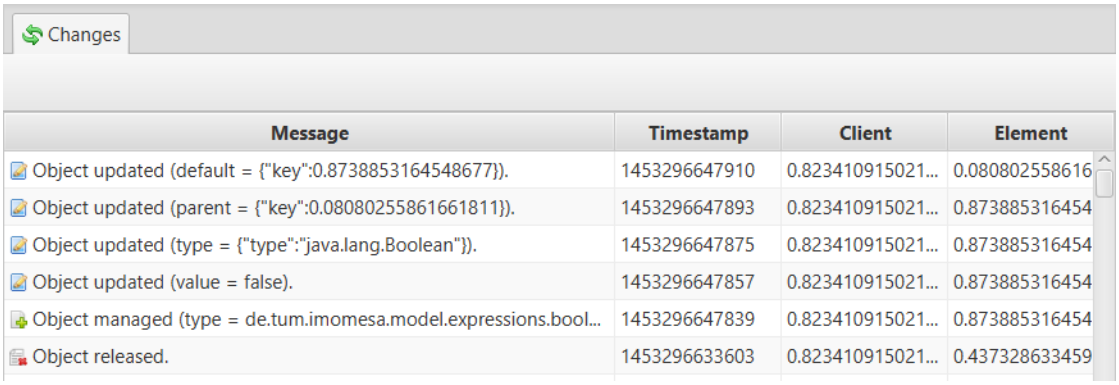


Figure 7.9: Change view of the modeling interface.

The changes view consists of a *table view* with four columns: The change *message*, the change *timestamp*, the key of the software *client* causing the change, and the database key of the model *element*, to which the change applies. Then, each row of the table represents one change event in the stream of change events. The different types of events (i.e. model element creation, element attribute modification, and element deletion) are distinguished with icons. Technically, every change to the model is pushed to a central database server, which stores the event information into a log file and synchronizes all connected clients automatically. Consequently, for each model the entire change history is persisted and can be analyzed. Furthermore, collaboration between different engineers

and, hence, concurrent engineering [WSX+02] is facilitated in principle. However, note that this research direction is not investigated further in this doctoral thesis. Rather, the prototypical tooling focuses on the single user case.

7.1.7 Attributes view

Finally, the attributes view can be found at the right of the modeling interface. The attribute view is responsible for displaying the attributes of the last object selected in the explorer view (see Section 7.1.2) or the editor view (see Section 7.1.3). Furthermore, the attribute view allows one the change the values of certain attributes. A screenshot of the attribute view is shown in Figure 7.10.

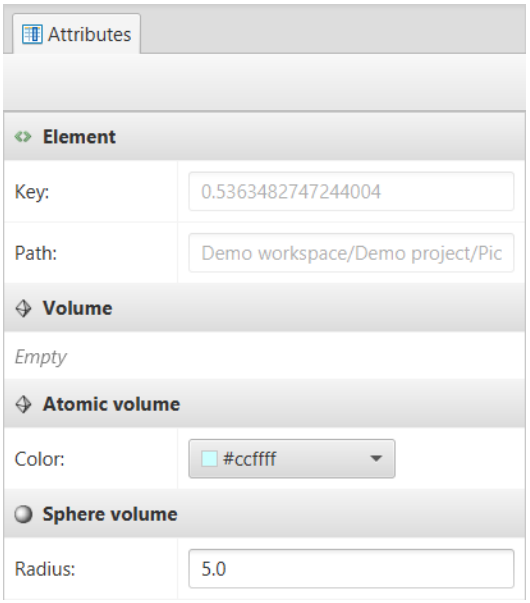


Figure 7.10: Attribute view of the modeling interface.

The attributes view is structured into several *sections* and one attribute *table* per section. Each section represents one class in the class hierarchy of the selected design object (e.g. `Element`, `Volume`, and `AtomicVolume` in the depicted case). Then, the rows of the section tables represent the attributes associated with the respective class (e.g. *key* and *path* for the class `Element` or *color* for the class `AtomicVolume`). Hereby, the first table column shows the attribute name, while the second column provides a control for its value (e.g. text fields for the *key* and *path* attributes of the class `Element`

or color choosers for the *color* attribute of the class `AtomicVolume`). Technically, the attribute view is realized using table templates for each class of design object introduced in Chapter 5. Furthermore, reflection is used for selecting and displaying the right table templates. This mechanism allows one to reuse large parts of the code and to integrate new modeling elements easily with the existing implementation.

7.2 Testing interface

The testing interface integrates spontaneous (i.e. user-triggered) semantic quality checking capabilities into the tool, which has been explained formally in Section 6.2. A screenshot of the testing interface is provided in Figure 7.11.

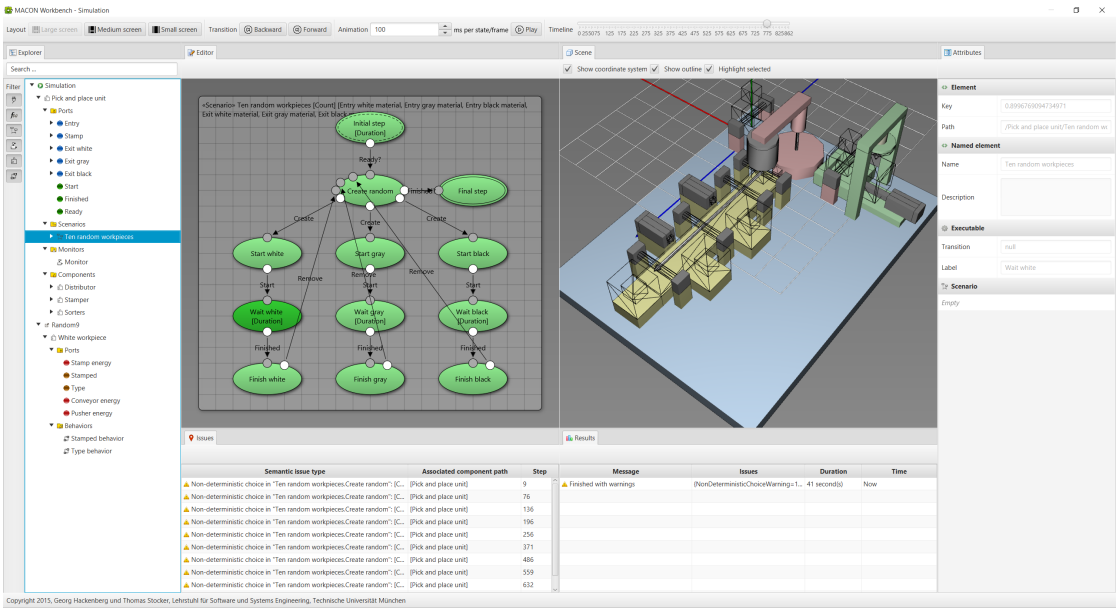


Figure 7.11: Testing interface integrating semantic quality checking.

Analogous to the modeling interface (see Section 7.1), the testing interface consists of seven views each serving a different purpose: The *toolbar view* (see Section 7.2.1), the explorer view (see Section 7.2.2), the *editor view* (see Section 7.2.3), the *scene view* (see Section 7.2.4), the *issues view* (see Section 7.2.5), the *results view* (see Section 7.2.6), and the *attributes view* (see Section 7.2.7). In the following, each view is described in detail including their purpose, contents, user interactions, as well as events published

and consumed. Note that, again, the views interact by means of events published and received over a common event bus to decouple their implementations.

7.2.1 Toolbar view

The toolbar view can be found at the top of the testing interface. The toolbar provides four functionalities: (1) Changing the layout of the testing interface, (2) stepping forward and backward between the simulation steps, (3) animating the simulation steps, and (4) jumping between the simulation steps. A screenshot of the toolbar view is given in Figure 7.12 cropping contents at the left and right sides due to space limitations.

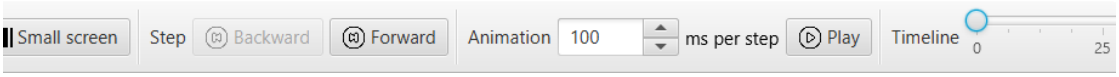


Figure 7.12: Toolbar view of the testing interface.

The toolbar view is structured into the sections *layout*, *step*, *animation*, and *timeline*. Analogous to the toolbar view of the modeling interface (see Section 7.1.1), the layout section provides buttons for switching between a large screen, a medium screen, and a small screen layout. In the large screen layout a separate screen region is assigned to each view of the testing interface. In the medium screen layout the editor view (see Section 7.2.3) and the scene view (see Section 7.2.4) as well as the issue view (see Section 7.2.5) and the result view (see Section 7.2.6) share a common region, and in the small screen layout the previous four views share the same screen region. In contrast, the step section provides two button for stepping forward and backward in the simulation. The backward button is disabled at the first simulation step, while the forward button is disabled at the last simulation step (i.e. when a timeout, a severe semantic issue, or the final step of the scenario is reached). Then, the animation section provides a number field for adjusting the number of milliseconds used per simulation step and a play/pause button. When the animation is active the simulation step is incremented automatically after the defined number of milliseconds. Note that the simulation step can be incremented only in case the simulation engine has calculated the respective step information in the background. If the step information is not available, the animation waits for another period of milliseconds. When the end of the simulation is reached, the animation stops automatically. Finally, the timeline section provides a number slider for jumping between the simulation steps. The slider interval ranges from the first simulation step to the last simulation step, which has been calculated in the background by the simulation engine. Consequently, the slider interval increases constantly until the end of the simulation is reached.

7.2.2 Explorer view

Then, the explorer view can be found at the left of the testing interface. The explorer view provides access to the model elements, which are relevant for the current simulation run (in particular the components as well as their ports, properties, monitors, behaviors, and scenarios). Moreover, the explorer view allows one to select the respective model elements to obtain further information in the other views of the testing interface. A screenshot of the explorer view is shown in Figure 7.13.

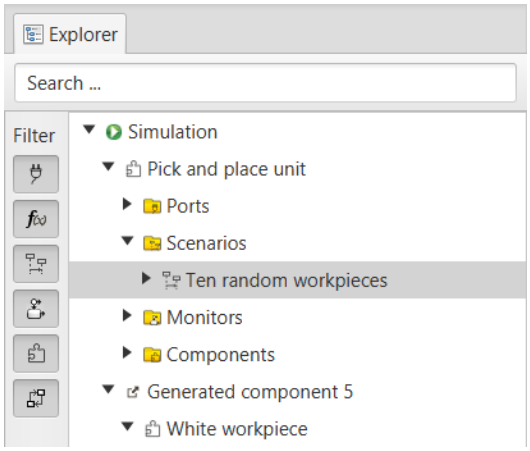


Figure 7.13: Explorer view of the testing interface.

The explorer view consists of a *top toolbar* and a *left toolbar* as well as a center *tree view*. The top toolbar provides a text field for keyword-based searching and filtering the tree view contents. Similar to the explorer view of the modeling interface (see Section 7.1.2), the tree nodes remain visible in case their label or the label of any descendant tree node matches the search string. Then, the left toolbar contains toggle buttons for filtering the tree view contents according to the features of the modeling technique (i.e. ports, properties, scenarios, monitors, components, and behaviors). Note that the other features (i.e. requirements and parts) are not displayed during simulation because they are not affected by the system state. Finally, the tree view itself shows a simulation node at its root containing the main component (e.g. the *pick and place unit* in Figure 7.13) as well as template instances generated at runtime (e.g. *Generated component 5* in Figure 7.13). Then, each component node contains folder nodes for the features relevant during simulation. In turn, the folder nodes contain the respective child elements of the superordinate component. At last, each template instance node (e.g. *Generated component 5*) contains a node for the respective component template.

7.2.3 Editor view

Subsequently, the editor view can be found at the middle top left of the testing interface. The editor view contents depend on the type of node selected in the explorer view (see Section 7.2.2). Different implementations are provided for observation nodes, scenario, monitor, and behavior nodes as well as component folder nodes. The different implementations are explained in the following.

Observation nodes

For observation nodes (i.e. ports of components, material interaction ports, and scenarios as well as variables of scenarios, monitors, and behaviors) the editor view displays the observation valuations over time as well as their relative frequencies. The visualizations are intended to help understanding the system dynamics. A screenshot of the editor view for observation nodes is provided in Figure 7.14.

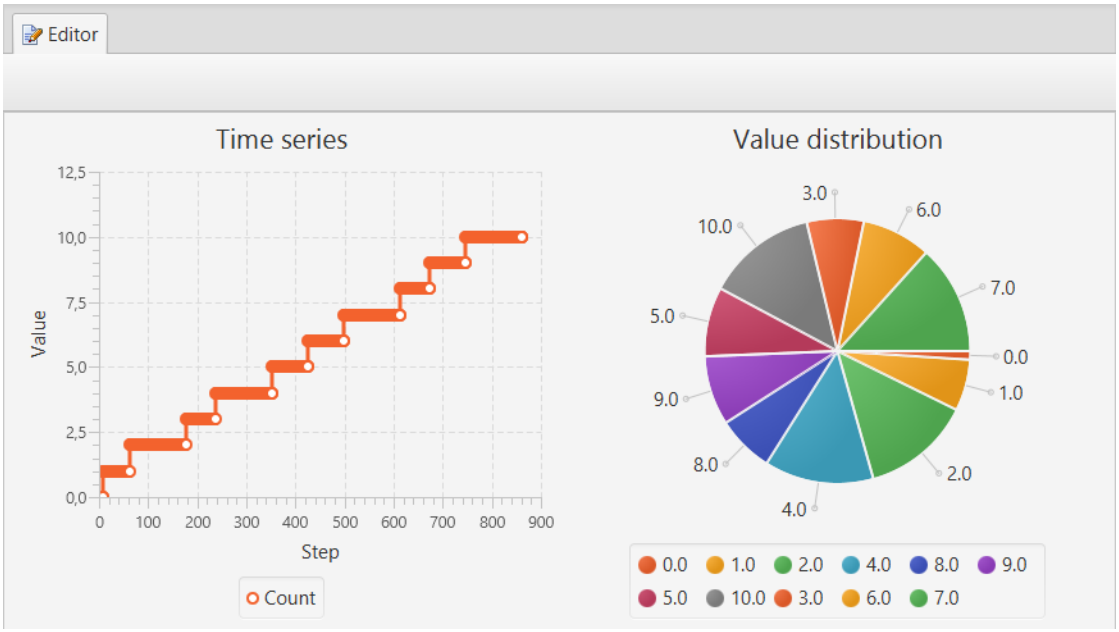


Figure 7.14: Observation view of the testing interface.

The editor view is divided into two charts, namely the *time series chart* and the *value distribution chart*. The time series chart depends on the data type of the observation. For numeric observations, matrix-valued observations, and set-valued observations a line

chart is presented (i.e. the data points are connected), while for boolean observations a scatter chart is provided instead (i.e. the data points remain unconnected). The x-axis of the time series charts represents the simulation step and the y-axis depicts the observation valuations. For matrices the norm of the largest eigenvector is used as a measure of the matrix effect, while for sets the set cardinality is displayed. For numbers and booleans the original value is depicted instead. Finally, the value distribution chart shows the relative frequencies of the possible observation valuations across all simulation steps. For the value distribution chart all original values are used instead. Consequently, matrices can be distinguished also by their individual coefficients.

Scenario, monitor, and behavior nodes

For scenario, monitor, and behavior nodes the editor view shows the same state machine diagram as the editor view of the modeling interface (see Section 7.1.3). However, this time interaction with the diagram elements is disabled. Instead, the diagram view is used to display the active states and transitions of the underlying executable. A screenshot of the editor view for scenario, monitor, and behavior nodes is given in Figure 7.15.

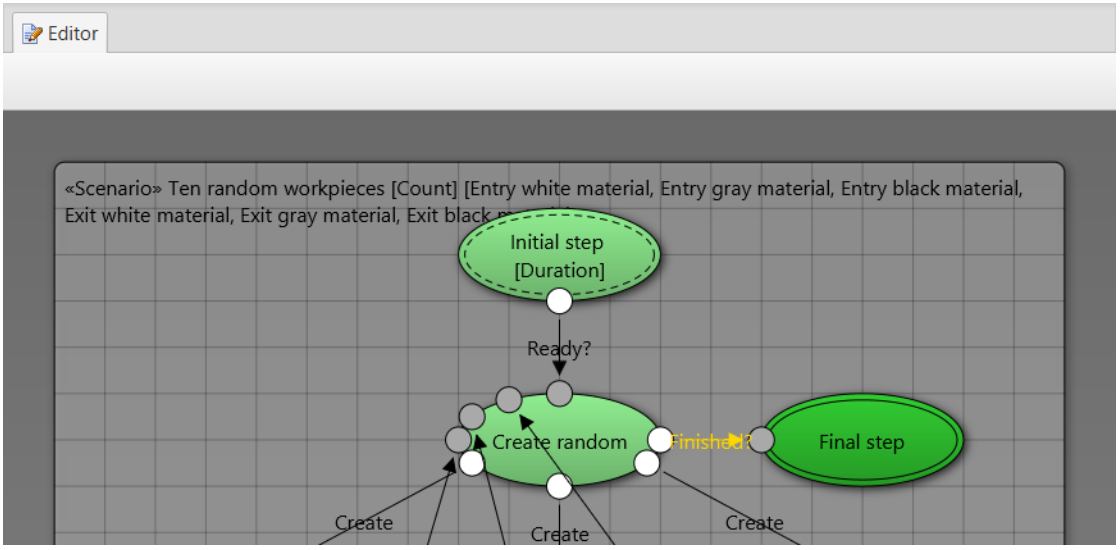


Figure 7.15: Scenario view of the testing interface.

The editor view only contains one single part, namely the *diagram area*. The diagram area shows a boundary box for the underlying scenario (see Section 5.3.4), monitor (see Section 5.3.3), or behavior (see Section 5.2.5). At the top of the boundary box the name

of the state machine as well as contained variables (and material entry/exit ports in the case of scenarios) are displayed. Then, inside the box the states are depicted as ellipses and the transitions are visualized as lines with arrow ending. Finally, the active state and (if available) the active transition of the current simulation step are highlighted. For example, in Figure 7.15 the final step of the scenario is reached through the transition *Finished?*. Note that when a transition is executed in the current simulation step, the active state is given by the target state of the executed transition. However, it is also possible that no transition can be executed in the current simulation step such that only the active state exists.

Component folder nodes

For component folder nodes the editor view shows the same component diagram as the editor view of the modeling interface (see Section 7.1.3). Again, interaction with the diagram elements is disabled during simulation. Instead, the diagram view is used to display the port and channel assignments for the current simulation step. A screenshot of the editor view for component folder nodes is provided in Figure 7.16.

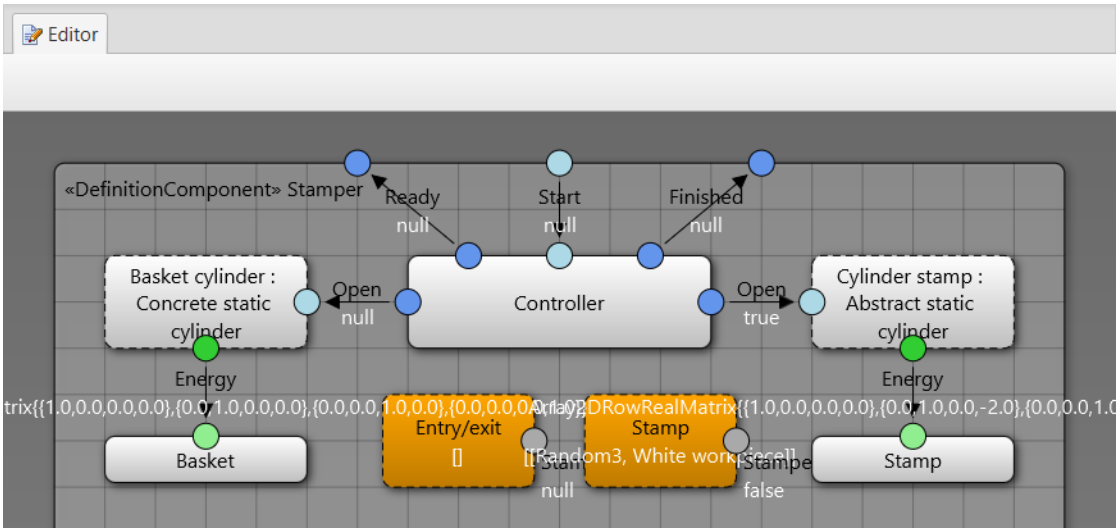


Figure 7.16: Components/channels view of the testing interface.

The editor view only consists of a single *diagram area*. The diagram area shows a boundary box for the superordinate component. At the top left of the boundary box the name of the superordinate component is displayed (e.g. *Stamper* in the upper case).

Then, on the border of the boundary box the ports of the superordinate component are shown (e.g. *Ready*, *Start*, and *Finished* in Figure 7.16). The interior of the boundary box depicts the subcomponents and material interaction ports as labeled rectangles including their own child ports. The label of the material interaction ports includes the set of components bound to the port in the current simulation step (e.g. none for the *Entry/Exit* port and *White workpiece* for the *Stamp* port in Figure 7.16). Finally, the diagram shows the channels between the ports as labeled lines with arrow ending. Again, the line labels include the channel assignments in the current simulation step, which is equal to the port assignments of the connected ports. In case a port remains unconnected in the diagram, the port assignment is displayed next to the port name.

7.2.4 Scene view

The scene view can be found at the middle top right of the testing interface. The scene view is responsible for displaying the material ports and the parts of the components. Note that during simulation the position and orientation of the components depends on the kinematic energy transmitted to the components up to the current simulation step. A screenshot of the scene view is provided in Figure 7.17.

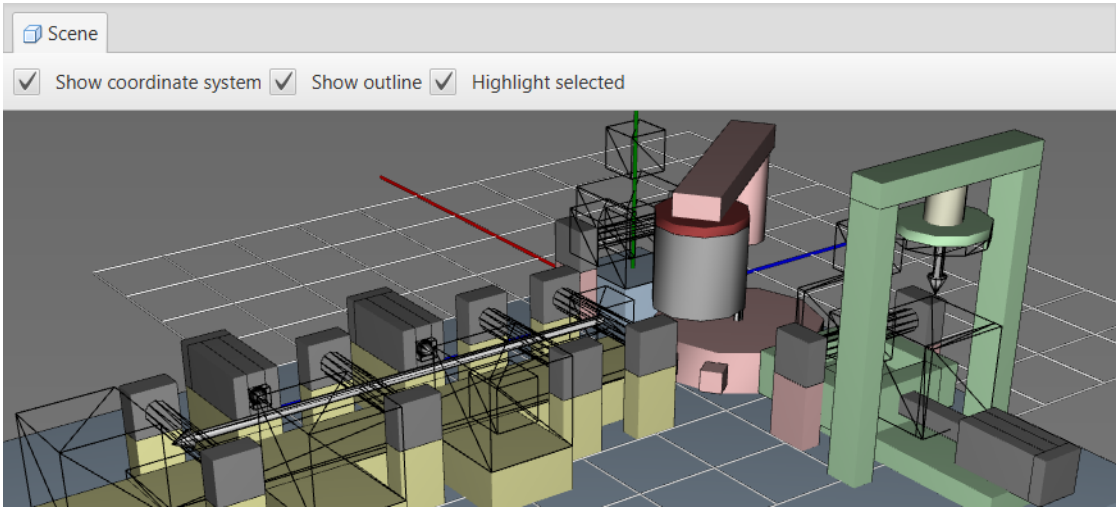


Figure 7.17: Scene view of the testing interface.

The scene view is divided into a top *toolbar* and a center *canvas*. The top toolbar provides three check boxes. The first checkbox allows one to enable and disable the display of the coordinate system. The second checkbox can be used to show and hide the outline

of physical parts. And the third checkbox controls whether model elements selected in the explorer view (see Section 7.2.2) and the editor view (see Section 7.2.3) should be highlighted. In contrast, the canvas displays an optional coordinate system as well as the material interaction ports and the parts of the selected component and all its sub-components. The coordinate system includes three colored axes as well as a grid in the x-z-plane. The material interaction ports are displayed as wire-framed volumes, while the physical parts are shown as solid shapes with optional black outline. Furthermore, if selection highlighting is enabled, the color of parts is darkened or lightened based on the current selection in the explorer or the editor views. Finally, the canvas implements mouse-based interaction for changing the camera perspective (see Section 7.1.4).

7.2.5 Issues view

The issues view can be found can be found at the middle bottom left of the testing interface. The issues view is responsible for showing the semantic issues (see Section 6.2), which have been discovered during the current simulation run. Note that simulation does not guarantee to find all semantic issues that are present in the model. A screenshot of the issue view is provided in Figure 7.18.

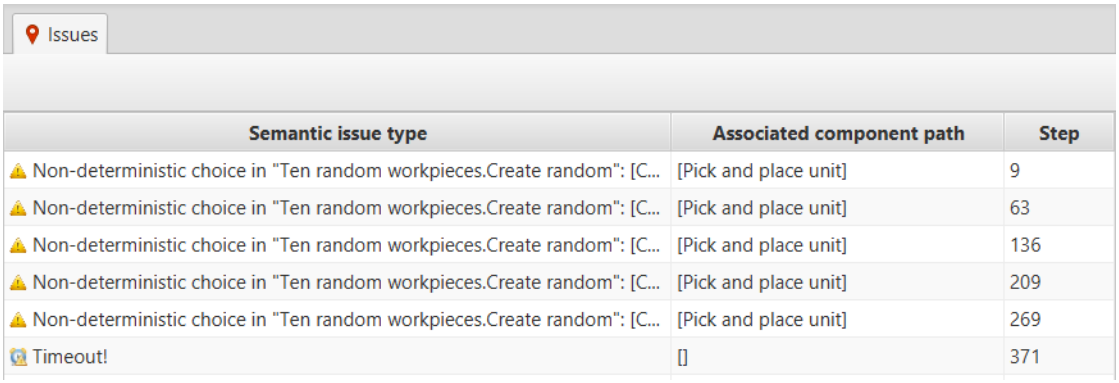


Figure 7.18: Issue view of the testing interface.

The issues view only contains a *table view* with three columns: The first column displays the type of the semantic issue, the second column shows the path of the associated component, and the third column denotes the simulation step during which the semantic issue has been discovered. Then, the rows of the table represent the semantic issues, which have been discovered so far during the current simulation run. Note that the list of semantic issues might grow while the simulation engine in running in the background.

For example, in the above case multiple non-determinism issues (see Section 6.2.2) in the scenario *Ten random workpieces* of the component *Pick and place unit* as well as one timeout issue (see Section 6.2.2) have been discovered. Finally, for each semantic issue an icon is provided indicating the severity of the issue. Hereby, three cases are distinguished: Semantic deficiencies that can be resolved, semantic defects that must be resolved, and timeouts. A warning sign is used for deficiencies, a stop sign is used for defects, and a clock icon is used for timeouts.

7.2.6 Results view

The results view can be found at the middle bottom right of the testing interface. The results view is responsible for displaying a summary of the current and all previous simulation runs of the selected scenario. Consequently, the view provides a historical perspective on the simulation activity for each scenario of the system model. A screenshot of the result view is given in Figure 7.19.

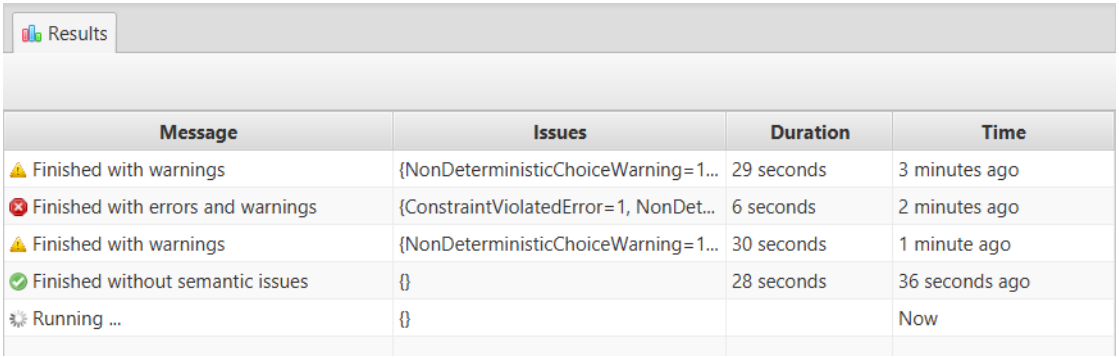


Figure 7.19: Result view of the testing interface.

The results view only contains one *table view* with four columns: The first column shows a message for each simulation run. Initially, the message says that the simulation is running. When the simulation terminates, the message says whether the simulation could be terminated successfully and whether semantic issues (see Section 6.2) have been discovered. Additionally, the message column contains an icon illustrating the result of the simulation run. A warning sign is used in case semantic deficiencies were found, an error sign is displayed in case at least one semantic defect (i.e. a severe semantic issue that has to be resolved) has been detected, and a timeout sign is depicted in case the simulation could not be completed within the prescribed time window. In contrast, the second column summarizes the semantic issues, which have been discovered during each

simulation run. In particular, counts for each type of semantic issue are provided. Then, the third column displays the time, which has been required to finish each simulation run (e.g. between 6 and 30 seconds in Figure 7.19). Finally, the fourth column depicts the time that has passed since executing the different simulation runs (e.g. between 36 seconds and 3 minutes in Figure 7.19).

7.2.7 Attributes view

The attributes view can be found at the right of the testing interface. The attributes view shows detailed information about the last element selected in the explorer view (see Section 7.2.2) or the editor view (see Section 7.2.3). The displayed information includes static model data as well as dynamic simulation data highlighted with yellow background color. A screenshot of the attributes view is provided in Figure 7.20.

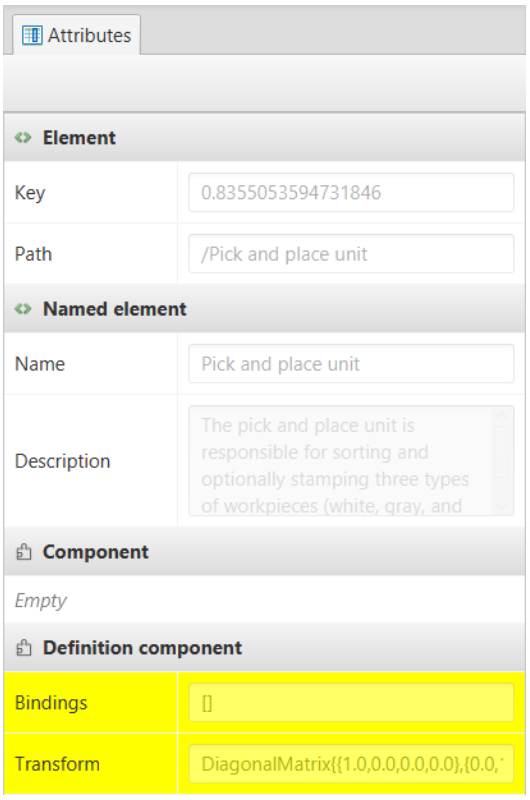


Figure 7.20: Attributes view of the testing interface.

The attributes view contains a single *table view* showing relevant static and dynamic attributes of the selected element. For example, in the above case the static *key*, *path*, *name*, and *description* as well as the dynamic *bindings* and *transform* attributes of the *Pick and place unit* component are shown. Note that the static attributes have been defined during the specification activities of the test-driven design method (see Chapter 3) using the modeling interface (see Section 7.1), while the dynamic attributes relate to the current simulation step during test-based verification. In particular, the bindings attribute shows to which material interaction port the selected component is bound (i.e. none in the upper case). The transform attribute shows the position and orientation of the selected component in the world coordinate system instead (i.e. the identity transform in Figure 7.20). Note that the position and orientation of each component C depends on the kinetic energy input history $i_M \in \vec{I}_M$ and the kinetic energy forwarding history $i_F \in \vec{I}_F$ (see Section 5.2.1). Finally, the attributes are grouped by the classes of the inheritance hierarchy. E.g., as shown in Figure 7.20, the class `DefinitionComponent` inherits from the class `Component`, which extends the class `NamedElement`, which finally originates from the class `Element`. Exploiting the inheritance hierarchy also enables reuse of user interface elements and improves the extensibility of the software tool.

7.3 Summary and outlook

This chapter presented a prototypical tool support for the modeling technique (see Chapter 5) and the quality issues (see Chapter 6). In particular, the results showed how to integrate conceptual design information from customer requirements and manufacturing processes to mechatronic architectures and software behaviors into a common modeling interface (see Section 7.1). Furthermore, the prototypical tool continuously monitors the syntactic quality of the model and displays immediate feedback in the modeling interface. Then, the testing interface was explained (see Section 7.2), which provides access to test execution results and displays semantic issues, which have been discovered. Subsequently, the test-driven design method, the modeling technique, the quality issues, and implicitly the prototypical tooling are evaluated based on an industry-close showcase. Therefore, the following chapter first presents the conceptual design model obtained during the experiment.

8 Industry-close showcase

For evaluation purposes the test-driven design method (see Chapter 3), the revised and extended modeling technique (see Chapter 5), the quality assurance mechanisms (see Chapter 6), and the prototypical tooling (see Chapter 7) are applied to an industry-close showcase: The *pick and place unit* developed at the Institute of Automation and Information Systems, Prof. Dr.-Ing. Birgit Vogel-Heuser, Technische Universität München[1]. The study is based on a detailed description of the pick and place unit provided in [VHLFF14]. The description includes several stages of expansion of the pick and place unit, from which the most complex one is selected. The configurations, on the other hand, are explained in terms of their geometry, their structure (using SysML [FMS14] block definition diagrams) as well as their behavior (using SysML [FMS14] state machine diagrams). Figure 8.1 depicts the assembled unit deployed at the institute.

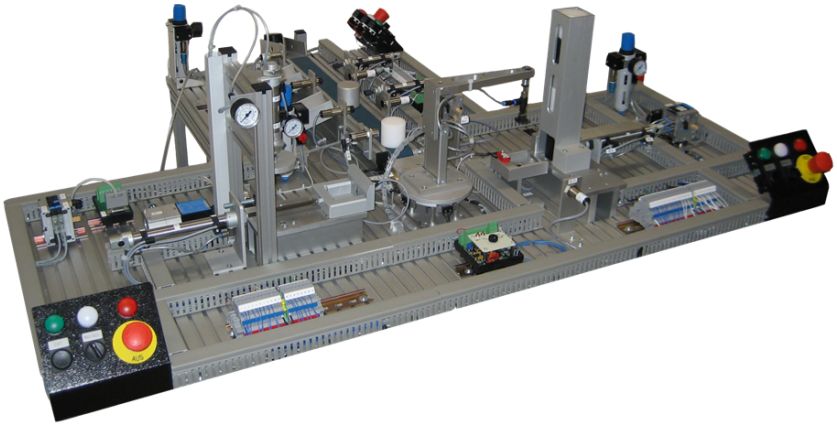


Figure 8.1: Pick and place unit [HCL+14].

In the following, the obtained model of the pick and place unit is explained in terms of its (reusable) template component definitions in Section 8.1 and its (non-reusable) system component definitions in Section 8.2. For each template and system component definition the requirements, ports, properties, scenarios, monitors, subcomponents/channels,

[1] https://www.ais.mw.tum.de/en/homepage/

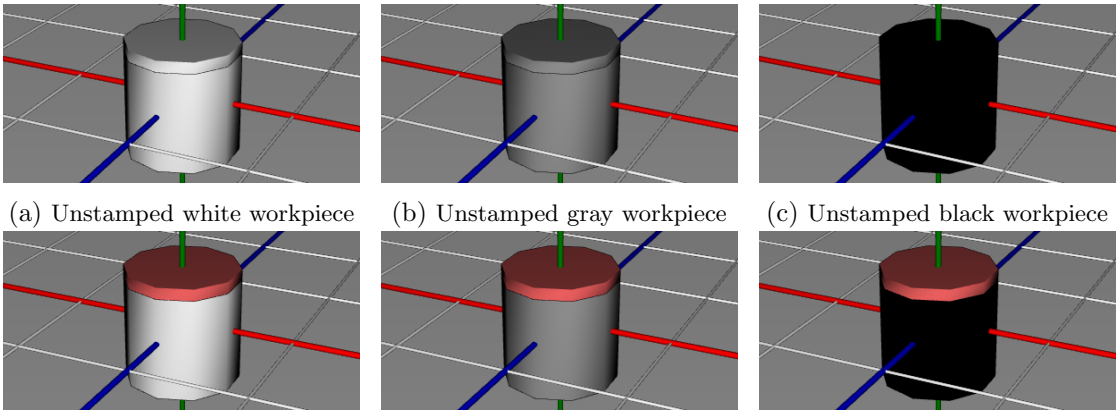
behaviors, and parts are described. This chapter is intended mainly for demonstrating the modeling technique in practice. Furthermore, the chapter resembles a possible design document, which can be obtained with the proposed approach. Note that in the next chapter a more thorough discussion of the suitability of the test-driven method, the validity of the model, and the relevancy of the quality issues is provided.

8.1 Templates

The model of the pick and place unit contains the following template component definitions: *White, gray and black workpiece* (see Section 8.1.1), *component sensor* (see Section 8.1.2), *workpiece sensor* (see Section 8.1.3), *abstract static cylinder* (see Section 8.1.4), *concrete static cylinder* (see Section 8.1.5), and *concrete dynamic cylinder* (see Section 8.1.7). In the following each template is explained briefly in terms of its geometry and its internal structure. Further details are omitted due to space limitations.

8.1.1 White/gray/black workpiece

The pick and place unit processes three types of workpieces, which are distinguished by their colors (i.e. white, gray and black). Depending on their type the pick and place unit is required to sort and optionally stamp the workpieces. Figure 8.2 provides the scene views for the possible combinations of workpiece types and states.



(a) Unstamped white workpiece (b) Unstamped gray workpiece (c) Unstamped black workpiece

(d) Stamped white workpiece (e) Stamped gray workpiece (f) Stamped black workpiece

Figure 8.2: Workpiece scene views.

The different workpiece models themselves consist of two energy input ports for moving

and stamping workpieces as well as two generic output ports for observing the workpiece type (i.e. white, gray, or black) and state (i.e. unstamped or stamped). Then, the models comprise two behaviors, one for describing the state output and the tip part (i.e. the white, gray, black, or red cylinder on the top) in reaction to the stamp energy input, the other for describing the constant type output. Finally, the models include a base part (i.e. the white, gray, or black cylinder on the bottom), which remains constant.

8.1.2 Component sensor

Then, for sensing component presence the model of the pick and place unit includes a (generic) component sensor template shown in Figure 8.3. Note that a very basic sensor model is provided, which neglects physical parts and therefore potential collisions within its environment. In practice, one can imagine several realizations such as circuit breakers, which require a physical contact between the sensor and the observed component, or light barriers, which do not require a physical contact.

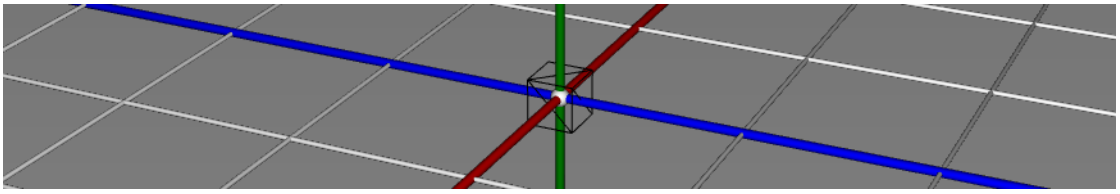


Figure 8.3: Generic sensor scene view.

The sensor model contains a cuboid material interaction port representing the observed physical space as well as a boolean data output port for reporting collisions with this physical space. The material interaction port, in turn, does not require any specific syntactic interface and, hence, binds all possible components during execution. Finally, the sensor behavior reports at time point $n \in \mathbb{N}$ whether collisions with the material interaction port $\beta \in B$ can be observed based on the cardinality $|a(\beta)(n)|$ of the material port activation history $a \in \vec{A}(B \cup Y \cup X)$.

8.1.3 Workpiece sensor

While the component sensor from Section 8.1.2 provides general means for sensing component presence at some physical space, the workpiece sensor reacts to instances of the workpiece template from Section 8.1.1 only. Consequently, a slightly more complex sensor model was developed including basic physical parts as shown in Figure 8.4.

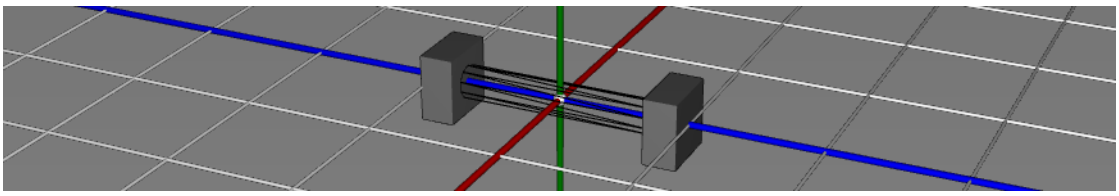


Figure 8.4: Workpiece sensor scene view.

The sensor model comprises one cylindrical material interaction port representing the observed physical space, which requires the syntactic interface of the workpiece template (see Section 8.1.1). Furthermore, the model contains two data output ports for reporting workpiece presence and the type of the workpiece. Then, one sensor behavior specifies the reactions to workpiece collisions with the material interaction port. The presence output is calculated analogous to the presence output of the generic component sensor template (see Section 8.1.2). The type output is derived from the type output of colliding workpieces instead. If no workpiece is bound to the material interaction port, however, the type output defaults to the empty message $\perp \notin M$. Finally, two cuboid parts model an emitter and a collector.

8.1.4 Abstract static cylinder

Besides the workpiece and sensor models described in the previous sections, the pick and place unit also includes a number of mechanical cylinder models for generating translational kinematic energy. The most basic cylinder model is the abstract static cylinder depicted in Figure 8.5. Note that again the model does not include any physical parts and, thus, neglects collisions within its environment.

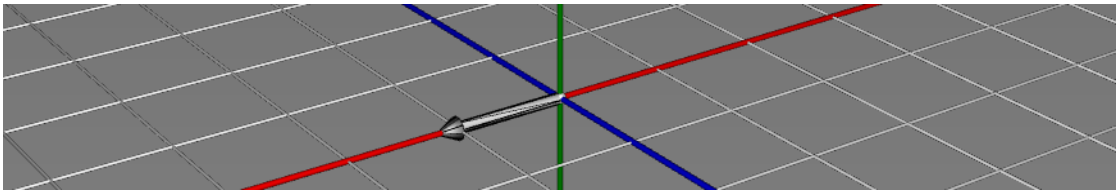


Figure 8.5: Abstract static cylinder scene view.

The model includes one boolean data input port for (de-)activation and one kinematic energy output port with an associated reference transform. Upon activation the behavior generates translation energy along the axis of the reference transform (i.e. the white

arrow in Figure 8.5) for at most five steps until being in extracted state. Upon deactivation the behavior generates opposite translation energy for the same number of steps until being in retracted state again. Hence, the abstract static cylinder model resembles a monostable cylinder, which is stable in retracted state only.

8.1.5 Concrete static cylinder

The next more complex cylinder model is the concrete static cylinder. The model of the concrete static cylinder reuses the model of the abstract static cylinder from the previous section and adds a basic physical representation shown in Figure 8.6. In particular, this model allows one to detect undesirable collision within its environment.

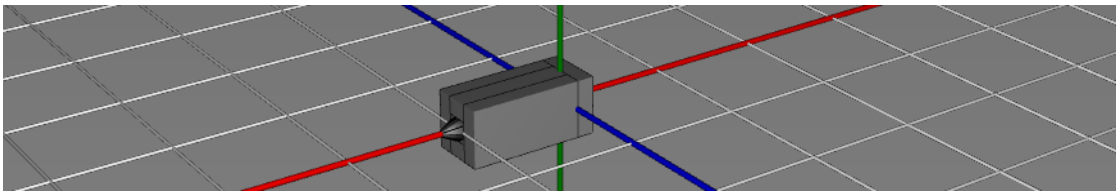


Figure 8.6: Concrete static cylinder scene view.

To reuse the definition of the abstract static cylinder, a decomposition into subcomponents and channels is defined as shown in Figure 8.7. The subcomponents include an instance of the abstract static cylinder template from Section 8.1.4 as well as a *piston*, which is explained in more detail in Section 8.1.6. Note that the instance of the template is depicted using a light gray rectangle with rounded corners and *dashed* outline as opposed to a *solid* outline for the piston component. Furthermore, note that a reuse mechanism such as *inheritance* is not supported in the proposed modeling technique yet, which is why an internal template instance has to be used.

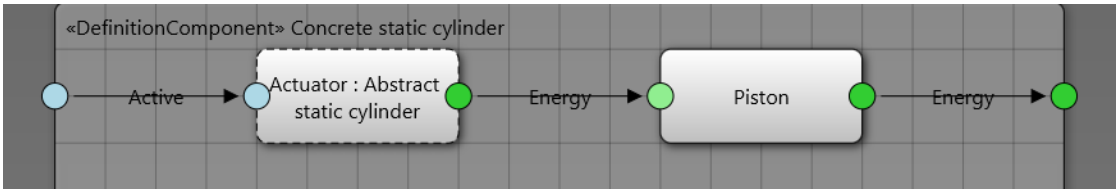


Figure 8.7: Concrete static cylinder components/channels view.

Moreover, the syntactic interface of the concrete static cylinder is identical to the syntactic interface of the abstract static cylinder when neglecting the physical parts (which

are part of the syntactic interface of revised spatio-temporal components according to the underlying theory; see Section 5.2.1). However, internally the kinematic energy generated by the abstract static cylinder is not transmitted to the environment directly. Rather, the kinematic energy is transmitted to the piston first by means of a channel, which forwards the energy to the environment by means of a second channel. Finally, the concrete static cylinder comprises four (mechanical) parts modeling the cuboid case holding and guiding the piston.

8.1.6 Concrete static cylinder / Piston

Subsequently, the piston of the concrete static cylinder from the previous section is responsible for transferring kinematic energy from some input location to some output location. Usually, to implement the energy transfer a solid body is used, where one end serves as the receiver and the other end serves as the emitter. Accordingly, the geometric model of the piston is shown in Figure 8.8.

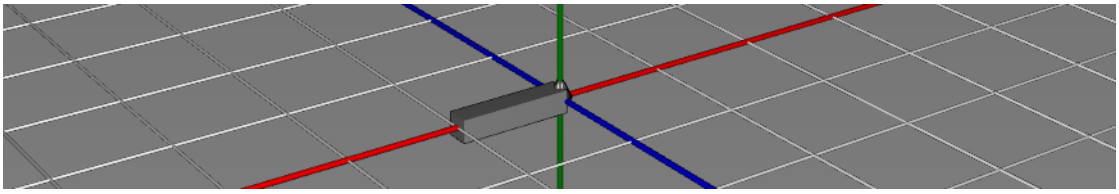


Figure 8.8: Concrete static cylinder piston scene view.

The model of the piston contains one kinematic energy input port and one kinematic energy output port. Hereby, the kinematic energy output port does not require a reference transform. Instead, a behavior specifies the direct and immediate (i.e. non-delayed) transfer from kinematic energy at the input port to kinematic energy at the output port. Finally, one constant part defines the cuboid piston geometry depicted in Figure 8.8, which is guided by the case explained in the previous section.

8.1.7 Concrete dynamic cylinder

Then, the next more complex cylinder model is the concrete dynamic cylinder as shown in Figure 8.9. Compared to the concrete static cylinder the concrete dynamic cylinder supports collision-based interaction by means of a material interaction port at the tip of the piston described in the previous section. In particular, the concrete dynamic cylinder can be used to move generated components $C' \in G(n)$ with computation step $n \in \mathbb{N}$

during system execution. In the given model, generated components C' typically are instances of the workpiece template (see Section 8.1.1).

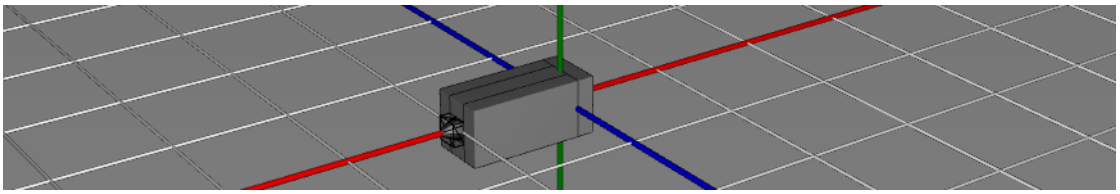


Figure 8.9: Concrete dynamic cylinder scene view.

Similar to the previous cylinder model, the concrete dynamic cylinder model reuses the concrete static cylinder template from Section 8.1.5 and adds a tip component as shown in Figure 8.10. While the concrete static cylinder is responsible for generating the translation energy, the tip is responsible for transmitting the translation energy to colliding components. Note that, again, template instantiation is used to achieve reuse in the proposed model instead of inheritance or other mechanisms.

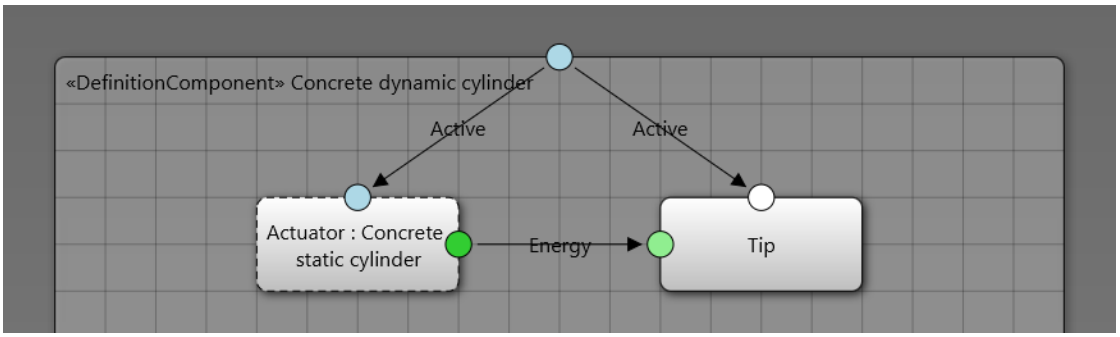


Figure 8.10: Concrete dynamic cylinder components/channels view.

Furthermore, note that the syntactic interface of the concrete dynamic cylinder changes with respect to the syntactic interface of the concrete static cylinder. While a boolean data input port is provided for (de-)activating the cylinder, a kinematic energy output port is omitted. The boolean data input is used for (de-)activating both the internal concrete static cylinder component and the internal tip component. Based on the boolean (de-)activation input, the internal cylinder component generates kinematic energy, which is transfered to the internal tip component by means of a channel. The internal tip component, in turn, is responsible for implementing the interaction with the environment of the concrete dynamic cylinder including generated components $C' \in G(n)$ with computa-

tion step $n \in \mathbb{N}$. Hereby, the tip component also requires the activation input to prevent kinematic energy from being forwarded when retracting the piston. This behavior is explained in more detail in the following section.

8.1.8 Concrete dynamic cylinder / Tip

Finally, the tip of the concrete dynamic cylinder template is responsible for binding colliding components in case of activation (i.e. while extending the piston of the underlying concrete static cylinder; see Sections 8.1.5 and 8.1.6) and, hence, transferring kinematic energy. The activation option has been added to enable energy transfer in one direction of motion of the piston and to disable energy transfer in the opposite direction. The geometric model of the tip is shown in Figure 8.11.

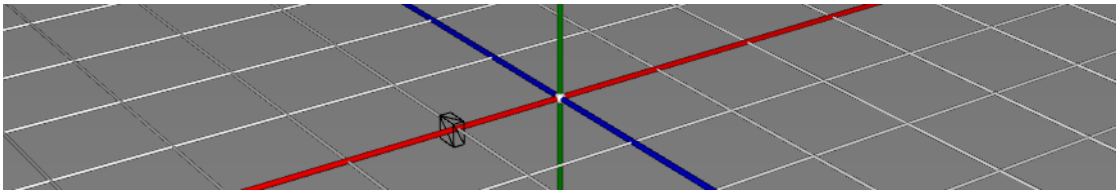


Figure 8.11: Concrete dynamic cylinder tip scene view.

The tip model comprises one generic boolean activation input port $\iota \in I$ and one cuboid material interaction port $\beta \in B$ representing the location of the tip. Hereby, the material interaction port β is deactivated by default and is configured to forward kinematic energy (see Section 5.2.2). During system execution, the internal behavior of the tip component (de-)activates the material interaction port $b(\beta)(n) = (v_n, \emptyset, true)$ at computation step $n \in \mathbb{N}$ with binding history $b \in \vec{B}$ based on the generic activation input $i(\iota)(n)$ with input channel history $i \in \vec{I}$ such that $v_n \neq \emptyset \Leftrightarrow i(\iota)(n)$. Consequently, bindings $C' \in a(\beta)(n)$ with material port activation history $a \in \vec{A}(B \cup Y \cup X)$ only appear upon activation.

8.2 Components

In the following, the (non-reusable) component definitions of the system model are explained. The components include the *pick and place unit* itself (i.e. the actual system under development, see Section 8.2.1) as well as respective subcomponents and template instances. The direct subcomponents of the pick and place unit are the *distributor* (for moving workpieces between different working positions, see Section 8.2.2), the *stamper*

(for stamping workpieces, see Section 8.2.7), and the *sorter* (for sorting workpieces into different exit locations, see Section 8.2.11). Each of the direct subcomponents contains further subcomponents such as distributed, but interacting software controllers as well as specific sensors and actuators forming self-contained mechatronic subsystems. Note that this component architecture has been selected specifically for demonstrating the mechatronic modularization capabilities of the approach. Subsequently, the first two levels of the component hierarchy are described in greater detail including requirements, scenarios, monitors, subcomponents and channels, behaviors, and parts. For the remaining levels of the component hierarchy (i.e. subcomponents of the distributor, the stamper, and the sorter) brief descriptions are provided only similar to the descriptions of the templates in Section 8.1 due to space limitations.

8.2.1 PPU

As mentioned previously, in the performed study the actual system under development is the pick and place unit (PPU) depicted in Figure 8.12. In general, the PPU is responsible for stamping and sorting workpieces, which have been introduced in Section 8.1.1. In fact, the purpose of the PPU is similar to the purpose of waste sorting plants processing different kinds of potentially contaminated containers.

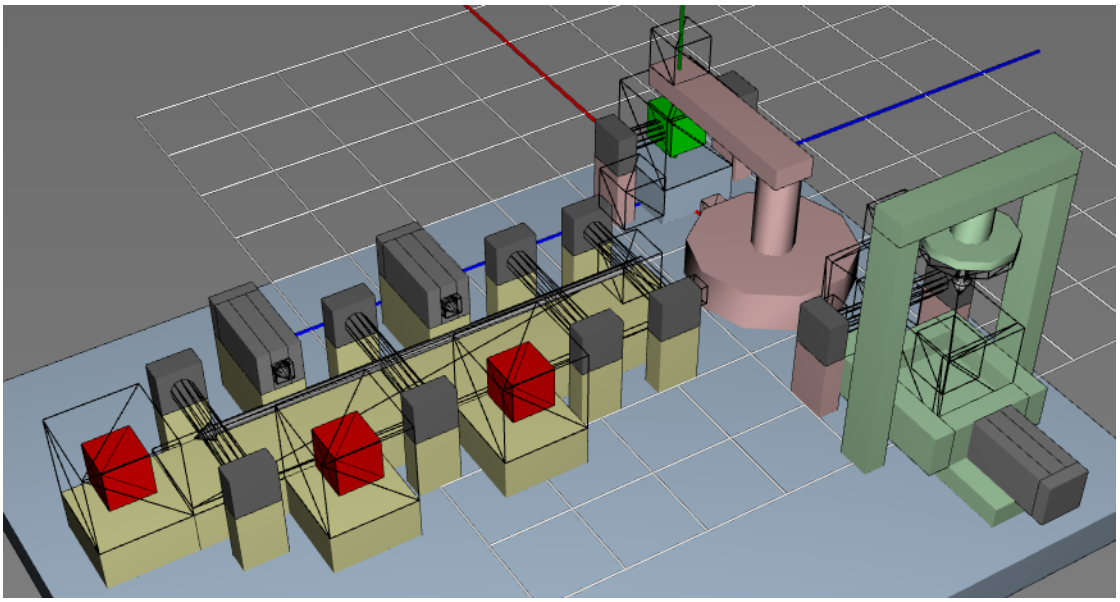


Figure 8.12: PPU scene view.

The model of the pick and place unit encompasses eight informal/natural language requirements. One requirement concerns the types of workpieces (i.e. white, gray, and black) to be processed by the system. Six requirements relate to the activities to be performed with the workpieces instead. The activities include sorting and optionally stamping the workpieces. Hereby, sorting is expressed in terms of moving the workpieces to different exist locations depending on their type. Finally, one requirement holds respective timing constraints regarding the activities.

#	Classification	Description
1	Input/Output	Handle white, gray and black workpieces
2	Activity	Move white workpieces from the entry to the exit one
3	Activity	Move gray workpieces from the entry to the exit two
4	Activity	Move black workpieces from the entry to the exit three
5	Activity	Stamp white workpieces
6	Activity	Do not stamp gray workpieces
7	Activity	Do not stamp black workpieces
8	Property	Move and stamp workpieces within 150 seconds

Subsequently, the syntactic interface of the pick and place unit contains eight ports (five material interaction and three data ports). Four material interaction ports represent the entry and exit locations of workpieces and can be derived from the requirements directly. The fifth material interaction port describes the location, where workpieces are stamped. Not that the stamp location cannot be derived from the requirements, but has to be defined by the engineers during process specification instead. Finally, all three data ports have been added during scenario specification, where the communication protocol with the environment has been concretized. In particular, the pick and place unit provides an initial ready signal, after which start signals can be issued for each workpiece. After processing the respective workpieces, the start signals, in turn, are acknowledged by finished signals.

Source	Name	Type (Parameters)	Description
Requirement	Entry	Material interaction port (Workpiece)	Entry location
Requirement	Exit one	Material interaction port (Workpiece)	Exit location one (white w.)
Requirement	Exit two	Material interaction port (Workpiece)	Exit location two (gray w.)
Requirement	Exit three	Material interaction port (Workpiece)	Exit location three (black w.)
Monitor	Stamp	Material interaction port (Workpiece)	Stamp location
Scenario	Start	Data port (Input, Boolean)	Start signal
Scenario	Ready	Data port (Output, Boolean)	Ready signal
Scenario	Finished	Data port (Output, Boolean)	Finished signal

Then, for each type of workpiece (i.e. white, gray, and black) one scenario has been defined. Hereby, the structure of the three scenarios is identical as shown in Figure 8.13. Each scenario waits for the ready signal to be received from the pick and place unit before generating a workpiece at the entry location, starting workpiece processing, and testing the outcome for compliance with the requirements. In particular, the scenarios verify the exit locations, workpiece states, and respective timing constraints.

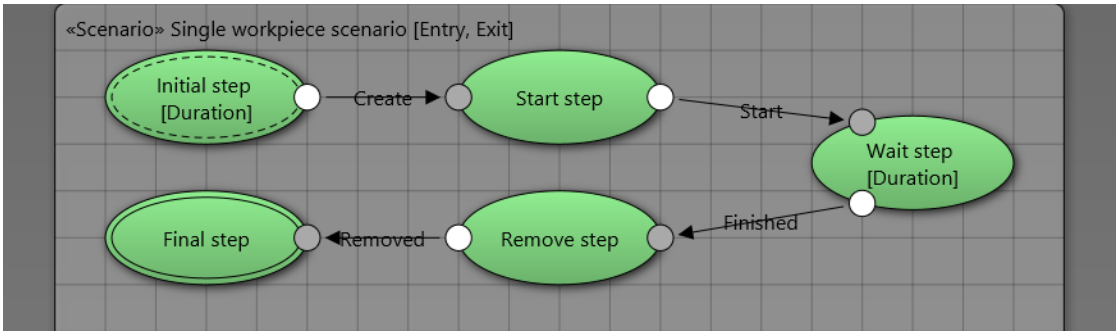


Figure 8.13: PPU single workpiece scenario view.

Technically, each scenario consists of two material life ports *entry* and *exit* (different locations for each workpiece type), five steps *initial step*, *start step*, *wait step*, *remove step*, and *final step* as well as four transitions *create*, *start*, *finished*, and *removed*. The scenario *entry* port corresponds to the PPU *entry* port introduced previously. The scenario *exit* port corresponds to the PPU *exit* port instead. First, the scenario waits for the *ready* signal to be received from the PPU. The waiting time is limited by a *duration* constraint. Then, a workpiece of the specific type is created at the *entry* location. Afterwards, the *start* signal is sent to the PPU. Now the scenario waits for the *finished* signal to be received from the PPU. Again, the waiting time is limited by a *duration* constraint. After receiving the *finished* signal, the scenario tries to remove the workpiece at the *exit* location. If the workpiece can be removed, the scenario terminates successfully. In all other cases the scenario fails.

In addition to the scenarios the pick and place unit comprises a (process) monitor, which is shown in Figure 8.14. The monitor defines the sequences of activities that need to be taken after receiving a start signal from the environment and observing a workpiece at the *entry* port. In particular, the monitor adds the *stamp* location as well as timing constraints for the individual activities, which have not been defined as part of the requirement specification. Note that the *entry* port has been defined as part of the syntactic interface of the pick and place unit before. In the future, one might want

to decouple monitor-specific material interaction ports from the syntactic interface of revised spatio-temporal components. The decoupling, however, requires a substantial revision of the modeling technique (see Chapter 5) and the prototypical tooling (see Chapter 7), which is why it has not been carried out yet.

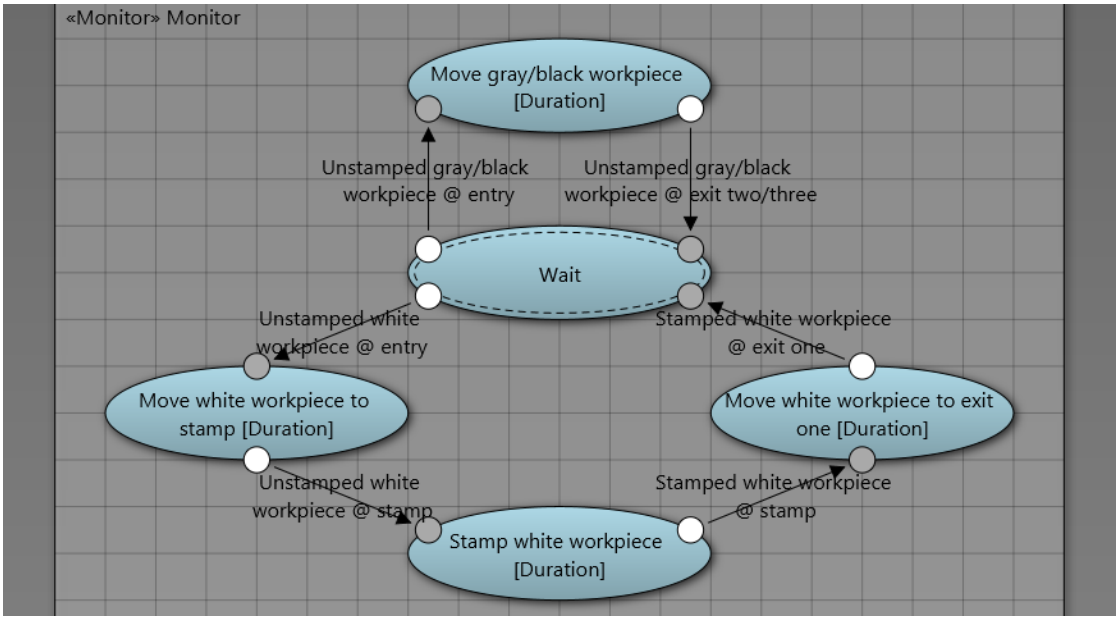


Figure 8.14: PPU monitor view.

Internally, the monitor model includes five activities *wait*, *move white workpiece to stamp*, *stamp white workpiece*, and *move white workpiece to exit one* as well as *move gray/black workpiece*. Furthermore, the specification contains six transitions *unstamped white workpiece @ entry*, *unstamped white workpiece @ stamp*, *stamped white workpiece @ stamp*, *stamped white workpiece @ exit one*, *unstamped gray/black workpiece @ entry*, and *unstamped gray/black workpiece @ exit two/three*. Initially, the monitor waits for an unstamped workpiece to be observed at the *entry* port. Then, depending on the workpiece type (i.e. white, gray, or black) the monitor switches to either the *move white workpiece to stamp* or the *move gray/black workpiece* activity. In both cases the moving time is limited by a *duration* constraint. Subsequently, in case of gray and black workpieces the monitor switches back to the *wait* activity as soon as an unstamped workpiece of the respective type is observed at the *exit two/three* port. Instead, in case of white workpieces the monitor switches to the *stamp white workpiece* activity as soon

as the unstamped workpiece can be observed at the *stamp* location. Again the stamping activity is limited by a *duration* constraint. The stamp activity finishes as soon as a stamped workpiece can be observed at the *stamp* location. Then, the monitor switches to the *move white workpiece to exit one* activity, which is also limited by a *duration* constraint. Finally, the monitor switches back to the *Wait* activity in case stamped white material can be observed at the *exit one* location.

Subsequently, the decomposition of the pick and place unit is shown in Figure 8.15. The pick and place unit comprises three subcomponents, namely a *distributor*, a *stamper*, and a *sorter*. Note that the diagram also shows the material interaction ports of the pick and place unit. However, at this level interactions between the subcomponents and the material interaction ports were not used yet. Rather, the ports are used by the test and process specifications from before as discussed previously.

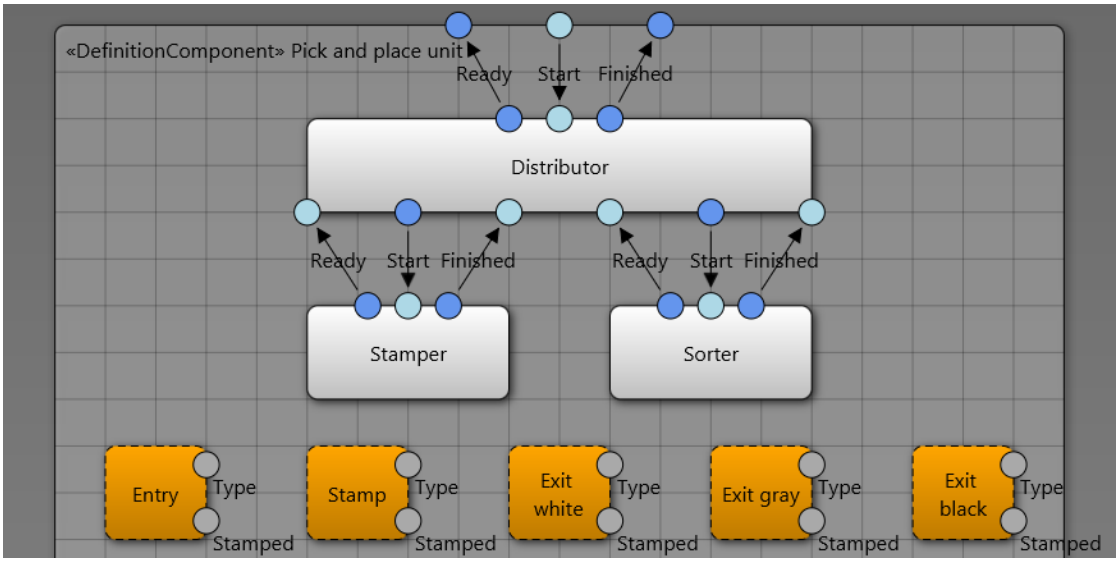


Figure 8.15: PPU components/channels view.

Internally, the distributor component is responsible moving workpieces between and coordinating the other two subcomponents (see Section 8.2.2). In contrast, the stamper component is responsible for stamping workpieces (see Section 8.2.7). Finally, the sorter component is responsible for moving workpieces to their respective exit location (see Section 8.2.11). Note that more detailed descriptions of the three components are provided in the subsequent sections. At last, the pick and place unit component also includes two physical/geometrical parts, namely the *base* and the *stack*. The base defines a large cube

onto which the other components are mounted permanently. The stack defines a small cube underneath the *entry* material ports instead, on which workpieces can be placed initially before starting workpiece processing.

8.2.2 PPU / Distributor

The distributor is the first subcomponent of the pick and place unit (see Section 8.2.1). As the name states, the distributor is responsible for distributing workpieces between the other subcomponents of the pick and place unit, namely the stamper (see Section 8.2.7) and the sorter (see Section 8.2.11). Furthermore, the distributor is responsible for coordinating the the subcomponents as well as communicating with the environment of the pick and place unit. The geometric model of the distributor is shown in Figure 8.16.

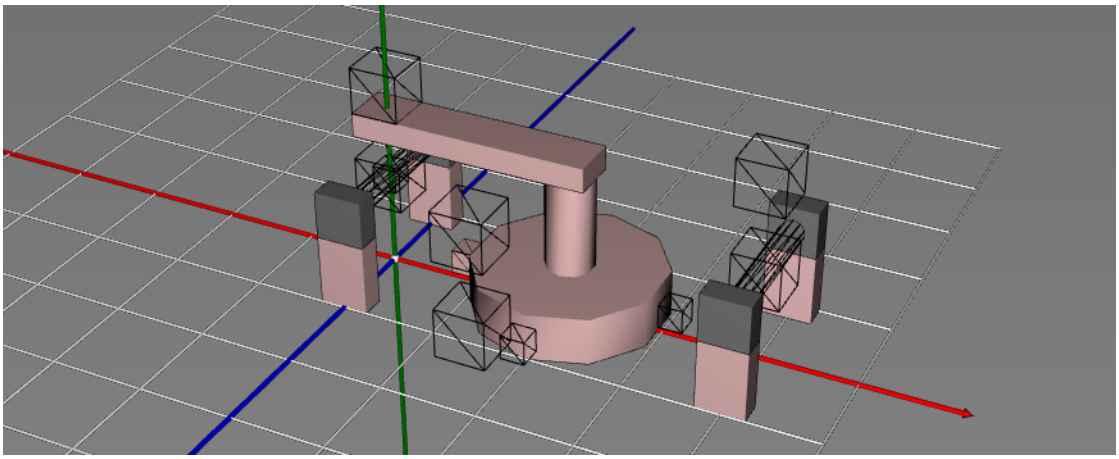


Figure 8.16: PPU distributor scene view.

The distributor model includes six original informal/natural language requirements. Similar to the pick and place unit, the first requirement concerns the types of workpieces (i.e. white, gray, and black) being processed. Then, four requirements describe the activities to be performed with the workpieces. As opposed to the pick and place unit, the distributor component is responsible for moving workpieces between different locations only. Finally, one requirement limits the duration of the activities. Note that the coordination and communication responsibilities are not part of the requirement specification, which has been derived from the original documentation [VHLFF14]. The reason is that the original documentation does not include much information about the software controllers. Furthermore, opposed to the original documentation a distributed control

approach was targeted for demonstrating the capabilities of the proposed approach. Consequently, control responsibilities were assigned to the distributor component, which is not the case in the original documentation.

#	Classification	Description
1	Input/Output	Handle white, gray and black workpieces
2	Activity	Move white workpieces from the entry to the stamper
3	Activity	Move gray workpieces from the entry to the sorter
4	Activity	Move black workpieces from the entry to the sorter
5	Activity	Move white workpieces from the stamper to the sorter
6	Property	Move workpieces within 50 seconds

The syntactic interface of the distributor component contains 15 ports (six material interaction ports and nine data ports). Three material interaction ports describe the locations, where workpieces are expected to be received from and returned to the environment, and can be derived from the requirements directly. The other three material interaction ports describe intermediate workpiece locations, which have been added during (manufacturing) process specification. Finally, the nine data ports are required for implementing the coordination and communication responsibilities and have been introduced during test specification.

Source	Name	Type (Parameters)	Description
Requirement	Stack low	Material interaction port (Workpiece)	Stack location low
Requirement	Stamper low	Material interaction port (Workpiece)	Stamper location low
Requirement	Sorter low	Material interaction port (Workpiece)	Sorter location low
Monitor	Stack high	Material interaction port (Workpiece)	Stack location high
Monitor	Stamper high	Material interaction port (Workpiece)	Stamper location high
Monitor	Sorter high	Material interaction port (Workpiece)	Sorter location high
Scenario	Stamper ready	Data port (Boolean, Input)	Stamper ready signal
Scenario	Sorter ready	Data port (Boolean, Input)	Sorter ready signal
Scenario	Start	Data port (Boolean, Input)	Start signal
Scenario	Stamper finished	Data port (Boolean, Input)	Stamper finished signal
Scenario	Sorter finished	Data port (Boolean, Input)	Sorter finished signal
Scenario	Ready	Data port (Boolean, Output)	Ready signal
Scenario	Start stamper	Data port (Boolean, Output)	Start stamper signal
Scenario	Start sorter	Data port (Boolean, Output)	Start sorter signal
Scenario	Finished	Data port (Boolean, Output)	Finished signal

Then, for each activity-related requirement one scenario is defined as shown in Figure 8.17. The structure of the distributor scenarios is identical to the structure of the scenarios of the pick and place unit (see Section 8.2.1). The reason is that all scenarios

share identical interaction protocols. The interaction protocol expects the system under test (i.e. the pick and place unit or the distributor) to send a ready signal. Then, the environment may place a workpiece at some entry location and start workpiece processing. Finally, the system under test has to return the workpiece at some exit location and send the finished signal. Note that such patterns could be exploited in the future to save modeling effort. However, first the question has to be answered, which patterns appear in practice frequently.

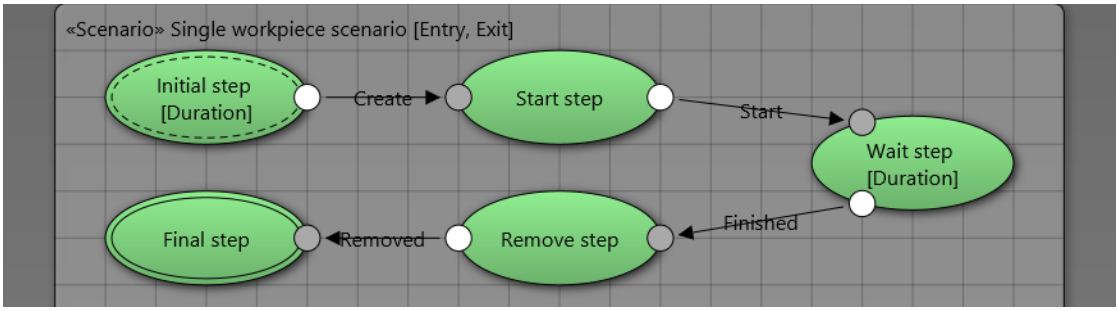


Figure 8.17: PPU distributor scenario view.

Technically, each distributor scenario consists of two material life ports *entry* and *exit*, five steps, and four transitions. The *entry* ports correspond to the stack and the stamper locations, while the *exit* ports correspond to the stamper and the sorter locations. First, the scenario sends the *stamper* and *sorter ready* signal to the distributor. Then, the scenario waits for the distributor to send its own *ready* signal to the environment within a predefined *duration*. Note that the distributor is responsible for communicating with the environment, while the stamper (see Section 8.2.7) and the sorter (see Section 8.2.11) communicate with the distributor. Consequently, the distributor is responsible also for coordinating the stamper and the sorter. Other system architectures might allocate the responsibilities differently. For example, the coordination responsibility could be allocated to a dedicated software component. However, the coordination responsibility matches well with the material distribution responsibility of the distributor component. Then, the scenario creates a workpiece at the *entry* location and sends the *start* signal. Subsequently, the scenario waits for a maximum *duration* until the *finished* signal is received. Finally, the scenario tries to remove the workpiece at the *exit* location.

Then, the model of the distributor includes on process specification, which is depicted in Figure 8.18. The process specification defines three alternative activity sequences, which correlate to the three activity-based requirements/scenarios. Additionally, the process specification introduces three intermediate workpiece locations, namely *stack*,

stamper, and *sorter high*. The intermediate workpiece locations are not part of the requirement specification, but have been introduced by the process engineers instead.

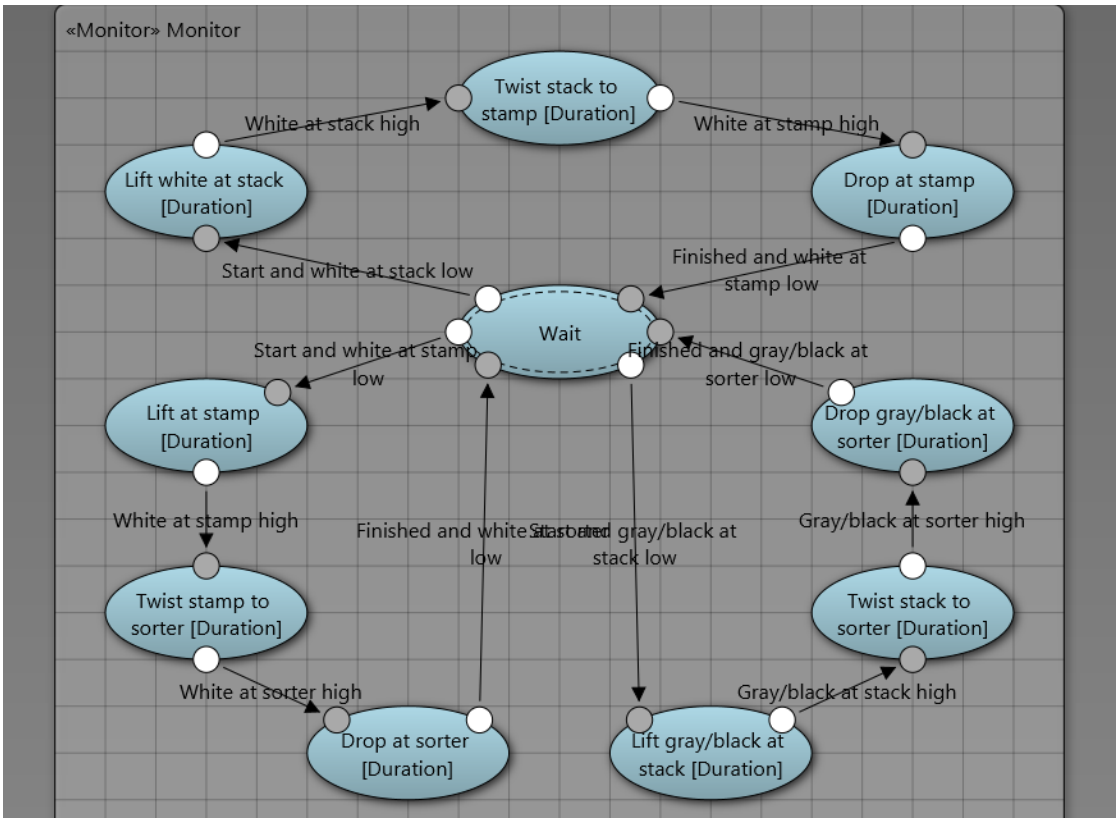


Figure 8.18: PPU distributor monitor view.

Technically, the process monitor contains ten activities and twelve transitions. Initially, the monitor waits for the *start* signal to be received from the environment and a workpiece to be observed at one of the *entry* locations. Then, the monitor expects the workpiece to be lifted from the respective *low* to the respective *high* location within a predefined *duration* (e.g. from *stack low* to *stack high*). Afterwards, the monitor expects the workpiece to be moved/twisted to the target *high* location within a second predefined *duration* (e.g. to *sorter high*). Finally, the monitor requires the workpiece to be dropped at the respective *low* or *exit* location (e.g. to *sorter low*) and the *finished* signal to be sent within a third, and last predefined *duration*. Subsequently, the monitor goes back to the initial activity and waits for the next workpiece to be processed.

To implement the process and test specifications introduced previously, the distributor component employs eight subcomponents depicted in Figure 8.19. The subcomponents include five *sensors* (see the left side of the diagram), one *controller* (see the middle of the diagram) and two *actuators* (see the right side of the diagram). The purpose of each subcomponent is explained in the following.

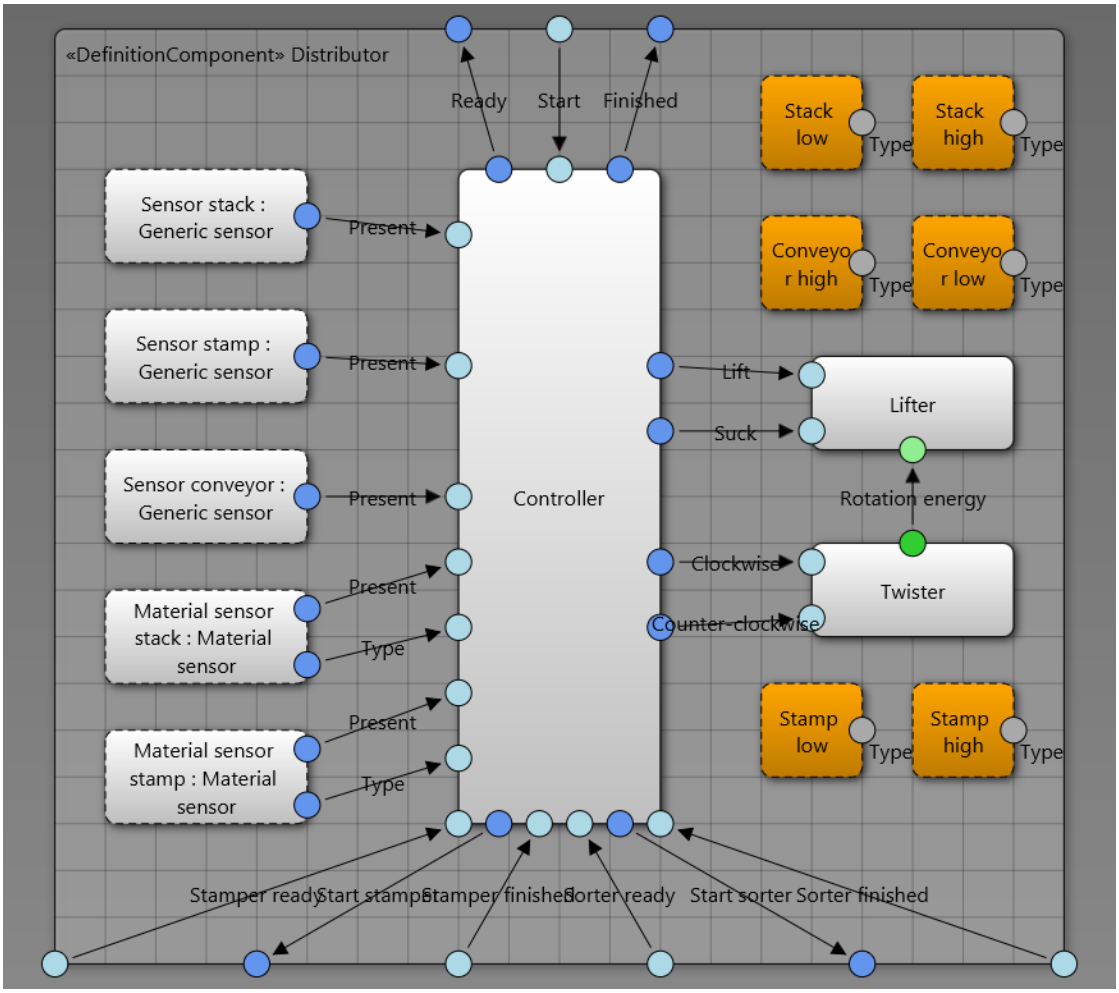


Figure 8.19: PPU distributor components/channels view.

Two instances of the workpiece sensor template (see Section 8.1.3) are used for detecting workpiece presence and the type of the workpiece at each potential *entry* location

(i.e. *stack low* and *stamper low*). Then, a *lifter* component (see Section 8.2.4) is used for lifting and dropping workpieces from and to respective *low* and *high* locations. In contrast, a *twister* component (see Section 8.2.3) is used for rotating the twister component and, therefore, the lifted workpieces around a predefined rotation axis. Note that the channel-based model of kinematic chains is used here (see Section 5.2.1 for more information). Furthermore, three instances of the component sensor template (see Section 8.1.2) are used to detect the rotational position of the lifter at discrete locations. Finally, the controller component (see Section 8.2.6) is used for communicating with the environment and coordinating the actuator movements. At last, the distributor component also includes parts, namely one *base* as well as different *mounts* for the emitters and collectors of the workpiece sensors, which can be seen also in Figure 8.16. The base comprises a cylinder volume, which is aligned with the rotation axis of the twister. Instead, the mounts contain box volumes underpinning the collector and emitter parts of the respective workpiece sensors.

8.2.3 PPU / Distributor / Twister

The twister component is used inside the distributor component from Section 8.2.2 for rotating the lifter component from Section 8.2.4 around a predefined axis, which is shown in Figure 8.20. A very basic model of the twister is provided neglecting physical parts and, hence, potential collisions within its environment. To implement the twister behavior, for example, electric drives can be used.

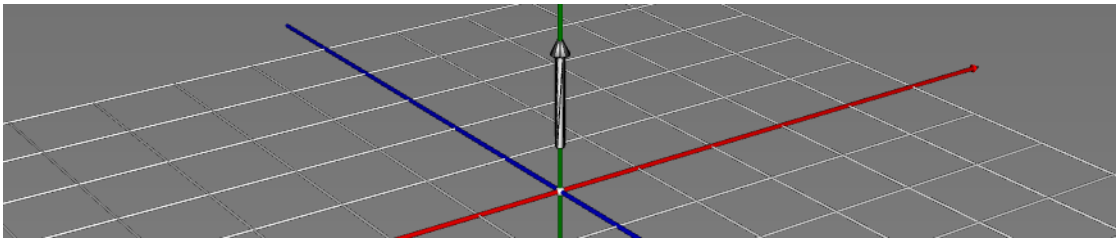


Figure 8.20: PPU distributor twister scene view.

The model of the twister component includes two boolean data input ports, one for clockwise rotation, the other for counter-clockwise rotation, and one kinematic energy output port with a rotational reference transform. Furthermore, one behavior is defined producing (counter-)clockwise rotation energy depending on the input assignments. Note that two data input ports are provided for security reasons. Consequently, only one direction must be activated at a time for kinematic energy to be produced by the actuator.

If none of the directions is activated or both are activated, the actuator is turned off. This behavior coincides with commercial actuator equipment.

8.2.4 PPU / Distributor / Lifter

Then, the lifter component is employed inside the distributor component from Section 8.2.2 for lifting and dropping instances of the workpiece template from Section 8.1.1. The geometric model of the lifter component is shown in Figure 8.21. The model represents a basic crane with gripper. The lifter can be implemented, for example, with pneumatic components for both lifting and gripping.

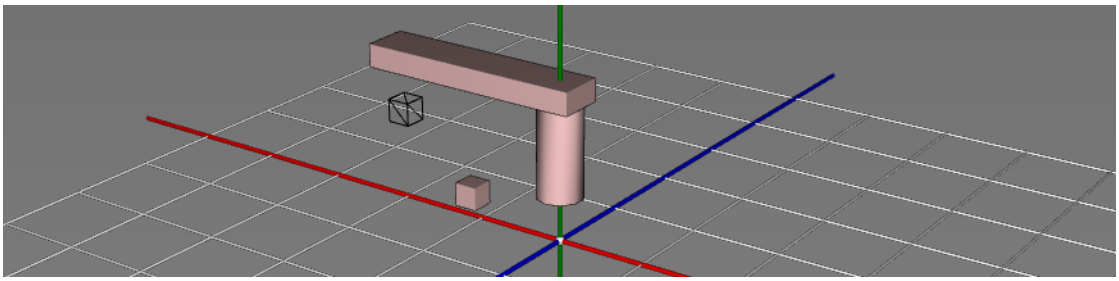


Figure 8.21: PPU distributor lifter scene view.

For implementation of the lifter component two subcomponents are used as shown in Figure 8.22, namely an instance of the abstract static cylinder template from Section 8.1.4 and an *arm* component described in more detail in Section 8.2.5. The arm component is responsible for gripping workpieces, while the (mechanical) cylinder component is responsible for lifting the arm component and, hence, gripped workpieces. Note that the physical shape of the cylinder is omitted in the model for simplicity. Furthermore, note that the cylinder is aligned with the y-axis and covered completely by the arm geometry, which is why the rotation axis of the reference transform the the cylinder's kinematic energy output port is not visible in Figures 8.21 and 8.23.

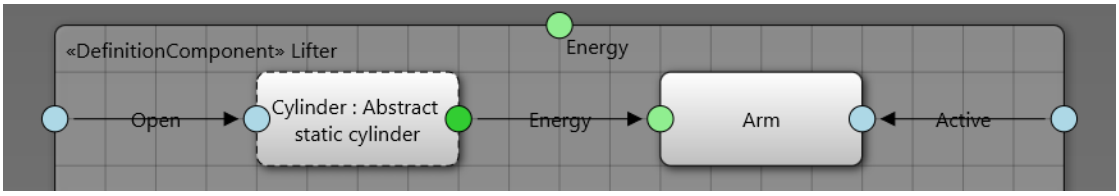


Figure 8.22: PPU distributor lifter components/channels view.

During execution the boolean data input of the lifter component is forwarded directly to the cylinder for controlling its kinematic energy output. Then, the kinematic energy output is forwarded to the arm component for changing its elevation along the y-axis. Again, note that the channel-based model of kinematic chains is used here (see Section 5.2.1 for more information). Moreover, a second boolean data input of the lifter is forwarded directly to the arm component for controlling its gripper state. The gripper state can be active and, thus, binding colliding workpieces or inactive and, hence, non-binding instead. Note that internally the binding mechanism is realized by means of a kinematic energy forwarding material interaction port not visible in the components/channels view from Figure 8.22. At last, the lifter component includes one physical part, namely a small cuboid *nub*. Similar to the cylinder and the arm components, the nub is rotated by the twister component from Section 8.2.3. Consequently, the nub collides with the instances of the (generic) component sensor template (see Section 8.1.2) of the distributor component (see Section 8.2.2). These collisions allow one to derive the rotational position of the lifter component at discrete locations.

8.2.5 PPU / Distributor / Lifter / Arm

Subsequently, the arm component is used inside the lifter component (see Section 8.2.4) of the distributor component (see Section 8.2.2) for gripping instances of the workpiece template (see Section 8.1.1). The geometric model of the arm is shown in Figure 8.23. Note that compared to the lifter component only the nub part is missing, which must not be elevated because otherwise the rotational position cannot be determined anymore.

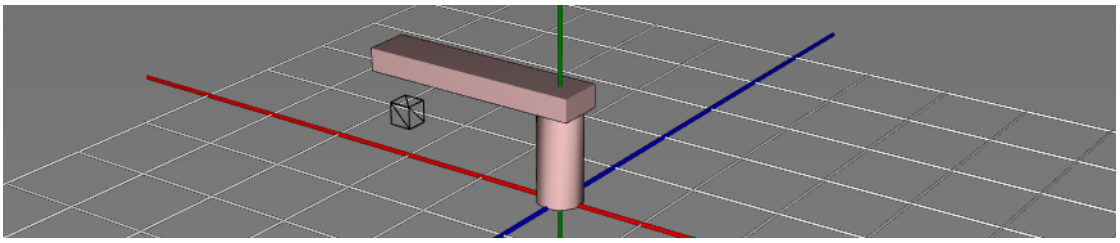


Figure 8.23: PPU distributor lifter arm scene view.

Internally, the arm component includes one boolean data input port and one kinematic energy forwarding material interaction port representing the gripper location. The gripper location corresponds to the small black wire frame box volume shown in Figure 8.23. Furthermore, one behavior is defined (de-)activating the material interaction port based on the boolean data input similar to the tip component of the concrete dynamic cylinder

template (see Section 8.1.8). Consequently, in case the material interaction port is active, colliding instances of the workpiece template are bound to the port and kinematic energies are forwarded, which are applied to any of the port's parent components (i.e. the arm component, the lifter component, the distributor component, or the pick and place unit itself). Note that at this point also defective gripper behavior can be modeled to design the system for robustness against component failure [RHZ14a]. Finally, the arm model also defines two physical parts, namely a *cylinder* and a *box*. The cylinder is aligned with the rotation axis of the twister component, while the box connects the tip of the cylinder to the material interaction port introduced previously. A more detailed model of the actual gripper is omitted here.

8.2.6 PPU / Distributor / Controller

Finally, the distributor component from Section 8.2.2 also includes a controller component for coordinating the twister component (see Section 8.2.3) and the lifter component (see Section 8.2.4) based on sensor measurements. The controller model contains data input ports for each sensor measurement (e.g. presence and type of workpiece at stack and stamp locations) and data output ports for each actuator control value (i.e. cylinder or gripper activation). In contrast, the controller dynamics is specified by means of a behavior depicted in Figure 8.24.

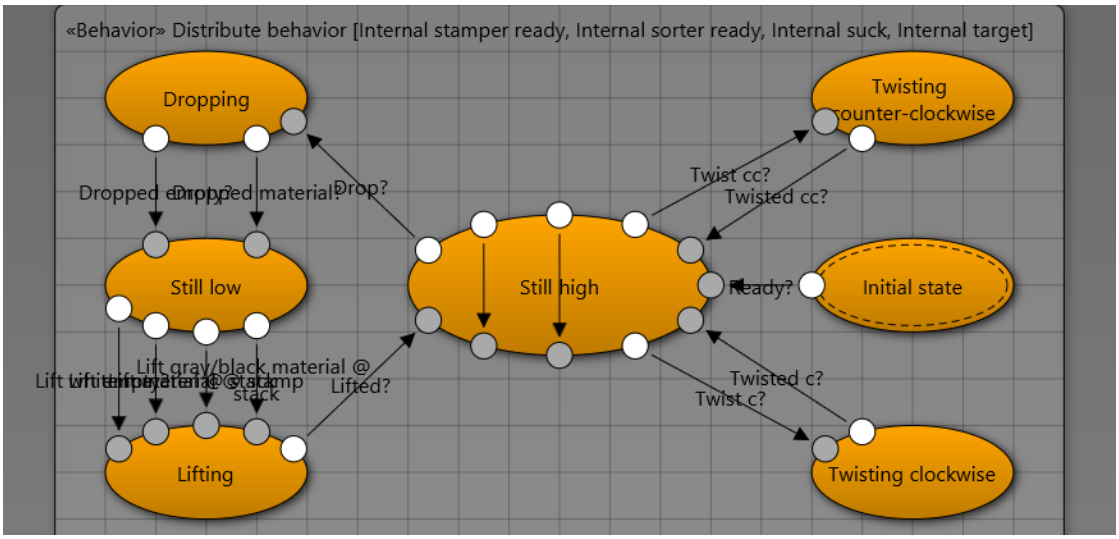


Figure 8.24: PPU distributor controller behavior.

The behavior consists of four variables, seven states and 15 transitions. Initially, the controller raises the lifter component and waits for the stamper component (see Section 8.2.7) and the sorter component (see Section 8.2.11) to send the ready signal before informing the environment about its own *ready* state and switching to the *still high* state. In *still high* state the controller waits for workpieces to be detected at the *entry* locations and the *start* signal to be received from the environment. Then, the controller updates internal variables to remember the workpiece type and the target location. Afterwards, the controller switches to the *twisting clockwise* or the *twisting counter-clockwise* state to move the lifter component to the location, where the workpiece has been detected. Being at the correct location, the controller switches to the *dropping* state to lower the lifter component. Then, in *still low* state the controller activates the gripper, determines the new target location, and starts raising the lifter component again. In *still high* state, the controller decides in which direction to twist the lifter component to. Then, *dropping* is triggered a second time until in *still low* state is reached. Now, the controller decides to deactivate the gripper and raise the lifter component again. Finally, the controller waits in *still high* state for another workpiece to be detected and a *start* signal to be received.

8.2.7 PPU / Stamper

The stamper component is the second subcomponent of the pick and place unit as introduced in Section 8.2.1. The stamper is responsible for stamping (i.e. transferring stamp energy to) instances of the workpiece templates (see Section 8.1.1). The geometric model of the stamper component is shown in Figure 8.25.

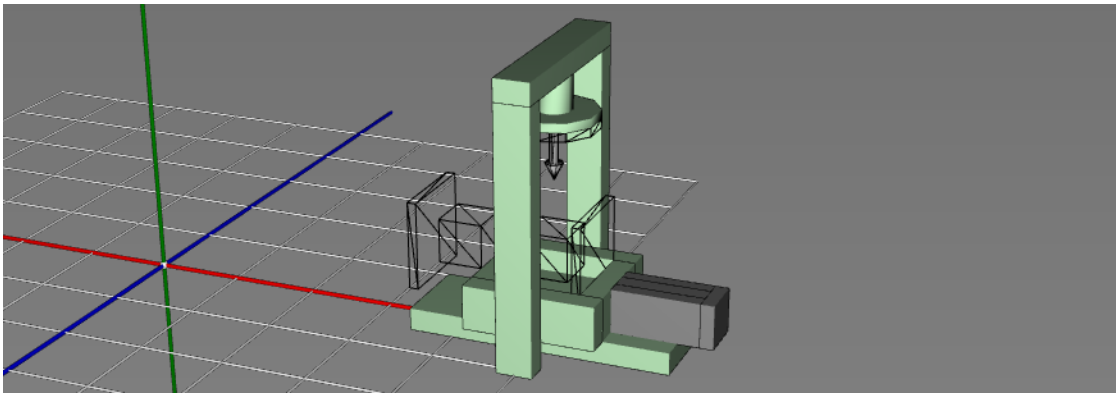


Figure 8.25: PPU stamper scene view.

Initially, five requirements have been defined for the stamper module one regarding the types of workpieces being processed (i.e. white, gray, and black), three regarding the processes (or activities) to be performed with the workpieces, and one regarding the time limits of the processes respectively activities. Note that, in contrast to the distributor component (see Section 8.2.2), the main purpose of the stamper component is stamping workpieces rather than moving workpieces between different locations. However, the requirement specification defines only the entry and exit locations of workpieces, while the stamp location itself is left undecided.

#	Classification	Description
1	Input/Output	Handle white workpieces
2	Activity	Receive workpieces at the entry location
3	Activity	Stamp workpieces
4	Activity	Provide workpieces at the exit location
5	Property	Handle workpieces within 50 seconds

Then, the model of the stamper comprises five ports, two material interaction ports and three data ports. One material interaction port represents can be derived from the requirement directly, namely the *entry/exit* location (which is identical in case of the stamper component). The second material interaction port is required for manufacturing process (or monitor) specification, namely the location where the instances of the workpiece template are *stamped*. Finally, the data ports are added during scenario specification to specify signal-based communication as part of the interaction protocol, which has been described previously. In particular, the interaction protocol requires the stamper to send a *ready* signal, before workpieces can be processed. Furthermore, a *start* signal is required for triggering workpiece processing. Finally, a *finished* signal has to indicate the end of workpiece processing.

Source	Name	Type (Parameters)	Description
Requirement	Entry/exit	Material interaction port (Workpiece)	Entry location
Monitor	Stamp	Material interaction port (Workpiece)	Stamp location
Scenario	Start	Data port (Boolean, Input)	Start signal
Scenario	Ready	Data port (Boolean, Output)	Ready signal
Scenario	Finished	Data port (Boolean, Output)	Finished signal

From the requirements one scenario was derived for white workpieces (see Section 8.1.1) only. The scenario is structured identically to scenarios of the distributor component (see Section 8.2.2) and the pick and place unit (see Section 8.2.1). Finally, a screenshot of the stamper scenario is shown in Figure 8.26.

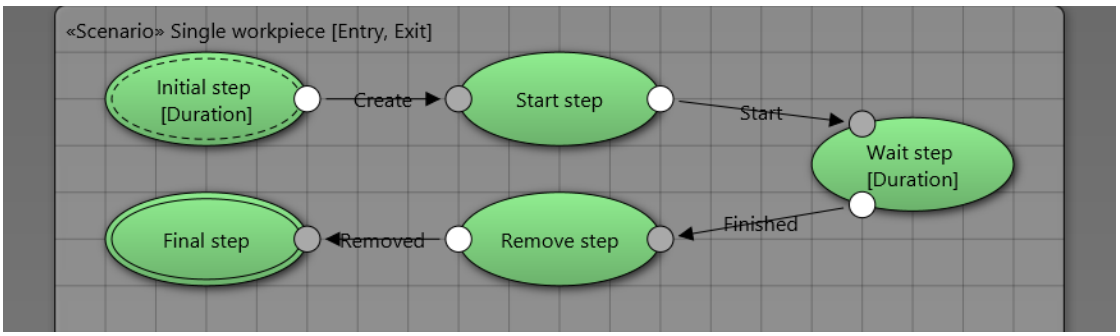


Figure 8.26: PPU stamper scenario view.

In the *initial step* the scenario waits for the stamper component to send the *ready* signal within a predefined maximum *duration*. Then, the scenario creates a white workpiece at the *entry/exit* location and switches to the *start step*. Next, the scenario sends the *start* signal to the stamper component and goes to the *wait step*. In the *wait step* the scenario waits for a maximum *duration* until the stamper component sends the *finished* signal. Thereafter, in the *remove step* the scenario tries to remove the stamped workpiece at the *entry/exit* location. If the removal succeeds, the scenario reaches the *final step* and terminates successfully. Otherwise, the scenario fails and produces a semantic issue.

From the scenarios one process specification (i.e. a monitor) was derived, which determines the intermediate process steps as depicted in Figure 8.27. In particular, the monitor adds the actual *stamp* location to the *entry/exit* location. Also, the monitor defines tighter time frames for moving and stamping activities. Both the *dtamp* location and the time frames are not defined in the requirement specification and, thus, represent design decision taken by the process engineers.

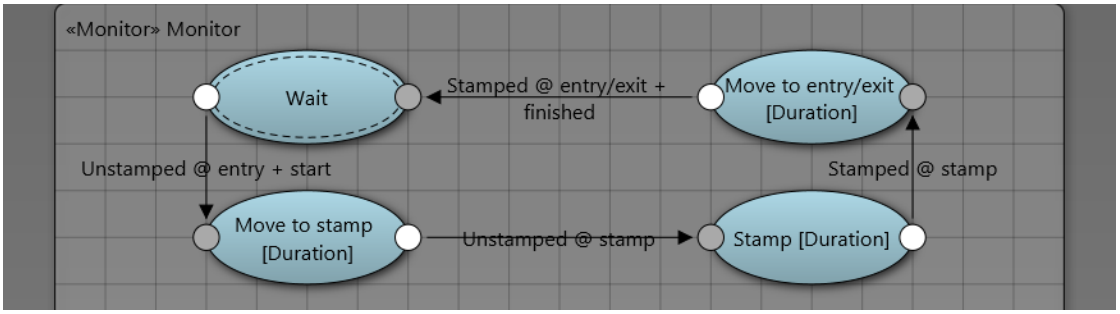


Figure 8.27: PPU stamper monitor view.

Initially, the monitor waits for an unstamped workpiece to be observed at the *entry/exit* location and the *start* signal to be received from the environment. Then, the monitor switches to the *move to stamp* activity for a maximum *duration* until the unstamped workpiece can be observed ad the *stamp* location. Afterwards, the monitor resides in the *stamp* activity for second maximum *duration* until a stamped workpiece is present at the *stamp* location. Afterwards, the monitor changes to the *move to entry/exit* activity for a third, and last, maximum *duration* until a stamped workpiece can be observed at the *entry/exit* location and the stamper sends the *finished* signal. Finally, the monitor goes back into the *wait* activity.

For implementing the scenario and process specifications five subcomponents were employed as shown in Figure 8.28. The subcomponents include one instance of the concrete static cylinder template (see Section 8.1.5), one instance of the abstract static cylinder template (see Section 8.1.4), a *basket* component (see Section 8.2.8), a *stamp* component (see Section 8.2.9), and a *controller* component (see Section 8.2.10).

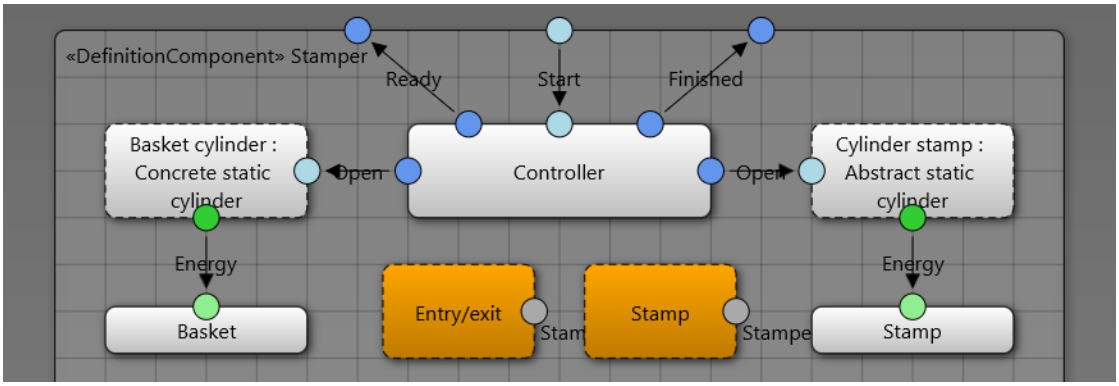


Figure 8.28: PPU stamper components/channels view.

The controller component is responsible for communicating with the environment and coordinating the other subcomponents. In particular, the controller receives the *start* signal from the environment, controls the cylinder components, and provides the *ready* and *finished* signals back to the environment. In contrast, the cylinders represent the actuators of the subsystem. More specifically, the cylinder components transfer their translation energy to the basket and stamp components respectively. The basket component is responsible for transporting workpieces from the *entry/exit* to the *stamp* location. Therefore, the basket component binds workpieces and moves in horizontal direction between the *entry/exit* and *stamp* locations. Furthermore, the stamp component is responsible for transferring the actual stamp energy to the workpieces. Therefore, the

stamp moves in vertical direction between its initial position and the *stamp* location, where it collides with the workpieces and transmits the energy. Finally, the stamper model also includes four parts, namely a *base* and three *stamp mounts* depicted in Figure 8.25. The base is represented by a flat horizontal box volume, while the stamp mounts are defined by two elongated vertical box volumes at the sides and one elongated horizontal box volume at the top forming a frame.

8.2.8 PPU / Stamper / Basket

As mentioned previously, the basket component of the stamper component from Section 8.2.7 is responsible for carrying instances of the workpiece template (see Section 8.1.1) from the *entry/exit* location of the stamper component to the *stamp* location of the stamper component and back. The geometric model of the basket is provided in Figure 8.29 including basic physical parts and one kinematic energy forwarding material interaction port.

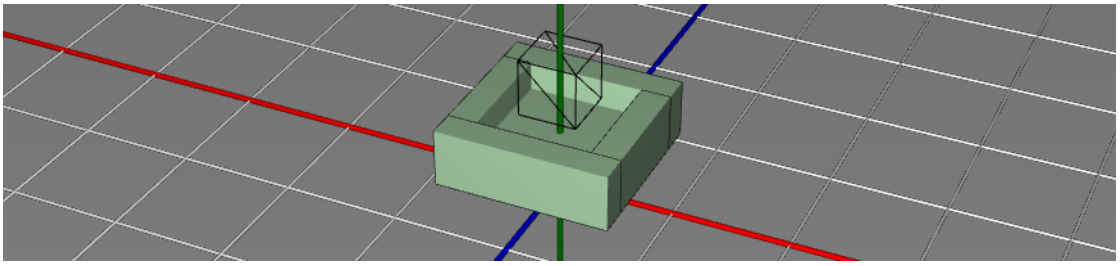


Figure 8.29: PPU stamper basket scene view.

The physical parts of the basket component provide the frame where instances of the workpiece template need to be placed. In particular, the parts allow one to test during simulation whether the workpieces are placed at the correct location or not. Slight misplacements are reported typically using part collisions (see Section 6.2.2). In contrast, the material interaction port is configured to be always active, binds any type of colliding component, and forwards kinematic energy (see Section 5.2.1).

8.2.9 PPU / Stamper / Stamp

Then, the stamp component of the stamper component from Section 8.2.7 is responsible for transferring stamp energy to colliding instances of the workpiece template (see Section 8.1.1). Note that the name “stamper” is supposed to represent a more functional unit, while the name “stamp” ought to represent a more physical unit. The model of

the stamp component includes basic physical parts and one material interaction port for binding colliding workpieces dynamically as shown in Figure 8.30.

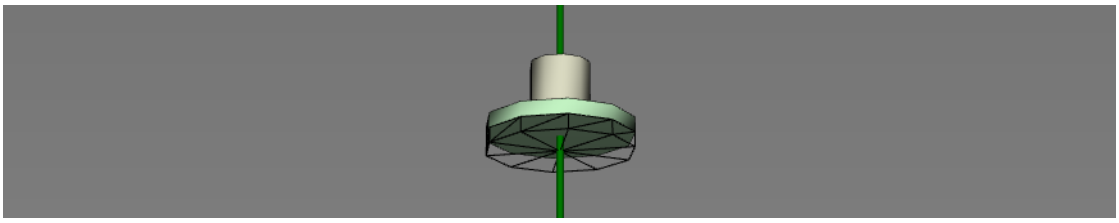


Figure 8.30: PPU stamper stamp scene view.

Internally, the stamp component comprises one behavior producing a constant stamp energy output at the respective generic energy input port of the material interaction port. Consequently, stamp energy can be transferred to colliding instances of the workpiece template. The material interaction port itself is always active in the given model, but could be activated when moving downwards only to be more realistic. Note that input ports of material interaction ports can be written directly by behaviors for simplicity. Similarly, output ports of material interaction ports can be read by behaviors directly. This feature is not supported explicitly by the underlying theory (see Section 5.2.5).

8.2.10 PPU / Stamper / Controller

Finally, the controller of the stamper component from Section 8.2.7 is responsible for (de-)activating the cylinder components, which move the basket component (see Section 8.2.8) and the stamp component (see Section 8.2.9) towards each other. The behavior of the controller component is depicted in Figure 8.31.

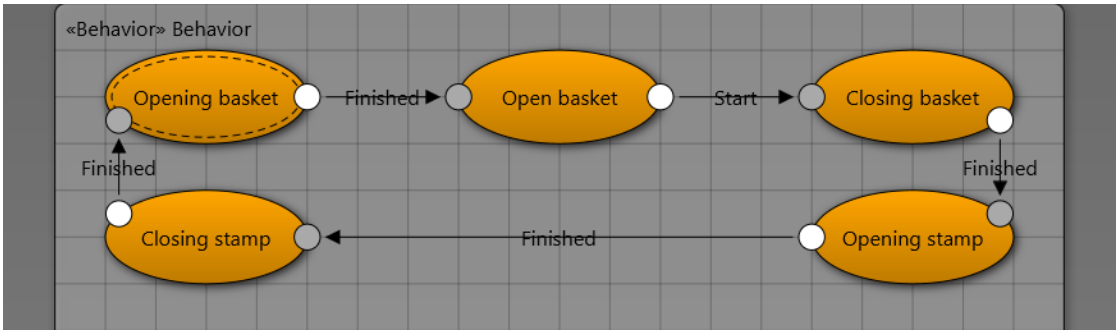


Figure 8.31: PPU stamper controller behavior view.

In *opening basket* state the controller activates the basket cylinder such that the basket moves from the *stamp* location to the *entry/exit* location (note that the basket component resides at the *stamp* location initially due to deactivation of the monostable cylinder component; see Section 8.1.4). Then, after a predefined number of time steps the controller switches to the *open basket* state. Note that sensors were not used to detect the position of the basket component (see Section 8.2.8) for simplicity. Consequently, the given system design might be susceptible to malfunctions of the monostable cylinder. However, such malfunctions can be neglected during conceptual design to reduce the complexity of the design problem and, hence, the time until verification and validation. Note, however, that a methodology has been devised for introducing malfunctions in later stages [RHZ14a] (which includes extending the behavior specifications with malfunctioning states, adding scenarios for testing the response to malfunctioning states, and extending the implementation to handle malfunctioning states appropriately). Then, in the *open basket* state the controller waits for the *start* signal to be received from the environment before switching in the *closing basket* state. In this state the (monostable) basket cylinder component is deactivated such that the basket component moves from the *entry/exit* location back to the *stamp* location. Again, after a predefined duration the controller goes into the *opening stamp* state assuming that the basket cylinder component was retracted successfully and the basket component resides at the *stamp* location. Note that, here, sensors also could have been used to improve the robustness of the system with respect to malfunctions. In the *opening stamp* state the (monostable) stamp cylinder component is activated such that the stamp component (see Section 8.2.9) moves from its initial location to the *stamp* location. Note that the motion of the stamp component causes the material interaction port of the stamp component to collide with the workpiece, which, in turn, causes generic stamp energy to be transferred from the stamp component to the workpiece. Then, after a predefined duration the controller switches to the *closing stamp* state. In this state the stamp cylinder component is deactivated such that the stamp component moves back to its initial position. After a predefined duration the controller switches back to the *opening basket* state and reiterates through the described procedure.

8.2.11 PPU / Sorter

The sorter component is the last subcomponent of the pick and place unit as introduced in Section 8.2.1. The sorter component is responsible for moving instances of the workpiece template (see Section 8.1.1) from the *entry* location of the subcomponent to their respective *exit* locations, which depends on the type of the workpieces. The geometric model of the sorter component is provided in Figure 8.32.

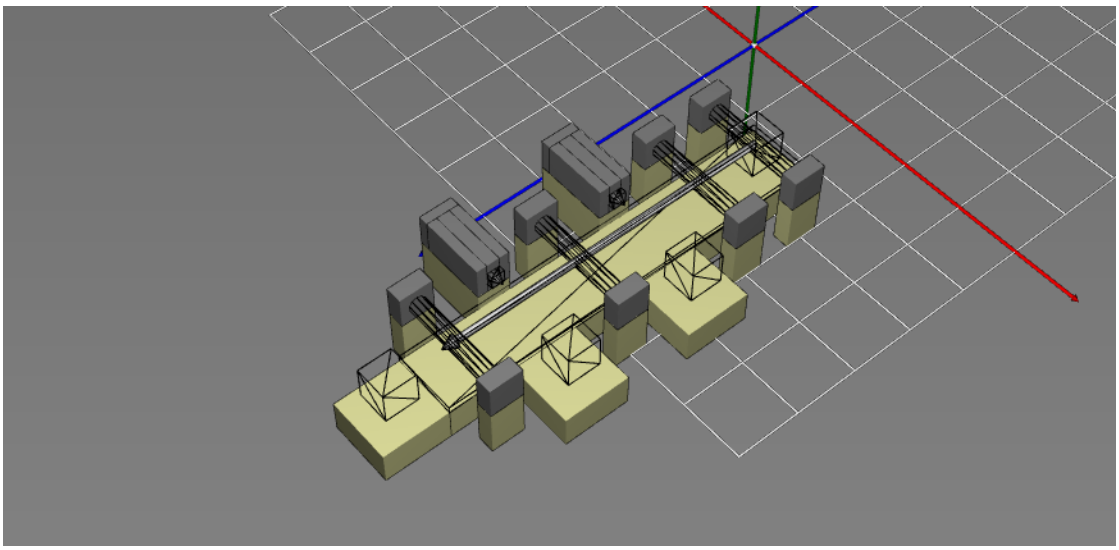


Figure 8.32: PPU sorter scene view.

The sorter component comprises six requirements, one regarding the types of workpieces being processed (i.e. white, gray, and black), four regarding the processes and activities to be performed (i.e. moving workpieces), and one regarding constraints for the respective processes and activities (i.e. time limits). Note that, as opposed to the stamper component (see Section 8.2.7), the main purpose of the sorter component is moving potentially stamped workpieces instead of stamping workpieces itself.

#	Classification	Description
1	Input/Output	Handle white, gray and black workpieces
2	Activity	Receive workpieces at the entry location
3	Activity	Move white workpieces to the exit one location
4	Activity	Move gray workpieces to the exit two location
5	Activity	Move black workpieces to the exit three location
6	Property	Handle workpieces within 50 seconds

Then, the sorter component contains five material interaction ports and three data ports. Four material interaction ports represent the entry and exit location of workpieces, which can be derived from the requirements directly (i.e. *entry* and *exit one/two/three*). The other material interaction port has been added during behavior specification (i.e. the *belt surface*) to transfer kinematic energy to workpieces being process by the sorter

component. Finally, the three data ports have been added during scenario specification. Again, the data ports are required to implement the interaction protocol introduced previously (see Sections 8.2.1, 8.2.2, and 8.2.7). Remember that the interaction protocol comprises a *ready* signal, a *start* signal, and a *finished* signal.

Source	Name	Type (Parameters)	Description
Requirement	Entry	Material interaction port (Workpiece)	Entry location
Requirement	Exit one	Material interaction port (Workpiece)	Exit one location
Requirement	Exit two	Material interaction port (Workpiece)	Exit two location
Requirement	Exit three	Material interaction port (Workpiece)	Exit three location
Behavior	Belt surface	Material interaction port (Workpiece)	Belt surface location
Scenario	Start	Data port (Boolean, Input)	Start signal
Scenario	Ready	Data port (Boolean, Output)	Ready signal
Scenario	Finished	Data port (Boolean, Output)	Finished signal

Subsequently, the model of the sorter component comprises three structurally identical scenarios (one for each type of workpiece, i.e. white, gray, and black) as depicted in Figure 8.33. Similar to the pick and place unit (see Section 8.2.1) the scenarios differ only in the types of workpieces created and the *exit* locations, where workpieces are expected to be returned. Again, note the recurring specification pattern here, which could be exploited in future version of the approach.

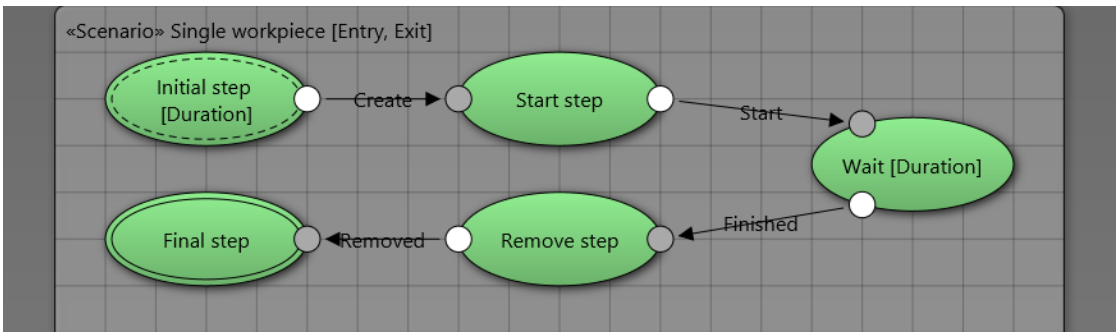


Figure 8.33: PPU sorter scenario view.

In the *initial step* the scenario waits for the sorter component to send the *ready* signal for a maximum *duration*. After receiving the *ready* signal the scenario creates a workpiece of the respective type at the *entry* location and switches to the *start step*. Then, the scenario sends the *start* signal to the sorter component and goes into the *wait step* for a second maximum *duration*. After receiving the *finished* signal from the sorter the scenario changes to the *remove step*. Thereafter, if a workpiece can be removed at the

expected *exit* location the scenario reaches the *final step* and terminates successfully. Otherwise, the scenario fails with a semantic issue (see Section 6.2).

This time, the monitor specification is omitted for simplicity. Note that, similar to the previous components, the monitor specification could add intermediate workpiece locations as well as additional timing constraints. Then, to implement the scenario specifications eight subcomponents are used as shown in Figure 8.34. The subcomponents include four instances of the workpiece sensor template (see Section 8.1.3), two instances of the concrete dynamic cylinder template (see Section 8.1.7), one *actuator* component, which is described in more detail in Section 8.2.12, and one *controller* component, which is explained in Section 8.2.13 instead.

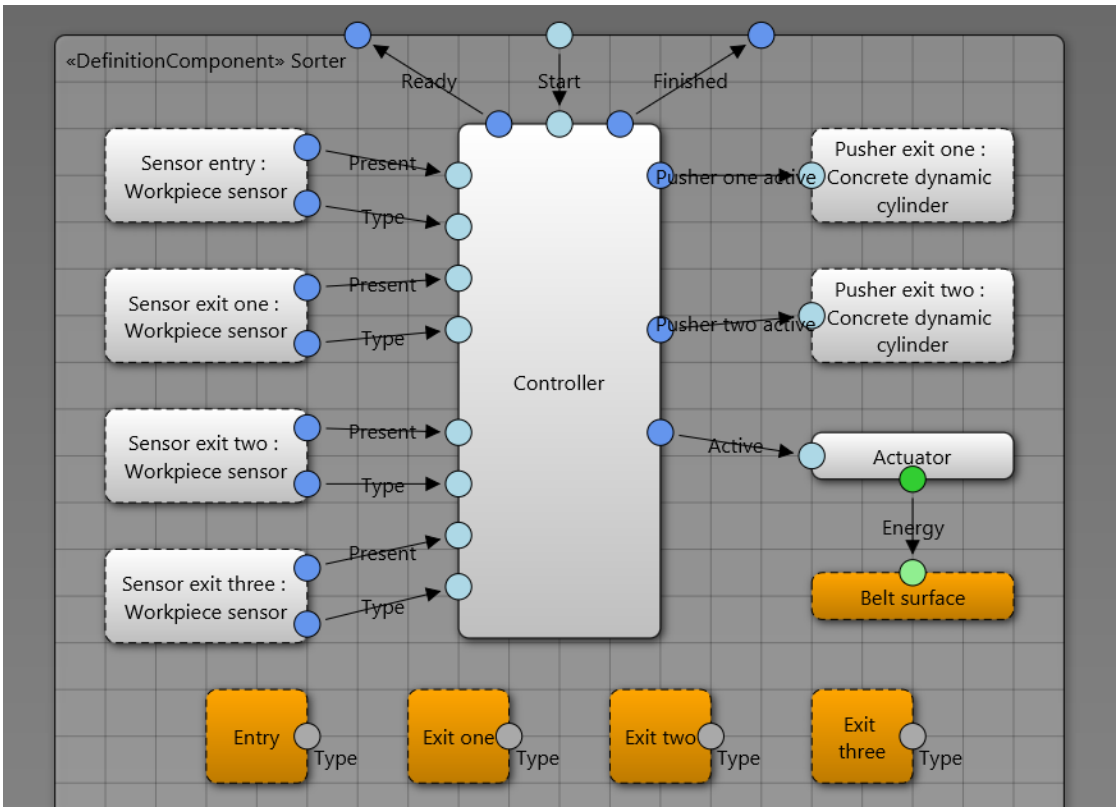


Figure 8.34: PPU sorter components/channels view.

Again, the controller component is responsible for communicating with the environment and coordinating the other subcomponent. In particular, the controller component

receives the *start* signal from the environment and provides the *ready* and *finished* signals back to the environment. Furthermore, the controller component uses the inputs from the workpiece sensor template instances to determine the location and the type of workpieces at discrete locations along the *belt surface*. Finally, the controller component coordinates the instances of the concrete dynamic cylinder template and the actuator component such that the workpieces are moved to the correct *exit* location. Note that, as opposed to the previous components, this time dynamic cylinders are used because workpieces are generated components $C' \in G(n)$ with computation step $n \in \mathbb{N}$. Finally, the sorter also includes 14 physical parts. The first part represents the elongated box geometry of the underlying conveyor belt. Then, eight small box geometries constitute the mounts for the collectors and emitters of the four instances of the workpiece sensor template. Another two box geometries stand for the mounts of the two instances of the concrete dynamic cylinder template. Finally, three box geometries represent the ramps underpinning the three *exit* locations.

8.2.12 PPU / Sorter / Actuator

The actuator component of the sorter component (see Section 8.2.11) is responsible for generating translational kinematic energy. The translation energy is used for moving workpieces along a predefined motion axis (i.e. the belt surface) towards their *exit* location. The geometric model of the actuator component is shown in Figure 8.35.

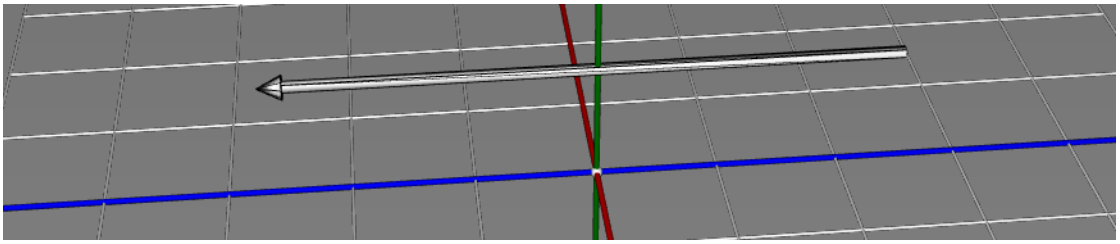


Figure 8.35: PPU sorter actuator scene view.

Furthermore, the actuator component comprises one boolean data input port for (de-)activation of the kinematic energy output. Based on the data input the internal behavior decides to generate a constant kinematic energy output along the depicted axis or not. Note that in this basic model acceleration as well as the weight of the workpiece are not considered. The weight could be modeled, for example, as a generic output of the workpiece template (see Section 8.1.1). Consequently, one could adjust the belt velocity with respect to the sum of weights of the bound workpieces. Note, however,

that more accurate physical models require increased modeling efforts. Such modeling efforts might be necessary in some cases for catching the critical system states during conceptual design. However, in case of the pick and place unit the simple model is considered to be sufficient. Note that a more thorough discussion of the validity of the model is provided later in Section 9.2.

8.2.13 PPU / Sorter / Controller

Finally, the controller component of the sorter component from Section 8.2.11 is responsible for communicating with the environment and coordinating the sorter actuators based on the sorter sensor measurements. The respective behavior of the controller component is depicted in Figure 8.36.

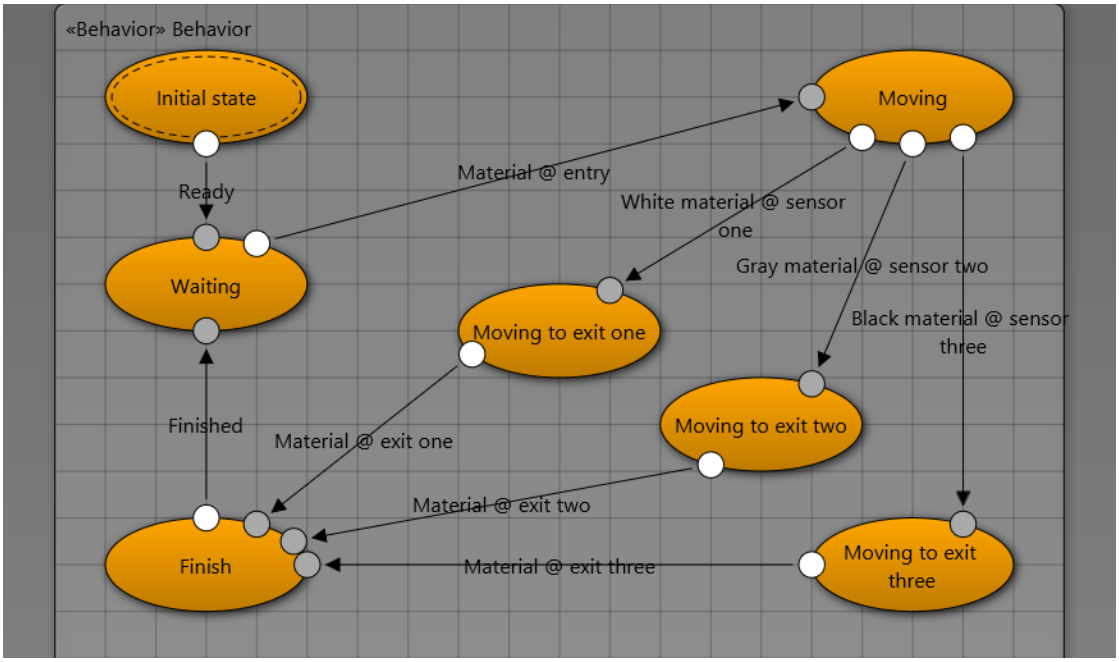


Figure 8.36: PPU sorter controller behavior view.

In the *initial state* the controller sends the *ready* signal and switches to the *waiting* state. Note that the sorter component is ready immediately, while the distributor component (see Section 8.2.2) and the stamper component (see Section 8.2.7) need to perform actions first. Then, if the *start* signal is received from the environment and a

workpiece is detected by the workpiece sensor at the *entry* location of the sorter component the controller switches to the *moving* state and activates the actuator component (see Section 8.2.12). Afterwards, three cases can be distinguished: (1) If a white workpiece is detected by the workpiece sensor at the *exit one* location the controller switches to the *moving to exit one* state. (2) If a gray workpiece is detected by the workpiece sensor at the *exit two* location the controller switches to the *moving to exit two* state. And (3) if a black workpiece is detected by the workpiece sensor at the *exit three* location the controller switches to the *moving to exit three* state. In the first two states the controller waits for a predefined duration before deactivating the actuator component and activating the respective instances of the concrete dynamic cylinder template (see Section 8.1.7), which pushes the workpiece to its expected *exit* location. In the third case an extra instance of the concrete dynamic cylinder template can be omitted. Instead, the controller only waits for a predefined duration before deactivating the actuator component and switching to the *finish* state. Note that, in the last case, the controller assumes that the workpiece has be moved past the end of the conveyor belt, where the *exit three* location can be found. Finally, in the *finish* state the controller sends the *finished* signal to environment and changes to the initial *waiting* state for reiterating the described procedure. Note that, again, malfunctions of the conveyor belt or the (monostable) cylinders were not considered, which is why additional sensors could be omitted.

8.3 Summary and outlook

This chapter demonstrated the result of applying the test-driven method (see Chapter 3), the underlying modeling technique (see Chapter 5), the quality issues (see Chapter 6), and the prototypical tooling (see Chapter 7) to an industry-close showcase: The *pick and place unit*. In particular, the reusable components of the model (called *templates*; see Section 8.1) such as the workpieces, typical sensors, and typical actuators were explained. Then, the main system component as well as the three top-level subcomponents were described: The distributor, the stamper, and the sorter (see Section 8.2). Hereby, the presentation concentrated on the requirements, properties, monitors, and scenarios of the components. Together, the described components provide an example of how future design documents could look like with the proposed approach. The next chapter discusses critically the results, which have been obtained during the experiment.

9 Critical discussion

Subsequently, the *feasibility* of the test-driven design method (see Chapter 3), the *validity* of the system model (see Chapter 8) and the underlying modeling technique (see Chapter 5), as well as the *relevancy* of the quality issues (see Chapter 6) are discussed. To support an objective discussion, the prototypical tooling (see Chapter 7) was instrumented for collecting usage data. The data comprises (1) tool session start and end events, (2) model element creation, modification, and deletion events, (3) test execution start and end events, and (4) syntactic issue appearance and disappearance as well as semantic issue appearance events. Note that for each event its global time stamp was recorded to be able to reconstruct the temporal order of as well as the durations between the individual events. At first, Table 9.1 provides an overview over the collected data.

Category	Measurement	Value	Unit
Tool sessions	Tool session start events	49.00	Events
	Tool session end events	49.00	Events
	Tool session duration	18.44	Hours
Model elements	Model element creation events	3,397.00	Events
	Model element modification events	145,785.00	Events
	Model element deletion events	903.00	Events
Test executions	Test execution start events	686.00	Events
	Test execution end events	686.00	Events
	Test execution duration	3.72	Hours
Quality issues	Syntactic quality issue appearance events	2,574.00	Events
	Syntactic quality issue disappearance events	2,574.00	Events
	Semantic quality issue appearance events	340.00	Events

Table 9.1: Overview of the data collected during tool usage (see Chapter 8).

Overall, the prototypical tool was started and ended 49 times. Hereby, the tool session end events indicate that the prototypical tool did not crash once during the experiment. Furthermore, the 49 tool sessions amount to 18.44 hours of tool usage, during which the entire showcase (see Chapter 8) has been developed from scratch. Then, during development of the showcase 3,397 model elements were created, 145,785 modification

events were recorded, and 903 model elements were deleted again. Note that the modification events include every little change to any attribute of any model element or to any association between any two model elements. Subsequently, during tool usage 686 test executions have occurred including individual test runs as well as the generation of test reports (see Section 7.1.3). Again, none of the test executions crashed, which is indicated by the same number of test execution end events. Moreover, 3.72 hours of tool usage can be attributed to test execution and inspection of the test results, which amounts to 20% of the total tool session duration. Finally, during modeling 2,574 syntactic issues appeared, which all could be resolved successfully as indicated by the respective number of disappearance events. In contrast, during test execution 340 semantic issues could be detected. Note that for semantic issues disappearance events are not recorded because semantic issues are detected only sporadically during test execution.

In the following, first the *feasibility* of the test-driven method is evaluated in Section 9.1. Then, the *validity* of the obtained model is discussed in Section 9.2. Subsequently, the *relevancy* of the quality issues is analyzed in Section 9.3. Finally, the *validity* of the presented study is explained in Section 9.4.

9.1 Method feasibility

First, one needs to consider whether the test-driven design method (see Chapter 3) could be applied successfully during the experiment. From a successful application one can conclude that the test-driven design method is feasible, in principle, for the cyber-physical manufacturing system domain. To answer the question of method feasibility, the actual design process is reconstructed from the usage data, which has been carried out during the experiment. In particular, the model change events are analyzed with respect to the activity sequences (see Section 9.1.1) as well as potential design revisions (see Section 9.1.2). Furthermore, the test execution events are evaluated with respect to potential system increments (see Section 9.1.3).

9.1.1 Activity sequences

Subsequently, the order, in which the design information has been added to the system model (see Chapter 8), is evaluated with respect to the order prescribed by the test-driven design method (see Chapter 3). For this purpose, Figure 9.1 shows the model change events over time assigned to the top-level components of the system model (i.e. the pick and place unit, the distributor, the stamper, and the sorter; see Chapter 8) and classified by the features of the modeling technique (i.e. requirements, ports, scenarios, monitors, components, behaviors, and parts; see Chapter 5).

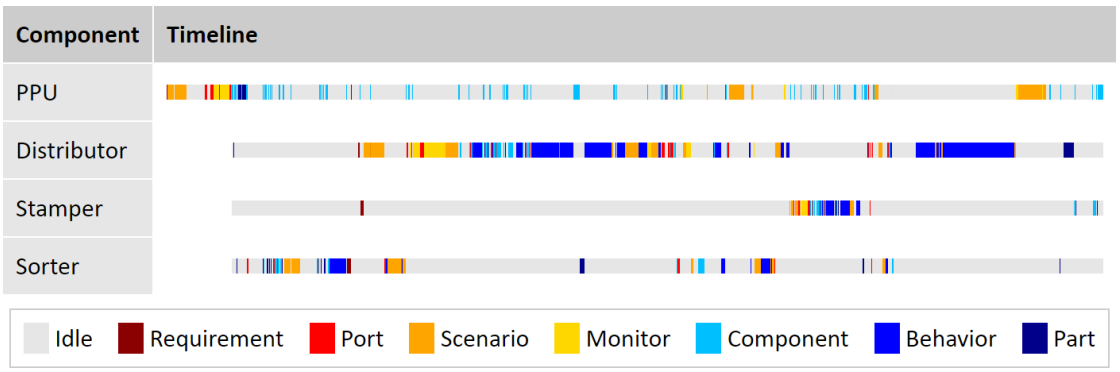


Figure 9.1: Model element creation, modification, and deletion events associated with the top-level components over time.

The diagram shows that for each top-level component of the design indeed the preparation (shades of red) and the implementation (shades of blue) phases can be distinguished. In particular, the scenarios (or test cases) are specified before working on the implementation-level details, which represents a core principle and prerequisite of test-driven development [Bec02]. Furthermore, one can observe that mechanical parts are added in the later phases of the development project. This observation indicates an effective resolution of the mechanical dominance challenge described in Section 1.3.1. Consequently, the sequence of modeling activities adheres mostly to the process prescribed in Chapter 3. Furthermore, the diagram shows that the process indeed proceeds in iterations because work on components, behaviors and parts (i.e. the implementation phase elements) might be followed by work on requirements, ports, scenarios, and monitors (i.e. the preparation phase elements). The iterative process is most distinct for the distributor component and least distinct for the stamper component. However, from the picture it does not become clear whether the iterations represent revisions of existing design knowledge or additions of novel design knowledge. Still, from the iterations one can conclude that a more agile approach has been selected over the traditional waterfall approach, which requires a strict order of the activities.

9.1.2 Design revisions

Next, the question arises whether and how much of the design information has been revised during the iterations observed in the previous section. To answer this question, the model element creation and deletion events are analyzed over time with respect to the features of the modeling technique (i.e. requirements, ports, scenarios, monitors,

components, behaviors, and parts; see Chapter 5). The result of this second analysis is shown in Figure 9.2. Note that this time the elements of the modeling technique are ordered by their absolute frequencies.

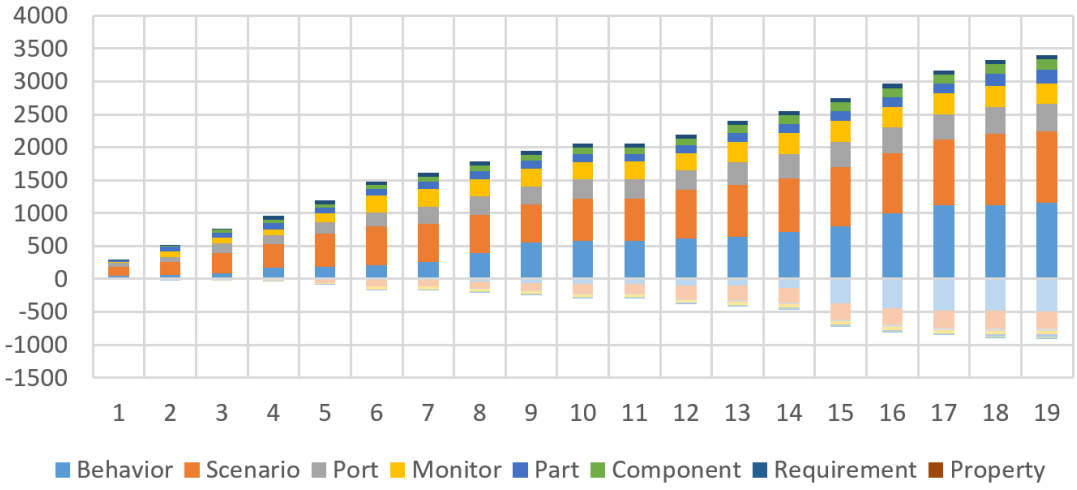


Figure 9.2: Model element creation and deletion events associated with the features of the modeling technique over time.

The diagram shows that during the course of the experiment 903 model elements have been deleted from the model, while 3,397 model elements have been created in total. Consequently, around 27% of the design knowledge was obsolete at some point of the development project and, hence, could be removed completely. Then, 489 out of the 903 deleted model elements represent behaviors (see Section 5.2.5), which amounts to approximately 54% of the deletions. Another 262 deleted model elements can be attributed to scenarios (see Section 5.3.4), which amounts to approximately 29% of the deletions. Furthermore, the most scenario deletion events have been recorded between 10.5 and 11.5 hours after starting the experiment, while the most behavior deletion events could be observed two hours later. When comparing this data to the figure from the previous section (see Figure 9.1), one can see that the scenarios of the pick and place unit have been revised before revising the sorter and distributor components. During the revision, the interaction protocols between the components and their environments have been changed. In particular, the original versions of the pick and place unit and the sorter component started autonomously when observing a workpiece at their start locations, while in the revisions an additional start signal was added. The revisions served to align

the interaction protocols of the pick and place unit and the sorter component with the interaction protocols of the distributor and stamper components, which both required a start signal from the beginning. In summary, one can conclude that indeed a substantial amount of design knowledge (i.e. at least 27%) has been revised during the experiment.

9.1.3 System increments

Finally, the question arises whether the system model has been developed in increments, which is particularly characteristic for test-driven development [Bec02]. To answer this question, the test execution events and the semantic issue appearance events are analyzed. In particular, the points in time are reconstructed when the individual scenarios (or test cases) could be terminated successfully and, thus, when partially complete system models were available. The result of this analysis is provided in Figure 9.3.

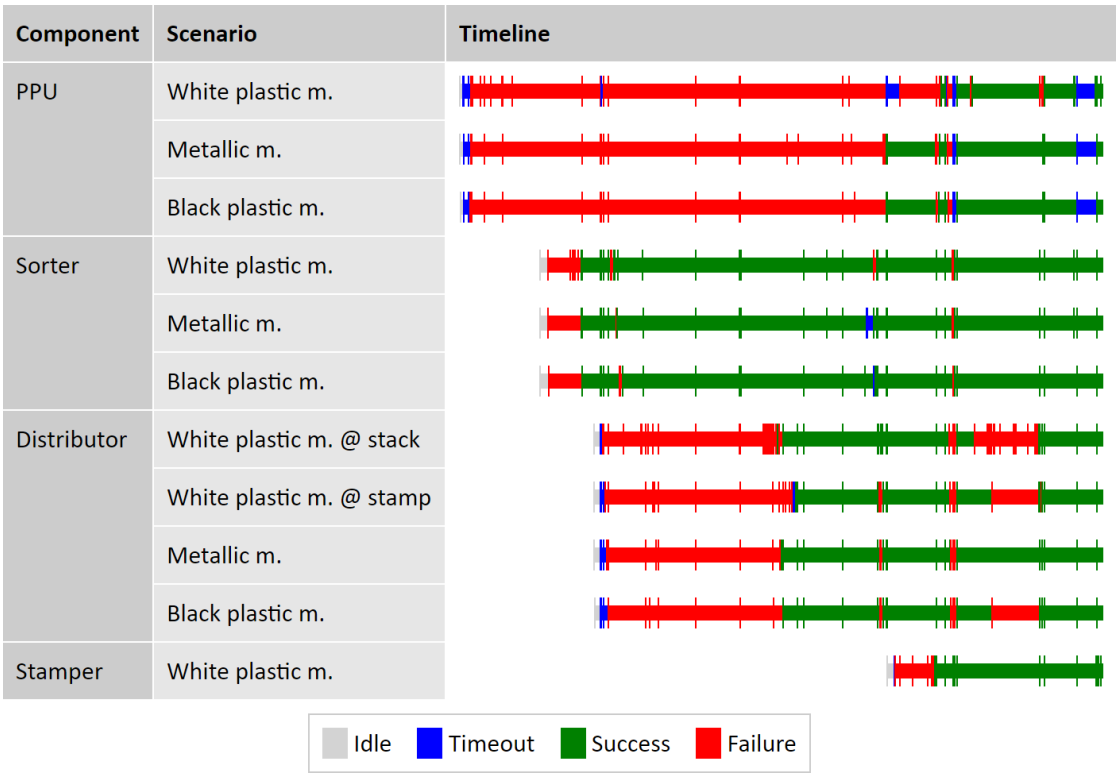


Figure 9.3: Test execution events and respective outcomes associated with the individual components and their scenarios over time.

The diagram shows that first the sorter component (see Section 8.2.11) is implemented. After 2.5 hours the sorter scenarios are executed for the first time leading to severe semantic issues. Then, one hour later all three scenarios terminate successfully. In between the white workpiece scenario is used for testing only, while the gray and black workpiece scenarios are omitted. Nevertheless, all three scenarios are passed simultaneously, which was possible due to the strong similarity in control behavior. Subsequently, the implementation of the distributor component (see Section 8.2.2) started after approximately 4 hours. Between hour 9 and 10 of the experiment the white, gray, and black workpiece at stack scenarios terminate successfully before passing the white workpiece at stamp scenario about 20 minutes later. Again, towards the end of this phase mostly the white workpiece at stack scenario is executed, while the gray and black workpiece at stack scenarios are neglected. However, all three scenarios pass almost simultaneously due to similar control behaviors. In contrast, the white workpiece at stamp scenario took some more time due to significant differences in the control behavior. Thereafter, the sorter and the distributor component are integrated to pass the gray and black workpiece scenarios of the pick and place unit (see Section 8.2.1). The integration of the two components took approximately 3 hours. Most of the time was spent on aligning the interaction protocols of the pick and place unit as well as the distributor and the sorter components. After successful integration the implementation of the stamper component (see Section 8.2.7) started around 12.5 hours after experiment initiation. The implementation of the stamper component took around 1 hour. Then, the stamper component is integrated into the pick and place unit to pass the white workpiece scenario. This time the integration only took about 30 minutes. Consequently, after 14 hours a complete conceptual design of the pick and place unit was available. In the remaining 4.5 hours the behavior of the distributor component is revised to improve readability and maintainability. Furthermore, an additional pick and place unit scenario was created testing ten random workpieces in sequence, which has been omitted in Chapter 8 and is not discussed further due to space limitations. Finally, the mechanical parts of the distributor are revised to reflect more closely a potential physical implementation. In summary, one can conclude that the system model indeed has been developed in increments over the test cases of the pick and place unit and its top-level subcomponents.

9.2 Model validity

Then, the question arises whether the system model from Chapter 8 and, hence, the underlying modeling technique from Chapter 5 indeed provide a valid representation for the conceptual design of cyber-physical manufacturing systems. Note that the validity of the

theoretical concepts underlying the notion of revised spatio-temporal components (see Section 5.2.1) has been discussed already in [Hum11]. Similarly, test automata [KT04] resembling scenarios (see Section 5.3.4) and observer automata [BHJP05] resembling monitors (see Section 5.3.3) are well-known concepts at least for a decade. Therefore, the question is tackled from a different, more practical angle here: In Section 9.2.1 the component architecture of the system model is evaluated, which has been developed during the experiment. Then, in Section 9.2.2 the behavior of the developed system model is analyzed. In both cases, the results of the experiment are compared to an appropriate baseline. In particular, the comparison emphasizes the similarities and differences between the obtained system model and the respective baseline.

9.2.1 System architecture

For assessing the validity of the component architecture of the system model from Chapter 8 the original SysML documentation of the pick and place unit [LFVH13] is used as a baseline. Note that the SysML documentation is assumed to contain a valid and practically relevant architectural description though it has been developed in an academic context only. Subsequently, first the top-level decomposition of the pick and place unit (i.e. the distributor, the stamper, and the sorter components) is compared, before considering the decomposition of the top-level subcomponents.

Pick and place unit

Figure 9.4 provides the component architecture of main the pick and place unit component (see Section 8.2.1), as it has been defined in the original SysML documentation [LFVH13]. Note that the SysML block definition diagram notation is used here. In this notation, the block represent components, while the edges represent containment (or part of) relationships. Furthermore, the black color marks strong similarities between the system model and the baseline, while the red color indicates discrepancies.

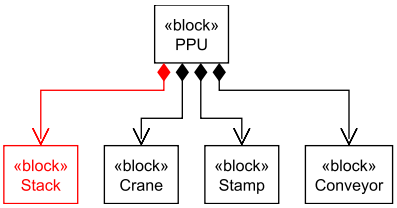


Figure 9.4: Component architecture of the pick and place unit in the SysML documentation [LFVH13].

Fundamentally, both the SysML documentation as well as the system design, which has been developed during the experiment (see Chapter 8), decompose the pick and place unit into three components (or modules), namely the distributor, the stamper, and the sorter. Note, however, that the SysML documentation additionally defines a *stack* module providing a workpiece buffering mechanism at the entry location, which has been neglected in the experiment to reduce the overall effort. Furthermore, note that in the SysML documentation the modules where called differently, namely *crane*, *stamp*, and *conveyor*. The renaming has been carried out to reflect more closely the functions of the different modules rather their physical implementations. Still, one can conclude that the top-level component architectures of the system model and the baseline are identical. Consequently, the decomposition at this stage is assumed to be valid.

Distributor component

Then, Figure 9.5 provides the component architecture of the crane module (or the distributor component; see Section 8.2.2) as defined in the SysML of the pick and place unit documentation [LFVH13]. Again, the black color marks strong similarities between the system model and the baseline, while the red color indicates discrepancies instead. Furthermore, the orange color indicates only minor discrepancies (e.g. different naming or slightly different assignment of responsibilities).

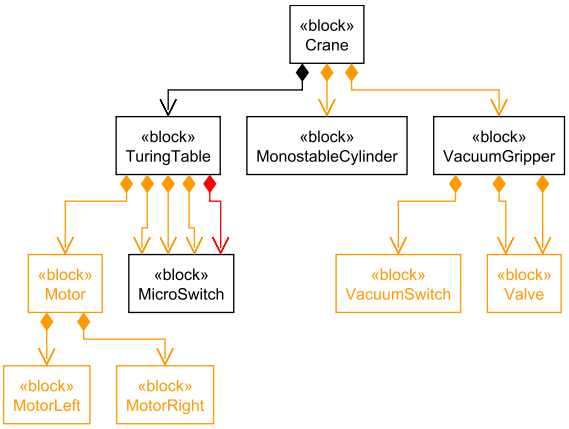


Figure 9.5: Component architecture of the crane module in the SysML documentation [LFVH13].

Some more differences can be observed between the distributor component of the system model and the crane module of the SysML documentation. For example, the system

design assigns two material sensors to the distributor component, while the SysML documentation assumes that the sensor signals are provided by the environment of the distributor component (i.e. no sensors are assigned). Consequently, a more autonomous distributor component design was chosen, while the original design assigned less responsibilities and, hence, more dependencies to the crane module. Furthermore, the system design combines the *monostable cylinder* and the *vacuum gripper* of the SysML documentation into a single lifter component (see Section 8.2.4), which is rotated by the *turning table* (or the twister component; see Section 8.2.3). In particular, in the model the effect of the turning table is stated more explicitly. Then, the system model uses only three *micro switches* (or generic sensor components; see Section 8.1.2), which are assigned to the distributor component directly rather than to the turning table. Note that the additional micro switch is obsolete, which is why it has been omitted. Furthermore, the micro switches were factored out from the turning table and moved them closer to the software controller (see Section 8.2.6) because this design was considered to be more flexible. Finally, the *motor left* and *motor right* components as well as the *vacuum switch* and the *valve* components are contained only implicitly in the system model. In particular, the model only uses one numeric motor control input, whose sign indicates the turning direction, instead of two boolean control inputs, one for turning left, the other for turning right. Hence, the model reflects more closely the function of the distributor component, while the SysML documentation reflects its physical implementation. Furthermore, the model does not prescribe the use of a vacuum gripper and, consequently, leaves some degrees of freedom for the implementation of the lifter component. For example, one could imagine using tong instead of vacuum grippers. Finally, the SysML documentation does not mention anything about the software controller and its integration with the electromechanical system, which represents one of the core components in the system design. In summary, one can conclude that the system architecture contains some modifications, which make interactions more explicit, improve the device flexible, and leave design decisions open. However, these modifications still yield a valid component architecture. In particular, the simplification of the vacuum gripper is considered to be even more appropriate for the conceptual design stage.

Stamper component

Subsequently, Figure 9.6 provides the component architecture of the stamp module (or the stamper component; see Section 8.2.7) as defined in the SysML of the pick and place unit documentation [LFVH13]. Again, the black color marks strong similarities between the system model and the baseline, while the orange color indicates minor discrepancies and the red color indicates major discrepancies.

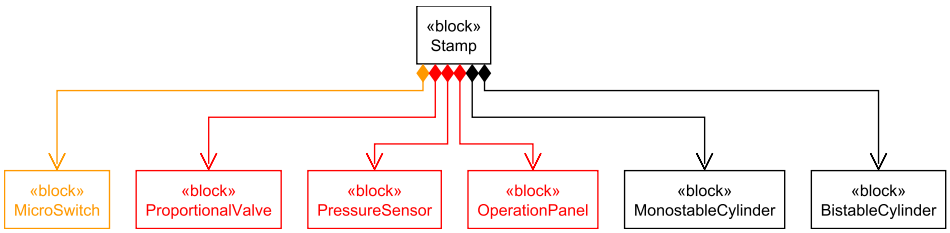


Figure 9.6: Component architecture of the stamp module in the SysML documentation [LFVH13].

Both the SysML documentation and the system model from Chapter 8 assign two cylinders to the stamp module (or the stamper component), one for moving the stamp head, the other for moving the workpiece basket. Then, however, the stamp head (see Section 8.2.9) and the basket (see Section 8.2.8) are not part of the SysML documentation. Consequently, the system design models the interaction between the stamp module and the workpiece in greater detail. This difference can be attributed to the ability of the modeling technique to represent dynamic interactions based on collisions between the stamp head or the basket and the workpiece, which is not available in SysML. On the other hand, the SysML documentation includes a *proportional valve* and a *pressure sensor* as well as *operation panel* component, which are not contained in the system model developed during the experiment. The proportional valve and the pressure sensor component are used to physically implement the stamping process. Consequently, the system model leaves some more degrees of freedom. For example, an engineer could decide to spray the label onto the workpiece instead of stamping the label onto the workpiece. In contrast, the operation panel provides buttons to start the stamp module, to switch to manual control, and to stop the stamp module in the case of emergency. Note that such functionality was neglected in the experiment to reduce the overall effort. Furthermore, the stamp module includes a *micro switch* for sensing the basket location in extended state. This functionality was covered partly using the workpiece sensor (see Section 8.1.3) at the *Stamp* location of the distributor component (see Section 8.2.2). However, note that the micro switch is important to guarantee exact positioning of the basket and to detect malfunctions. Exact positioning could be achieved easily in the model and malfunctions were not considered during the experiment, which is why the micro switch could be omitted. Finally, the SysML documentation again omits the integration of a software controller, which represents a core component in the developed system design. In summary, one can conclude that the developed system design describes some interactions more precisely, while leaving open their physical implemen-

tation. Describing the interactions more precisely has the advantage that the system model is executable and, hence, can be tested automatically. Furthermore, implementation details such as the micro switches were omitted, which were not necessary in the system model to achieve the same functionality as in the physical system. Still, one can conclude that a valid component architecture was obtained due to the strong similarities.

Sorter component

Finally, Figure 9.6 provides the component architecture of the conveyor module (or the sorter component; see Section 8.2.7) as defined in the SysML documentation of the pick and place unit [LFVH13]. Again, the black color marks strong similarities between the system model and the baseline, while the orange color indicates minor discrepancies and the red color indicates major discrepancies.

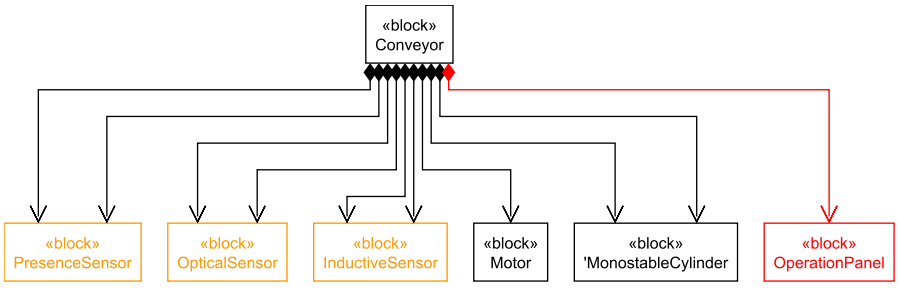


Figure 9.7: Component architecture of the conveyor module in the SysML documentation [LFVH13].

The conveyor module (or the sorter component) is defined almost identically in the SysML documentation and the developed system design. In particular, both the SysML documentation and the system model include a *motor* (or actuator component; see Section 8.2.12) and two *monostable cylinders* (or concrete dynamic cylinders; see Section 8.1.7) for moving workpieces to their target locations. Then, the SysML documentation includes two *presence sensors*, two *optical sensors*, and two *inductive sensors* for detecting the location and type of workpieces on the conveyor belt at four distinct points. The same functionality was covered using only four workpiece sensor components (see Section 8.1.3), which are able to detect both workpiece presence and the workpiece type. Consequently, the SysML documentation is closer to the physical implementation of the sensors, while developed model combines the functionality of the individual sensor types into one component. Hence, the developed model leaves additional degrees of freedom for the actual implementation of the functionality, which is desired during conceptual

design. Finally, the SysML documentation again includes an *operation panel* module for starting the conveyor, switching to manual operation, and stopping the conveyor in case of emergencies. Analogous to the stamper component this functionality was omitted to reduce the scope of the experiment. In summary, one can conclude that a component architecture of the sorter component was obtained, which shows strong similarity to the component architecture of the SysML documentation. Furthermore, the developed component architecture simplifies implementation details about the different sensor types, which is desired during conceptual design.

9.2.2 System behavior

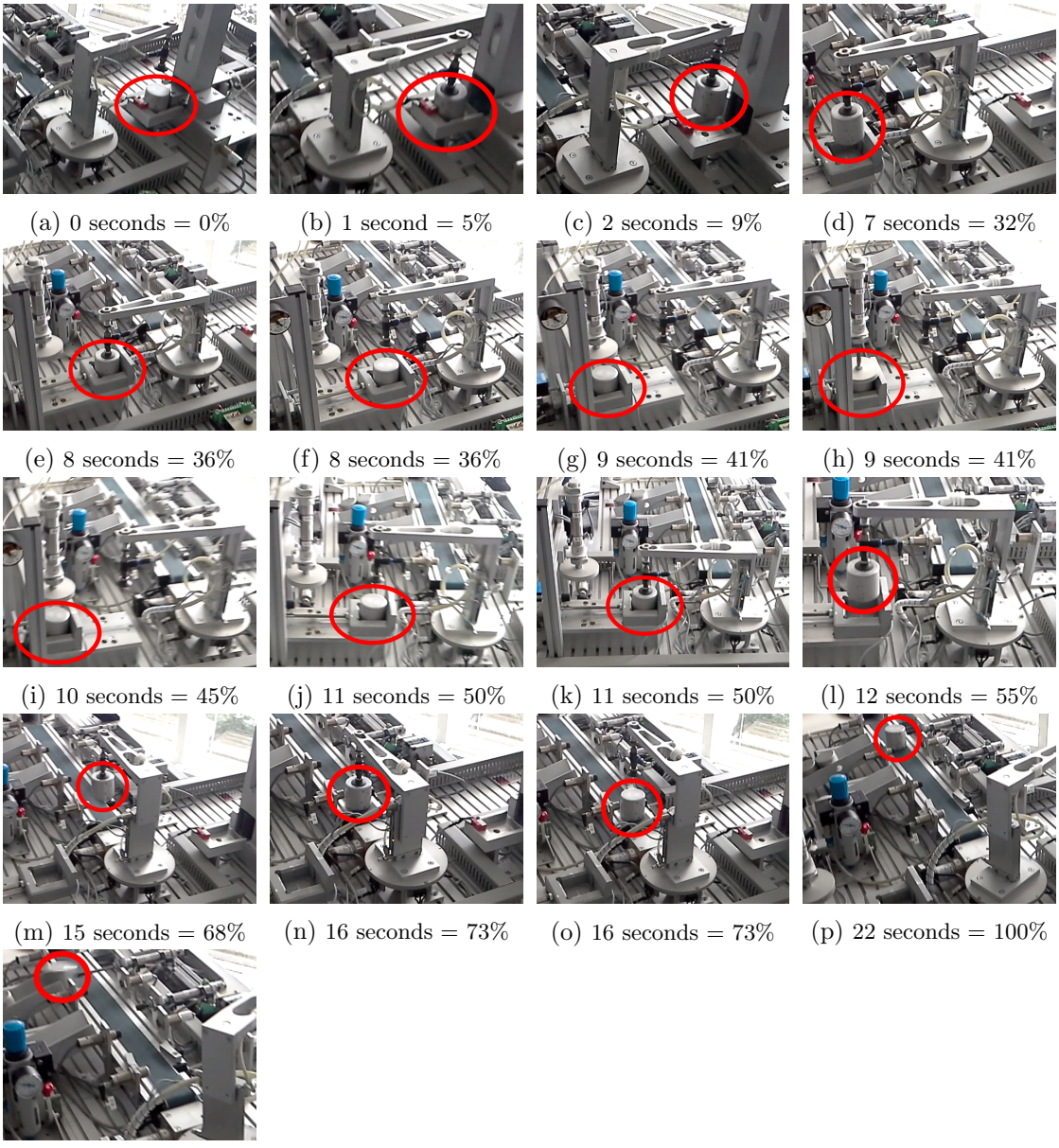
Then, for assessing the validity of the system behavior of the model from Chapter 8 the physical system installed at the Institute for Automation and Information Systems, Technische Universität München, Prof. Dr.-Ing. Vogel-Heuser[1] is used as a baseline rather than the SysML documentation [LFVH13]. Note that the SysML documentation is not suited for this task due to its informal semantics, which entails ambiguities and potential misinterpretation. However, in the case of the physical system one is limited to the observable states, which most importantly include the position and orientation of the individual components as well as critical states reachable during system operation. In the following, first the validity of the spatial configurations is discussed before evaluating the reachability of critical states.

Spatial configurations

Subsequently, the spatial configurations of the individual components during system operation both of the physical system and the system model are compared. In particular, the white workpiece scenario (see Section 8.2.1) is considered, which entails the most complex behavior of the pick and place unit and which subsumes the behaviors entailed by the gray and black workpiece scenarios. In the following, first the sequences of spatial configurations during workpiece processing are analyzed. Second, the timing of the spatial configurations relative to the entire processing time is evaluated.

Sequence of spatial configurations For comparing the sequences of spatial configurations, photographs of the physical system (see Figure 9.8) and screenshots of the system model (see Figure 9.9) during system operation are provided. Furthermore, in each photograph and screenshot the workpiece location is marked with a red circle. Also, the durations are provided after which each individual spatial configuration is reached

[1] https://www.ais.mw.tum.de/en/homepage/



(a) 0 seconds = 0% (b) 1 second = 5% (c) 2 seconds = 9% (d) 7 seconds = 32%

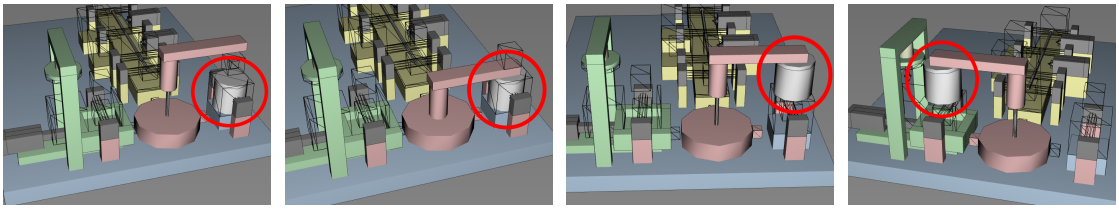
(e) 8 seconds = 36% (f) 8 seconds = 36% (g) 9 seconds = 41% (h) 9 seconds = 41%

(i) 10 seconds = 45% (j) 11 seconds = 50% (k) 11 seconds = 50% (l) 12 seconds = 55%

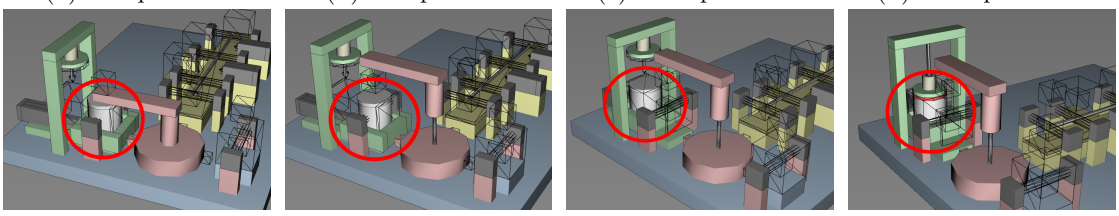
(m) 15 seconds = 68% (n) 16 seconds = 73% (o) 16 seconds = 73% (p) 22 seconds = 100%

(q) 22 seconds = 100%

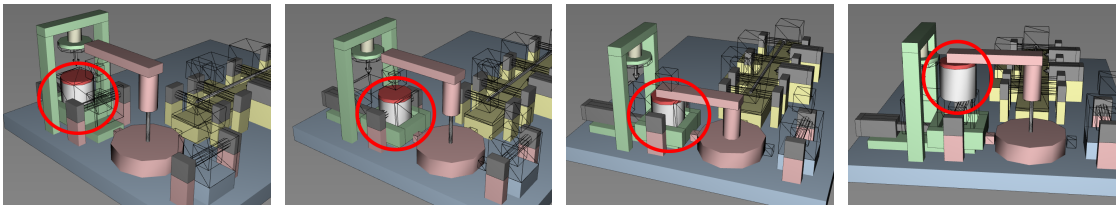
Figure 9.8: Sequence of spatial configurations in the physical system.



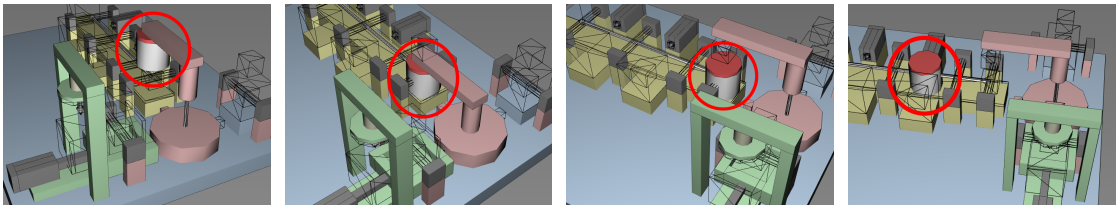
(a) 0 steps = 0% (b) 7 steps = 7% (c) 13 steps = 12% (d) 26 steps = 25%



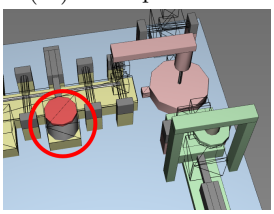
(e) 32 steps = 30% (f) 38 steps = 36% (g) 45 steps = 42% (h) 51 steps = 48%



(i) 56 steps = 53% (j) 63 steps = 59% (k) 69 steps = 65% (l) 76 steps = 72%



(m) 83 steps = 78% (n) 90 steps = 85% (o) 97 steps = 92% (p) 105 steps = 99%



(q) 106 steps = 100%

Figure 9.9: Sequence of spatial configurations in the developed system model.

both in absolute and relative numbers. For the physical system the absolute numbers are measured in seconds, while for the system model they are measured in computation steps (see Section 6.2 for more information about scenario computations). In contrast, the relative numbers are calculated with respect to the overall workpiece processing durations, i.e. the durations after which the final spatial configurations are reached.

The two figures reveal that both the physical system and the virtual system traverse almost the same 17 spatial configurations in exactly the same order. First, both the workpiece and the vacuum gripper of the crane module are located at the stack, while the vacuum gripper is in extended state (see Figures 9.8a and 9.9a), before the vacuum gripper is moving down and gripping the workpiece (see Figures 9.8b and 9.9b). Then, the vacuum gripper is moving up again and, hence, lifting the workpiece (see Figures 9.8c and 9.9c) and, subsequently, turning to the stamp (see Figures 9.8d and 9.9d). At the stamp, the vacuum gripper is moving down and releasing the workpiece into the basket of the stamper component (see Figures 9.8e and 9.9e) before moving up again and waiting (see Figures 9.8f and 9.9f). In the next step, the basket of the stamper component is retracted (see Figures 9.8g and 9.9g) and the stamp head is pressed onto the workpiece (see Figures 9.8h and 9.9h) before retracting the stamp head again (see Figures 9.8i and 9.9i). Then, the basket of the stamper component is extended (see Figures 9.8j and 9.9j) and the vacuum gripper of the crane module is moving down and gripping the workpiece (see Figures 9.8k and 9.9k). Subsequently, the vacuum gripper is moving up again and, hence, lifting the workpiece (see Figures 9.8l and 9.9l) before turning to the conveyor component (see Figures 9.8m and 9.9m) as well as moving down and releasing the workpiece (see Figures 9.8n and 9.9n). In the next step, the vacuum gripper is moving up again (see Figures 9.8o and 9.9o). Finally, in the physical system the workpiece is moved to the second exit location and ejected, while the vacuum gripper is turning to the stack (see Figures 9.8p and 9.8q). In contrast, in the system model the workpiece is moved to the first exit location and ejected, while the vacuum gripper is staying at the conveyor (see Figures 9.9p and 9.9q). Hence, one can observe a slight difference in the exit location of the stamped workpiece as well as the control behavior of the crane component. In particular, in the physical system the control behavior of the crane component assumes that the next workpiece will be found at the stack location. In contrast, in the system model the control behavior of the crane component assumes that the next workpiece can be found also at the stamp location (or some other location). Hence, the control behavior of the system model is more flexible with respect to potential changes in the material flow, but the control behavior of the physical system is more efficient with respect to the current requirements. Which version to prefer depends on the importance of flexibility over efficiency, which was not considered further during the experiment. In summary, one can conclude that the spatial configurations and their

sequential order – as found in the virtual system – are valid with respect to the physical system. In particular, only minor differences could be observed, which can be traced back to minor differences in functional and non-functional requirements.

Timing of spatial configurations Then, the question arises whether the transitions between the spatial configurations are valid. To answer this questions, the relative durations after which the spatial configurations are reached and, hence, the durations of the transitions between the spatial configurations are compared. The results of this evaluation is shown in Figure 9.10. The x-axis of the diagram shows the individual spatial configurations from the previous section. In contrast, the y-axis shows the relative duration, after which the respective spatial configuration is reached. Finally, the diagram contains one curve for the physical system and a second curve for the system model. Note that one curve being above the other curve indicates the one system being slower than the other system in achieving the respective spatial configuration and vice versa. Also, note that the gradient between two subsequent spatial configurations denotes the speed and duration of the respective transition. In particular, the gradients of the physical system and the system model can be used to compare the duration of transitions.

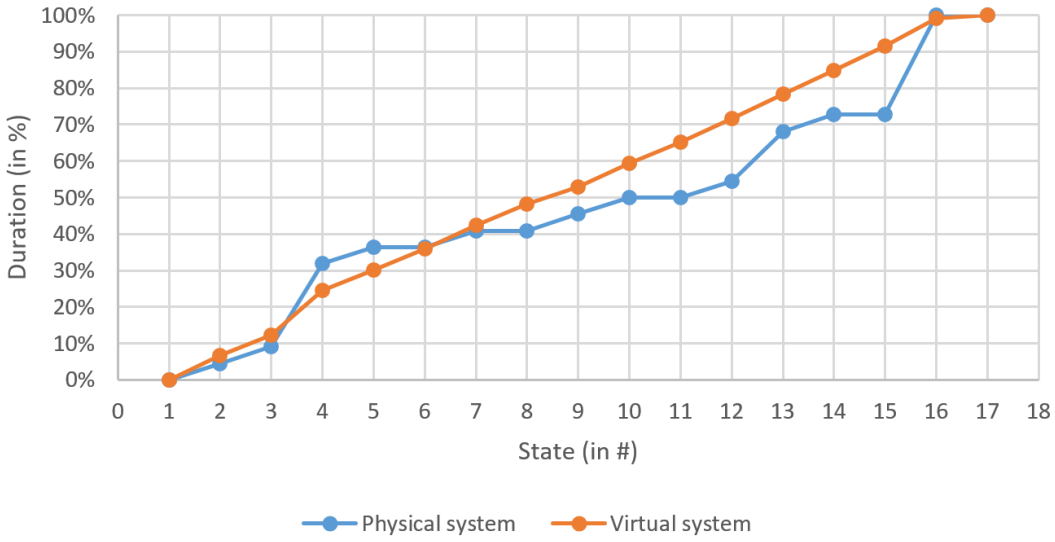


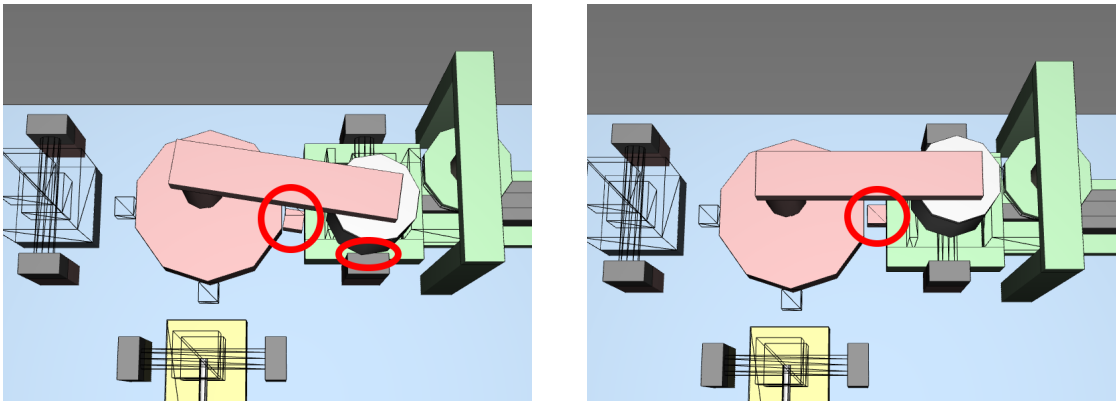
Figure 9.10: Relative timing of the spatial configurations in the physical system and the developed system model.

The diagram shows that some transitions are similarly fast in the physical system and the system model (e.g. the transition from spatial configuration 1 to spatial configuration 2 and 3). These transitions do not impair the validity of the system behavior. Consequently, the discussion focuses on the significant differences between the two systems. The first significant deviation between the two systems can be observed during the transition from spatial configuration 3 to spatial configuration 4, which represents turning the vacuum gripper and, hence, the workpiece from the stack to the stamp location. Hereby, the physical system uses more than 20% of the overall workpiece processing time, while the system model uses only slightly more than 10%. Consequently, the vacuum gripper turns faster in the system model than in the physical system. Then, the next deviation can be observed in the transition from spatial configuration 5 to spatial configuration 6, namely moving up the vacuum gripper after releasing the workpiece at the stamp. In this case, the physical system requires almost no time, while the system model requires slightly more than 5% of processing time. Similar deviations can be observed for the transition from spatial configuration 7 to spatial configuration 8 (i.e. pressing the stamp head onto the workpiece), the transition from spatial configuration 10 to spatial configuration 11 (i.e. moving the vacuum gripper down and gripping the workpiece at the stamp), and the transition from spatial configuration 14 to spatial configuration 15 (i.e. moving the vacuum gripper down at the conveyor location). The reason for these deviations is that the monostable cylinder (or the abstract static cylinder; see Section 8.1.4), which is built into the crane module, switches much faster into the stable configuration in the physical system than in the virtual system. Subsequently, the transition from spatial configuration 12 to spatial configuration 13 again indicates that the vacuum gripper turns faster in the virtual system than in the physical system. Finally, a deviation can be observed for the transition from spatial configuration 15 to spatial configuration 16, namely moving the workpiece from the conveyor location to the respective exit location. In particular, the physical system uses approximately 30% of the processing time, while the system model uses only 10%. Consequently, the conveyor motor is faster in the system model than in the physical system. Note, however, that in the system model the first exit is used, while in the physical system the second exit is used. Consequently, in the system model the workpiece has to travel less distance than in the physical system. In summary, one can conclude that there are significant differences in the timing of the spatial configurations, which can be attributed to the different behaviors of monostable cylinders as well as the crane and conveyor motors. However, such behaviors are not expected to be exact during conceptual design to reduce effort and maintain focus on important aspects such as causalities and the logics of material flow as well as energy and data exchange. Consequently, the obtained timing behavior can be considered to be valid for the purpose of conceptual design.

Critical states

Finally, the reachability of critical states both in the physical system and in the system model are compared. In particular, critical states are distinguished that (1) could be observed in both the physical system and the system model, (2) could be observed in the physical system only, and (3) could be observed in the system model only. Hereby, critical states refer to states that inhibit the correct functioning of the pick and place unit as prescribed by its requirements and test cases (see Section 8.2.1). Such states include, for example, the collision between parts of different components.

Observable in both systems Some critical states could be observed both in the physical system and in the system model. One example is discussed here, namely the angular positioning of the vacuum gripper, which depends on the angular velocity of the turning table, the signal delays of sensors and actuators (including conversion between analog and digital representations, communication via network equipment and protocols), and the cycle time of the programmable logic controller (PLC). Only if the correct ratio between the parameters is achieved, correct angular positioning of the vacuum gripper can be guaranteed. This problem is illustrated in Figure 9.11.



(a) Low angular velocity versus signal delay

(b) Medium angular velocity versus signal delay

Figure 9.11: Critical state observed in both the physical and the system model.

The problem can be explained as follows: If the angular velocity of the vacuum gripper is low compared to the signal delays and the PLC cycle time (see Figure 9.11a), then the vacuum gripper stops too early. Consequently, the workpiece cannot be placed into the basket correctly. In the best case, only the current workpiece is lost and the system

can return to normal operation autonomously. In the worst case, the stamp module is not usable anymore and human operator intervention is required. The problem can be solved by adjusting the ratio between the angular velocity, the signal delays, and the PLC cycle time (see Figure 9.11b). If the ratio is correct, then the reaction of the vacuum gripper is delayed. Consequently, the vacuum gripper can be stopped at the correct location and the workpiece can be placed exactly into the basket. Note, however, that the angular velocity can be too high as well. As a result, the vacuum gripper might overshoot. The fine-tuning of the respective parameters requires more precise physical behavior. However, the fine-tuning is considered to be inappropriate during conceptual design. Rather, the conceptual design is required to indicate such potential problems, as observed during the experiment.

Observable in the physical system only Then, some critical states could be observed in the physical system, but not the system model. Again, one example is provided, namely the unexpected intervention of a human operator. Note that, in principle, any other environmental component could be used instead, which has spatial extent and / or is able to exert kinetic energy. Furthermore, note that the selected example is somewhat artificial, but represents well a broader class of environmental interferences potentially leading to the malfunctioning of the pick and place unit. Subsequently, Figure 9.12 shows the unexpected human intervention and the unpredictable physical effect.

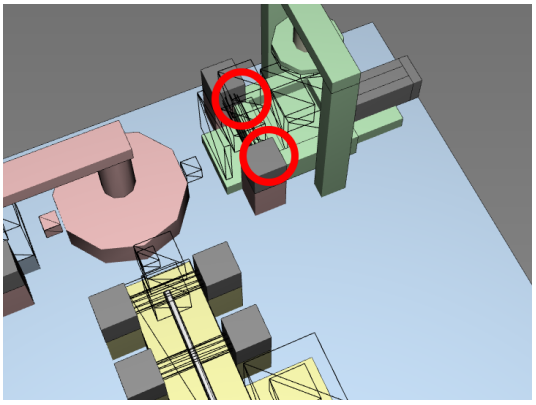
(a) Unexpected human intervention (b) Unpredictable physical effect

Figure 9.12: Critical state observed in the physical system only.

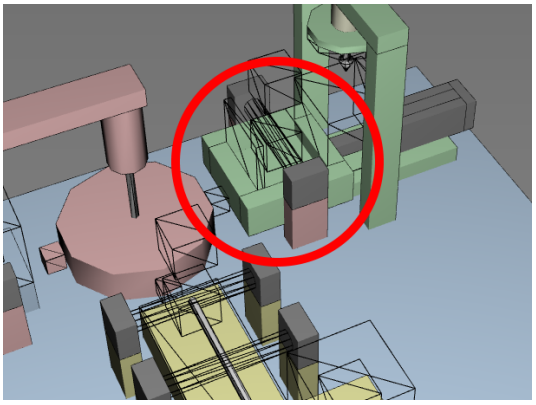
In the selected example, the human operator places a workpiece manually into the basket of the stamp module (see Figure 9.12a). At the same time, the crane module already

has picked up a workpiece at the stack and is turning towards the stamp location. When the crane reaches the stamp location, the workpiece picked up by the vacuum gripper and the workpiece placed into the basket by the human operator collide (see Figure 9.12b). Consequently, the vacuum gripper loses the workpiece and the lost workpiece performs an uncontrolled motion due to gravity and impact forces. Depending on where the lost workpiece comes to rest, the functioning of the pick and place unit is impaired or not. For example, sensor areas might be obscured or motion paths might be jammed. Note that such situations could have been covered in separate scenarios of the pick and place unit (see Section 8.2.1) placing one workpiece at the start and the other workpiece at the stamp location. Consequently, the collision between the two workpieces could have been detected already during conceptual design. However, the uncontrolled motion is difficult to describe in the model. Here, multi-body system dynamics (see Section 2.1.2) might be more suited to analyze the effects. Furthermore, it requires experience to select the scenarios that need to be considered during conceptual design.

Observable in the system model only Finally, some critical states could be observed in the system model, while not being observable in the physical system. For example, a problem with the spatial extent and location of workpiece sensors (see Section 8.1.3) has been detected while integrating the stamper component (see Section 8.2.7) into the pick and place unit (see Section 8.2.1). Note at this point the distributor and sorter components have been integrated successfully already. Figure 9.13 illustrates the original problem and its solution by means of resizing and repositioning the workpiece sensors.



(a) Erroneous stamper integration



(b) Revised stamper integration

Figure 9.13: Example of a critical state not covered by the physical system.

As mentioned previously, the problem occurred when integrating the stamper component into the pick and place unit after approximately 13.5 hours. At that point in time the pick and place unit already was able to pass the gray and black workpiece scenarios. Furthermore, the implementation of the stamper component has been completed successfully. However, during integration of the stamper component into the pick and place unit, it was detected that the basket did not fit between the emitter and collector of the workpiece sensor installed at the stamp location (see Figure 9.13a). Note that the workpiece sensor belongs to the distributor component (see Section 8.2.2) and not to the stamper component, which is why the problem has not been detected earlier. To resolve the problem, the spatial extent of the workpiece sensor as well as its mounts has been reduced. Consequently, the basket did not collide with the respective parts anymore and the system was able to pass the white workpiece scenario also (see Figure 9.13b). Note that alternatively the emitter and the collector could have been placed further apart or the spatial extent of the basket could have been reduced. Furthermore, note that these design decision might constrain the technologies and principles that can be used for their physical implementation. Consequently, the presented example represents an important case for conceptual design. Also, note that these problems could not be observed in the physical system, but might have occurred during the original design and implementation phases. However, information about these phases was not available.

9.3 Issue relevancy

After having evaluated the feasibility of the test-driven design method in Section 9.1 as well as the validity of the system model and the underlying modeling technique in Section 9.2, the question arises whether and to what degree the quality issues (see Chapter 6) are relevant during conceptual design of cyber-physical manufacturing systems. Note that, here, an objective rather than a subjective relevancy measure is used, which can be derived from the data collected during tool usage. In the following, the relevancy of the quality issues is discussed separately for the syntactic quality issues in Section 9.3.1 and the semantic quality issues in Section 9.3.2.

9.3.1 Syntactic issues

First, the relevancy of the syntactic issues is discussed (see Section 6.1), which are concerned with the rules and constraints of the modeling technique (see Chapter 5). Subsequently, it is evaluated how often syntactic issues of a certain kind appeared during the experiment. Then, the time it took until the quality issues could be resolved is analyzed. Both measures together determine an objective notion of relevancy.

Frequencies of syntactic issues

The first diagram (see Figure 9.14) provides for each kind of syntactic issue the relative frequency of appearances throughout the experiment. Note that the incompleteness issues (see Section 6.1.1) are listed in detail, while the inconsistency issues (see Section 6.1.2 are aggregated into a single value. The reason for the aggregation is that only few inconsistency issues appeared throughout the experiment (i.e. 16 events or 1% of appearances) compared to the large number of incompleteness issues (i.e. 2,558 events or 99% of appearances).

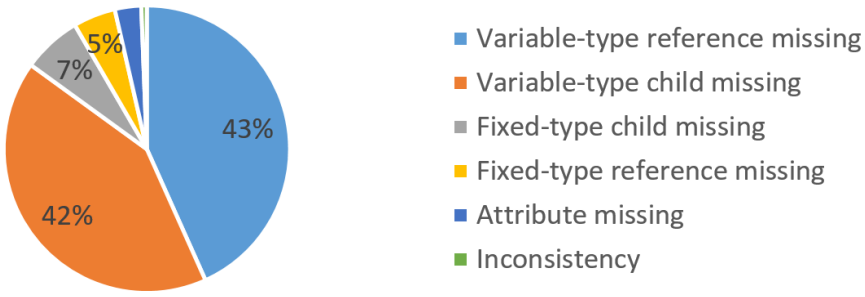


Figure 9.14: Relative frequency of the appearances of syntactic issues.

The diagram reveals that most appearances (i.e. 43%) are variable-type reference missing issues (see Section 6.1.1). When looking deeper into the data, one can observe that one third of the variable-type reference missing issues relates to the observation read by observation expressions (see Section 5.1.3) and another third relates to the observation written by actions (see Section 5.1.3). The remaining variable-type reference missing issues are spread across the model. Then, 42% of the issue appearances fall into the variable-type child missing category (see Section 6.1.1). When looking closer at the data, one can observe that 40% of the variable-type child missing issues relate to the arguments of nary expressions (see Section 5.1.3), one third relates to the expression of actions (see Section 5.1.2), and one fourth relates to the default expression of observations (see Section 5.1.1). Again, the remaining variable-type child missing issues are spread across the model. Thereafter, 7% of the appearances are fixed-type child missing issues (see Section 6.1.1). Hereby, all but one fixed-type child missing issues relate to the guard of transitions (see Section 5.1.2), while the remaining issue refers to the main component of the project. Note that, in principle, these issues could be resolved automatically. Hence, there is space for improvement of the prototypical tooling. Subsequently, 5%

belong to fixed-type reference missing issues (see Section 6.1.1). Finally, 3% of the appearances concern missing attributes (see Section 6.1.1). One third of the attribute missing issues relate to the name of transitions (see Section 5.1.2) and 20% relate to the name of observations (see Section 5.1.1). Again, the remaining attribute missing issues distribute across the model. One can conclude that most syntactic issues relate to the incompleteness of the model. One possible reason might be, that the model was built from scratch. Consequently, the numbers might look different when revising an existing design. However, these correlations are not investigated further here. Furthermore, most of the incompleteness issues relate to the behavioral aspects of the system rather than the static structure and spatial extent. A possible interpretation is that the behavior is much more complex to describe than the static aspects. Finally, the analysis revealed potential for improvement of the prototypical tooling. In particular, certain design objects could created automatically and, hence, some issue appearances could be avoided.

Durations of syntactic issues

Then, the second diagram (see Figure 9.15) provides for each kind of syntactic issue the minimum, average, and maximum absolute duration in minutes between appearance and disappearance of the respective issues. Note that again the incompleteness issues (see Section 6.1.1) are listed individually, while the inconsistency issues (see Section 6.1.2) are aggregated into one single node. Furthermore, note that the issue categories are sorted by average duration in descending order.

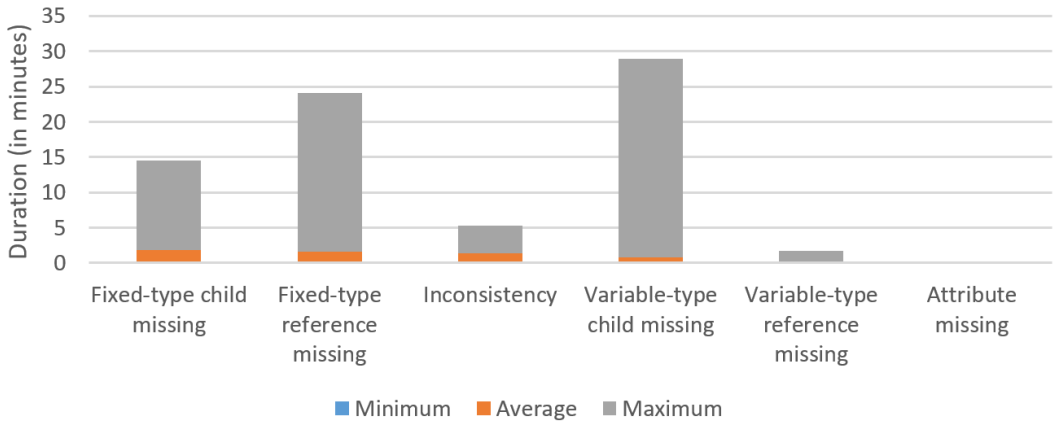


Figure 9.15: Absolute duration until the disappearances of syntactic issues.

The diagram shows that the fixed-type child missing issues (see Section 6.1.1) take the longest to be resolved with 1:50 minutes on average and 14:29 minutes in the worst case. The fixed-type child missing issues mostly comprise the guard of transitions (see Section 5.1.2) as well as the main component of the project. Again, note that these issues could be resolved automatically and, hence, the issue duration could be reduced. Then, the fixed-type reference missing issues (see Section 6.1.1) take 1:38 minutes on average to be resolved, while in the worst case 24:08 minutes are used. The fixed-type reference missing issues comprise the final step of scenarios (see Section 5.3.4) with 4:11 minutes on average and 24:08 minutes in the worst case, the initial state of executables (see Section 5.1.2) with 1:58 minutes on average and 9:41 minutes in the worst case, the component definition of material life ports (see Section 5.2.2) with 14 seconds on average and 52 seconds in the worst case, and the component definition of component references (see Section 5.2.1) with 12 seconds on average and 37 seconds in the worst case. Subsequently, the inconsistency issues (see Section 6.1.2) take 1:21 minutes on average and 5:15 minutes in the worst case to be resolved. The inconsistency issues comprise the type of expressions contained in actions (see Section 5.1.2) as well as the type of default expressions contained in observations (see Section 5.1.1). Then, the variable-type child missing issues take 45 seconds on average and 28:56 minutes in the worst case to be resolved. The variable-type child missing issues comprise, besides others, the expression of guards (see Section 5.1.2) with 2:22 minutes on average and 28:56 minutes in the worst case, the volume of material ports (see Section 5.2.2) with 1:02 minutes on average and 2:45 minutes in the worst case, and the default expression of observations (see Section 5.1.1) with 42 seconds on average and 9:57 minutes in the worst case. Then, the variable-type reference missing issues take 5 seconds on average and 1:44 minutes in the worst case to be resolved. The variable-type reference missing issues comprise the observation written by actions (see Section 5.1.2) as well as the observation read by observation expressions (see Section 5.1.3). Finally, the attribute missing issues take 1 second on average and 13 seconds in the worst case to be resolved. The attribute missing issues comprise the name of various model elements such as the states and transitions of executables (see Section 5.1.2) or observations (see Section 5.1.1). One can conclude that most time is spend on resolving incompleteness issues while working on behavioral aspects of the system. Also, inconsistencies can be resolved in comparably little time.

9.3.2 Semantic issues

Subsequently, the relevancy of the semantic issues is evaluated (see Section 6.2), which are concerned with the meaning of the design information expressed using the modeling technique from Chapter 5. Again, first the kinds of semantic issues, which appeared

during the experiment, are analyzed before considering the time it took to resolve them. Together these measures constitute an objective notion of relevancy.

Frequencies of semantic issues

The first diagram (see Figure 9.16) provides for each kind of semantic issue the relative frequency of appearances throughout the experiment. Note that only semantic issues (and their causes) are listed, which actually appeared during the project. Furthermore, the issues are sorted by relative frequency in descending order both in the figure and in the following discussion.

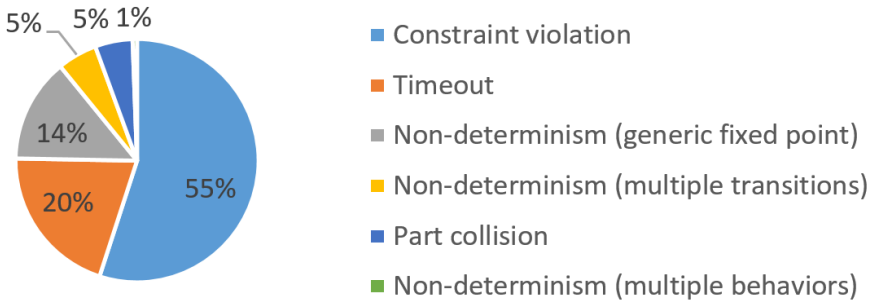


Figure 9.16: Relative frequency of the appearances of the semantic issues.

The diagram shows that more than half (i.e. 55%) of the semantic issues are constraint violations (see Section 6.2.1). When looking closer at the data, one can observe that 13% of the constraint violations can be assigned to moving gray and black workpieces from start to exit in time, which were the first functions implemented during the experiment. The remaining constraint violations distribute rather equally across the properties of the system model (see Chapter 8). Thereafter, 20% of the semantic issues are computation timeouts (see Section 6.2.2). 62% of the computation timeouts relate to the scenarios of the pick and place unit (see Section 8.2.1), while only 29% of the issues can be assigned to the distributor (see Section 8.2.2). Obviously, computation timeouts are more likely the more complex the respective component is. Note that complexity comprises both static spatial extent and dynamic state space. Then, 14% of the semantic issues are non-determinisms caused by general fixed points of weakly causal behaviors (see Section 6.2.2). Hereby, 60% of the generic fixed point non-determinisms are caused by the distributor component (see Section 8.2.2), while 34% can be assigned to the pick and place unit (see Section 8.2.1). The data indicates that generic fixed point

non-determinisms are more likely the more complex the component architecture is. This correlation is not surprising when considering that generic fixed-point issues only occur when composing behaviors of different components (see Section 6.2.2). Subsequently, 5% of the semantic issues represent non-determinisms caused by multiple enabled transitions in the same executable (see Section 6.2.2). A more detailed analysis shows that 89% of the multiple-transitions non-determinisms are caused by the ten random workpieces scenario, which was not described in Chapter 8. In particular, the non-determinism was added on purpose to generate random sequences of workpieces during testing. Furthermore, 5% of the semantic issues indicate part collisions (see Section 6.2.2). Hereby, 48% of the part collisions were detected when integrating the stamper component (see Section 8.2.7) into the pick and place unit (see Section 8.2.1). Note that this case has been discussed previously in Section 9.2.2. Finally, 1% of the issues are non-determinisms caused by multiple behaviors writing the same output port (see Section 6.2.2), which occurred during distributor component development (see Section 8.2.2). The reason for the occurrence of this issue was a software bug in the simulation engine of the prototypical tooling, which could be identified and resolved quickly. Consequently, the issue occurrence can be neglected with respect to issue relevancy. At last, non-determinism issues due to multiple bindings or generic fixed points with mutual forwarding of kinetic energy did not occur during the experiment. In summary, one can conclude that the proposed approach indeed is able to detect a wide variety of semantic issues in conceptual designs. Most importantly, test execution can be used effectively to uncover issues with the implementation of requirements and manufacturing processes, which are formalized as properties (see Section 5.3.2). But, also intrinsic semantic issues such as part collisions and non-determinisms can be detected in practice. Finally, the experiment has shown that semantic issues might also occur due to software bugs in the simulation engine. However, this problem decreases with simulation engine maturity.

Durations of semantic issues

Subsequently, the second diagram (see Figure 9.17) provides for each kind of semantic issue the minimum, average, and maximum absolute duration in minutes between appearance and disappearance of the respective issues. Again, the semantic issues are sorted by average duration in descending order. Furthermore, note that disappearance events are not recorded explicitly for semantic issues. Rather, a semantic issue is said to disappear, if it has been detected in the preceding scenario execution, but not in the current scenario execution. However, note that the semantic issue might not have disappeared, but might be occluded by another semantic issue. Furthermore, semantic issues also might have disappeared much earlier than revealed by test execution.

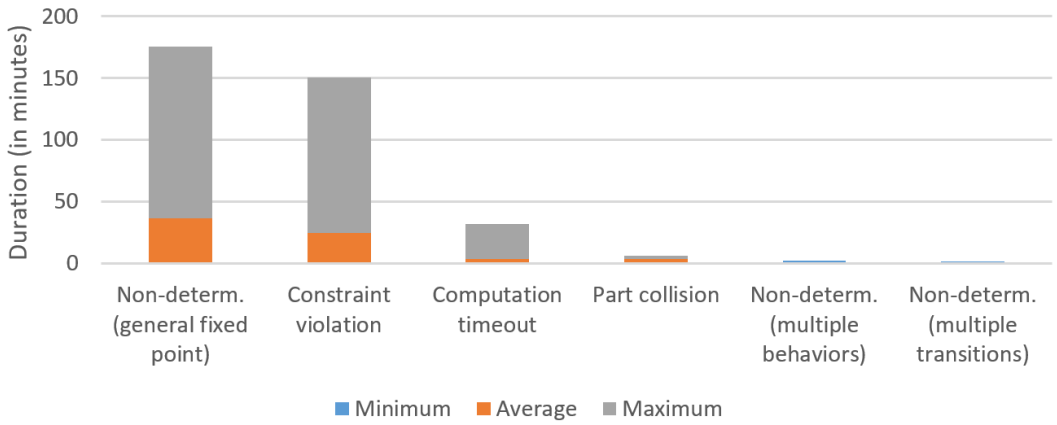


Figure 9.17: Absolute duration until the disappearances of the semantic issues.

The diagram shows that non-determinisms caused by generic fixed points of weakly causal behaviors (see Section 6.2.2) take the longest to disappear with 36:02 minutes on average and 2:55 hours in the worst case. When looking closer at the data, the generic fixed points of the pick and place unit (see Section 8.2.1) took 1:02 hours on average and 2:55 hours in the worst case to disappear, while the generic fixed points of the distributor component (see Section 8.2.2) took 25:03 minutes on average and 56:11 minutes in the worst case. The difference can be explained with the fact that first the distributor component had to be implemented correctly, before the scenarios of the pick and place unit could be considered any further. Then, constraint violations (see Section 6.2.1) took 24:21 minutes on average and 2:30 hours in the worst case to disappear. The largest durations have been recorded for moving gray and black workpieces from entry to exit with 1:10 hours on average and 2:30 hours in the worst case. Note that this function was the first to be implemented during the experiment, which explains its dominant position here and in the frequencies discussed in the previous section. The second largest durations have been recorded for positioning workpieces under the stamp head with 1:03 hours both on average and in the worst case. Note that this duration also correlates with the duration that was needed for implementing the stamper component (see Section 9.1.3). Subsequently, the computation timeouts (see Section 6.2.2) took 3:26 minutes on average and 31:59 minutes in the worst case to disappear. The computation timeouts distribute among the stamper component (see Section 8.2.7) with 6:20 minutes both on average and in the worst case, the distributor component with 3:31 minutes on average and 8:17 minutes in the worst case, and the pick and place unit with 3:22 minutes

on average and 31:59 minutes in the worst case. Note that computation timeouts only appear when generating test reports for selected subsystems (see Section 7.1.3). Consequently, the numbers indicate the durations between generating subsequent test reports. Then, part collisions took 3:15 minutes on average and 5:48 minutes in the worst case to disappear. The longest duration has been recorded for resolving the collision between the stamper basket (see Section 8.2.8) and the workpiece sensor of the distributor component installed at the stamp location (see Section 8.2.2). Note that this example has been discussed previously in Section 9.2.2. Afterwards, the non-determinisms caused by multiple behaviors writing the same output port (see Section 6.2.2) took 2:02 minutes to disappear. Note that the duration in fact represents the time it took to identify the semantic issue as a software bug in the simulation engine. Finally, the non-determinisms caused by multiple enabled transitions of the same executable (see Section 6.2.2) took 29 seconds to be disappear. Note that such non-determinisms can be resolved easily by correcting the guard expressions (see Section 5.1.2). Furthermore, note that not all such non-determinisms have been resolved. In particular, the non-determinism of the random workpiece scenario, which has been omitted in Chapter 5, has been added on purpose. One can conclude that the time until semantic issues disappear depends on a number of factors. In particular, the more local the semantic issue is, the faster it can be resolved (e.g. the part collisions and the non-determinisms caused by multiple enabled transitions). In contrast, the more global a semantic issue is, the harder it might be to resolve (e.g. the constraint violations of the pick and place unit or the distributor component).

9.4 Study validity

Finally, according to common practice in experimental research [SCC01], the internal and external validity of the study are discussed. Hereby, the internal validity (see Section 9.4.1) is concerned with the reliability of the conclusions that have been drawn from the data collected during the experiment (see Chapter 8) with respect to (1) the *method feasibility*, (2) the *model validity*, and (3) the *issue relevancy*. Instead, the external validity (see Section 9.4.2) is concerned with the degree to which those conclusions can be generalized from the experiment to the entire domain of cyber-physical manufacturing systems.

9.4.1 Internal validity

In the following, the internal validity of the conclusions drawn from the experiment is discussed with respect to the three main research questions: (1) The feasibility of the

test-driven method for the conceptual design of cyber-physical manufacturing systems, (2) the validity of the system model obtained during the experiment and the modeling technique in general, and (3) the relevancy of the syntactic and semantic quality issues in practical applications.

Method feasibility

To guarantee internal validity with respect to the question of *method feasibility*, (1) tool instrumentation was used for collecting data automatically during tool usage, (2) appropriate aggregations and visualizations of the collected data were developed, and (3) the results of these visualizations were interpreted. Note that a high degree of automation in the process was targeted to remove the bias of subjective interpretation. However, one can question whether the right data has been collected, whether the aggregations and visualizations are appropriate, and whether the interpretation of the visualizations is valid. For data collection only the most basic events were used, which can be obtained from tool usage. In particular, the events cover all possible changes to the system model as well as all possible outcomes of test execution. Then, the aggregations by component of the system model (see Section 9.1.1), feature of the modeling technique (see Section 9.1.2), and result of the test execution (see Section 9.1.3) were calculated fully automatically. Again, the aggregation can be derived unambiguously. The same holds for the visualization using time lines. Finally, the interpretation included mapping the time line information back to the phases and activities of the test-driven design method (see Chapter 3). In particular, the mapping to phases can be achieved unambiguously, because the phases do not share any model elements. However, some ambiguity can be found in the mapping to the activities of the phases, as the requirement specification (respectively formalization) and the process specification potentially share the monitor elements. However, this ambiguity can be foreclosed because the explicit formalization of requirements was omitted in the experiment to save effort.

Model validity

Then, to guarantee internal validity with respect to the model validity, the authors have had access to the SysML documentation [LFVH13] as well as the physical system at the Institute of Information and Automation Systems, Technical University of Munich. Again, the comparison of the system architectures is rather straight forward. In SysML, the architecture is specified in terms of block diagrams consisting of *blocks*, which correspond to the components introduced in Section 5.2.1, and *containment relationships*, which correspond to subcomponents in the proposed modeling technique. In contrast,

the interactions between the components are defined only implicitly as part of the textual descriptions and the behavioral diagrams in the SysML documentation, while in the developed system model interactions are defined explicitly using ports (see Section 5.2.2) and channels (see Section 5.2.3). The comparison of the system behaviors, on the other hand, is more difficult to achieve. For this purpose, the observable component locations and the reachable critical states during operation of the physical system and the system model were used. Hereby, the simplifications are known that have been made with respect to the physical behavior (e.g. omitting the weight of workpieces as well as acceleration and deceleration phases of workpiece motion). As stated previously, such simplifications are considered to be valid during conceptual design. Furthermore, such effects can be modeled with sufficiently large efforts. Also, critical states have been observed in system model, which can be observed in the physical system also (e.g. the effect of sensor and actuator signal delays on the correct positioning of the vacuum gripper).

Issue relevancy

Finally, to guarantee internal validity with respect to the issue relevancy, again tool instrumentation for data collection as well as various aggregations and visualizations of the collected data were used. Consequently, one might question the appropriateness of the collected data, the developed aggregations, and the employed visualizations. In contrast to the process feasibility, this time the appearance and disappearance events of syntactic issues as well as the appearance events of semantic issues during test executions were used. Furthermore, disappearance events of semantic issues were derived from subsequent test executions with changing outcomes. Note that the collected events and derived events are completely objective and unambiguous. However, also note that the derived disappearance events of semantic issues do not necessarily mean that the semantic issues have been resolved, but they might be occluded by other semantic issues. Furthermore, the semantic issues might have been resolved already earlier than uncovered by test execution, but not any later. Then, the aggregations by issue category (i.e. incompleteness or inconsistency and their subclasses or extrinsic and intrinsic and their subclasses) and duration until disappearance can be performed automatically and, hence, unambiguously as well. Finally, the comparison of the relative frequencies of appearances as well as the minimum, average, and maximum durations until disappearance provide good overall measures for how long certain issues are active. The activity of the quality issues determine an objective measure of relevance. Note, however, that in the perception of engineers certain issues might be more relevant than others (e.g. the inconsistency issues might be more relevant than the incompleteness issues). The subjective relevancy was not evaluated in this doctoral thesis.

9.4.2 External validity

In contrast, the external validity of the study is limited mainly due to the academic case (i.e. the pick and place unit). Originally, the case has been designed to resemble industrial plants closely [LFVH13]. Consequently, the case comprises a number of important features that can be found in industrial systems. Such features include the transportation, selective manipulation, and separation of different types of material within certain timing constraints. However, the case mainly lacks functional, structural, and behavioral complexity. In fact, the individual functions to be performed by the pick and place unit (i.e. processing white plastic, metallic, and black plastic material) can be separated rather easily and have limited influence on each other. It will be interesting to see how the test-driven approach performs on systems, where the different functions cannot be separated that easily. For example, one could think about a system where multiple materials can be processed in parallel to increase productivity. Such investigations are left to future research. Another deficiency of the study is that the target design existed prior to executing the study. Hence, the developed design is influenced by a priori knowledge. To circumvent this deficiency an experimental setup is required, where the participants are not aware of the target design. Then, however, evaluation of the model validity would have become more difficult as no baseline exists to compare the system model to.

9.5 Summary and outlook

This chapter discussed critically the results of applying the proposed approach to an industry-close example: The pick and place unit. In particular, the feasibility of the test-driven method for the conceptual design of cyber-physical manufacturing systems was evaluated (see Section 9.1). Then, the validity of the obtained model and the underlying modeling technique was analyzed (see Section 9.2). Subsequently, the relevancy of the individual quality issues was discussed (see Section 9.3). Finally, the validity of the entire study and the respective conclusions was considered (see Section 9.4). The following chapter summarizes the contributions of this doctoral thesis and point to future work in the area of conceptual design of cyber-physical manufacturing systems.

10 Conclusion

In the following, first the content of this doctoral thesis is summarized in Section 10.1. Then, an outlook on potential future work is provided in Section 10.2.

10.1 Summary

The summary first concentrates on the practical challenges explicated by management personnel of six German machine tool builders in Section 10.1.1. Then, the problems are revised that remain unsolved by related approaches and commercial tools in Section 10.1.2. Finally, the scientific contributions claimed by this doctoral thesis in order to fill the gaps are explained in Section 10.1.3.

10.1.1 Practical challenges

In several interviews with management personnel of six German machine tool builders four practical challenges have been identified, which served as the starting point for this doctoral thesis. First, the conceptual design of cyber-physical manufacturing systems is dominated by mechanical engineers, while electrical engineers and software engineers get on board in later phases only (see Section 1.3.1). Consequently, synergy potentials between the different engineering disciplines cannot be exploited. Then, as soon as the other engineering disciplines get involved, design decisions are not synchronized among the engineering disciplines sufficiently (see Section 1.3.2). Consequently, the individual engineering disciplines might work with out-to-date design knowledge and, hence, compatibility issues might arise. Furthermore, the design information created by the different engineering disciplines is evaluated only within the discipline boarders, but rarely with respect to the design information from other engineering disciplines (see Section 1.3.3). Consequently, design flaws spanning different engineering disciplines remain hidden and, hence, might have considerable effects on the design process. Finally, the design process itself is arranged typically sequentially starting with mechanical engineering, then electrical engineering, and finally software engineering (see Section 1.3.4). Consequently, complete system designs are available only late in the project and design flaws might cause costly design iterations.

10.1.2 Remaining problems

When looking at related work on conceptual design of cyber-physical manufacturing systems from both academia and industry, one can observe four remaining challenges, which are addressed by this doctoral thesis. First, the existing approaches typically cover only a limited part of the information relevant during conceptual design (see Section 2.2.1). Consequently, the individual engineering disciplines might not be able to express certain design knowledge regarding the customer requirements, the manufacturing processes, test cases, or the implementation details including the spatial extent and discipline-spanning behaviors. Then, even if many design details are covered, the approaches typically lack an integrated formalism describing the syntactic and semantic relations between the design elements (see Section 2.2.2). Consequently, the design documents might be ambiguous leading to misinterpretation and misunderstanding between team members and across engineering disciplines. Furthermore, the lack of an integrated formalism prevents the automated evaluation of the design information (see Section 2.2.3). Consequently, only parts of the design information can be evaluated or the evaluation has to be carried out manually, which is a costly process. And finally, the existing approaches with sufficient information coverage lack a practical methodology, which engages all engineering disciplines equally, delays assignment of responsibilities to specific engineering disciplines where possible, and fosters early verification and validation of system functions (see Section 2.2.4). Consequently, suboptimal and inappropriate solutions might be developed consuming unnecessary project budget.

10.1.3 Claimed contributions

To overcome the remaining problems and, hence, address the practical challenges, this doctoral thesis claims six contributions. First, test-driven [Bec02] and top-down, compositional [BS01] software development ideas and principles were adapted to the cyber-physical manufacturing system domain (see Chapter 3). Therefore, material, manufacturing process, and part specification activities were integrated, which are not relevant for pure software systems. Then, a modeling technique and integrated formalism was devised for capturing design knowledge as well as the underlying syntactic and semantic relations (see Chapter 5). In particular, the modeling technique and integrated formalism is based on an existing approach covering the spatial extent as well as spatial, energy, and data interactions between components [Hum11]. The approach is adapted and extended to enable the formal specification of requirements, manufacturing processes, and test cases in relation to the spatial extent, behavior, and interaction of components. Subsequently, a taxonomy of quality issues is developed, which is intended to be used

for the (possibly automated) evaluation of design information (see Chapter 6). For this purpose, the taxonomy covers syntactic and semantic aspects and provides formal definitions of the underlying design constraints. In the following, it was demonstrated how the modeling technique and the automated evaluation of quality issues can be implemented into a software tool (see Chapter 7). In particular, the software tool demonstrates how the variety of design information as well as the evaluation results can be integrated into a common user interface consisting of different views onto the design information with filters for the individual aspects of the modeling technique. After having devised the various parts of the proposed approach, the approach was applied to an industry-close example: The pick and place unit installed at the Institute for Information and Automation Systems, Technical University of Munich (see Chapter 8). The showcase already demonstrates how conceptual designs and design documents could look like when using the proposed approach. Furthermore, the showcase demonstrates the maturity of the prototypical tooling. Finally, the test-driven method was shown to be indeed feasible for the conceptual design of cyber-physical manufacturing systems, the model of the pick and place unit can be considered to be valid, and the quality issues prove to be relevant in practical applications (see Chapter 9). In particular, the analysis showed that iterative and incremental development is possible and that both syntactic and semantic quality issues can be discovered effectively. Consequently, partially complete system designs are available much earlier in the process, which enables early design validation. Furthermore, most syntactic issues indicated that the system model is incomplete, while the semantic issues indicated that the requirements and / or manufacturing processes were not implemented correctly.

10.2 Outlook

Finally, an outlook on potential future work in the field of conceptual design of cyber-physical manufacturing systems is provided. First, the question arises how efficient the test-driven design method really is (see Section 10.2.1). Then, one has to evaluate the practical suitability of the modeling technique and the integrated formalism (see Section 10.2.2). Furthermore, one might question whether all quality issues are covered by the taxonomy or not (see Section 10.2.3). And at last, one can be concerned with the usability of the prototypical tooling (see Section 10.2.4).

10.2.1 Method efficiency

Regarding the efficiency of the test-driven method two critical directions for future work are foreseen: (1) Method comparison and (2) design refinement.

Method comparison

First, one needs to compare the test-driven method to other design methods to get an objective measure of efficiency. Hereby, it is easier to compare different methods based on the same modeling technique because different modeling techniques might cover different design information. Furthermore, one might need to consider different situations such as greenfield development and design evolution as well as different application domains such as automation systems and machine tools.

Design refinement

Second, one needs to think about how to work with the design information after finishing the conceptual design phase. Note that preliminary thoughts on design refinement have already been described in [HRZ15a], which include an extension of the modeling technique, the systematic refinement of individual components, and model transformations to feed other engineering tools and tool chains. Alternatively, one might think about switching the engineering tools completely and feeding them manually.

10.2.2 Model suitability

Then, regarding the suitability of the modeling technique five critical directions for future work are foreseen: (1) Spatio-temporal logics, (2) advanced geometry representations, (3) continuous time dynamics, (4) flexible multi-body dynamics, and (5) computational fluid dynamics.

Spatio-temporal logics

First, spatio-temporal logics [MWZ03] could be used to formalize the requirements and, possibly, the manufacturing processes. Note that, currently, expressions (see Section 5.1.3) contained in properties (see Section 5.3.2) and monitors (see Section 5.3.3) are employed to accomplish this task. A suitable dialect of spatio-temporal logics might provide more concise means for formalization. However, today, spatio-temporal logics typically is used for querying movie databases [BVZ95] rather than the conceptual design of cyber-physical manufacturing systems.

Advanced geometry representations

Thereafter, the modeling technique should be extended to support the full set of constructive solid geometry (CSG) operations including the intersection and the difference

of volumes [RV77]. Note that, currently, the proposed modeling technique only supports the union of volumes (see Section 5.1.4), which simplifies the detection of collisions. However, the full set of CSG operations would allow one to model more complex geometries more easily. Alternatively, boundary representations (e.g. [Pie91]) could be used.

Continuous time dynamics

Then, it might be worthwhile to integrate facilities for modeling continuous-time dynamics. For example, one could use differential equations as provided by Modelica [Hau06b] or hybrid input-output automata [LSV03], which provide more intuitive graphical modeling means. Being able to express continuous-time dynamics, one could develop more realistic models. However, then, simulation and simulation-based testing of semantic execution constraints becomes more difficult to achieve.

Flexible multi-body dynamics

Subsequently, it might be interesting to integrate flexible multi-body dynamics [Sha97] into the approach. Consequently, more realistic motion behaviors could be described with less effort. Note that, currently, each motion has to be described explicitly independent of what causes the motion (e.g. an impact force or gravity). Furthermore, the deformation of bodies could be studied, which can have considerable effects on the system behavior [ZOM04] and, hence, on the correctness of the implementation.

Computational fluid dynamics

Finally, computational fluid dynamics [ADD+10] could be added to the approach. Consequently, one could describe more precisely the behavior of liquids and gases and their effect on the system behavior. For example, liquids and gases are used to cool workpieces and tools or to clean the working space during operation. Furthermore, liquids and gases can be found in hydraulic and pneumatic components, which have not been considered in detail in this doctoral thesis.

10.2.3 Issue completeness

Subsequently, regarding the completeness of the quality issues five critical directions for future work are foreseen: (1) natural language processing, (2) test coverage criteria, (3) test case generation, (4) semantic consistency, and (5) continuous integration.

Natural language processing

First, natural language processing techniques [Man99] could be used to analyze textural requirements as well as the name and description of different model elements. For example, one could prevent the use of synonyms to avoid confusion during communication between different stakeholders. Furthermore, one might indicate the use of passive voice, which might hinder the correct and unambiguous interpretation of natural language requirements [BK04].

Test coverage criteria

Then, the question arises which kind of test coverage criteria [ZHM97] could be used with the presented approach. Typically, test coverage for pure software systems concerns the possible sequences of code statements. Furthermore, test coverage can be defined over the possible sequences of states and transitions of state machines [Cho78]. However, the proposed modeling technique additionally includes spatial characteristics and manufacturing process specifications, which are not covered by the existing approaches.

Test case generation

At the same time, one might think about the generation of test cases from the formalized requirements and the manufacturing process specifications. For example, approaches exist to generate test cases using a model checker and the coverage criteria from the previous section [RH01]. However, here, again the question remains how such techniques can be adapted to the coverage of spatial configurations of the system. In particular, it might be difficult to find relevant spatial configurations among all possibilities. Furthermore, one has to think about the methodological integration.

Semantic consistency

Subsequently, the question arises how the semantic consistency among different requirements as well as among manufacturing process specifications and test cases can be ensured. Note that, currently, only the consistency between the implementation and the requirements as well as between the implementation and the manufacturing process specifications is checked using test cases. Note that requirements consistency has been addressed, for example, in [HJL96]. Again, one has to think about how to adapt the ideas to spatio-temporal semantics.

Continuous integration

At last, having devised a test-driven method for the conceptual design of cyber-physical manufacturing systems, one could think about extending the ideas to continuous integration [DMG07]. Consequently, semantic issues could be evaluated much more frequently and reported much earlier to the engineers. However, note that continuous integration only exists for pure software systems. Consequently, it is not obvious how the spatio-temporal semantics can be supported effectively.

10.2.4 Tool usability

Finally, regarding the usability of the prototypical tooling two critical directions for future work are foreseen: (1) Usability evaluation, and (2) usability improvement.

Usability evaluation

First, one needs to evaluate the usability of the prototypical tooling (see Chapter 7). During the experiment the prototypical tooling was observed to be mature enough to accomplish the individual design tasks. However, a couple of usability flaws have been observed as well. For example, currently adding and removing model elements is done via the explorer view (see Section 7.1.2), which is rather inconvenient for expressions and state machine elements.

Usability improvement

Then, one has to think about possible improvements to the user interface and test their implementation. In particular, the improvements should target the collaboration and communication of as well as the synchronization between different engineers working on the same project. Note that the synchronization comprises the changes made by each individual engineer to the model. Hereby, an agile approach with tight integration of user feedback is suggested.

Bibliography

[ACJT96] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *11th Annual IEEE Symposium on Logic in Computer Science*, LICS'96, pages 313–321, New Brunswick, NJ, USA, July 1996. IEEE.

[ADD+10] J. Anderson, E. Dick, G. Degrez, R. Grundmann, J. Degroote, and J. Vierendeels. *Computational fluid dynamics – An introduction.* Springer, Berlin, Heidelberg, Germany, 3rd edition, November 2010.

[AH14] D. Ascher and G. Hackenberg. Early estimation of multi-objective traffic flow. In *International Conference on Connected Vehicles and Expo*, ICCVE'14, pages 1056–1057, Vienna, Austria, November 2014. IEEE.

[AH15] D. Ascher and G. Hackenberg. Integrated transportation and power system modeling. In *International Conference on Connected Vehicles and Expo*, ICCVE'15, pages 379–384, Shenzhen, China, October 2015. IEEE.

[AH16] D. Ascher and G. Hackenberg. The transp-0 framework for integrated transportation and power system design. In *19th IEEE International Conference on Intelligent Transportation Systems*, ITSC'16, pages 945–952, Rio de Janeiro, Brazil, November 2016. IEEE.

[AH17] D. Ascher and G. Hackenberg. The passenger extension of the transp-0 system design framework. In *5th IEEE International Conference on Models and Technologies for Intelligent Transportation Systems*, MT-ITS'17, pages 256–261, Naples, Italy, June 2017. IEEE.

[Alf85] M. Alford. Srem at the age of eight; the distributed computing design system. *Computer*, 18(4):36–46, April 1985.

[And89] S. J. Andriole. *Storyboard Prototyping: A New Approach to User Requirements Analysis.* QED Information Sciences, Inc., London, UK, August 1989.

[Asc13] D. Ascher. A model-based approach for specification and validation of smart e-mobility control requirements. Bachelor's thesis, Fakultät für Informatik, Technische Universität München, Garching, Germany, October 2013.

[Asc15] D. Ascher. A distributed approach to approximate numerical solution of discrete-time optimal control problems. Master's thesis, Fakultät für Informatik, Technische Universität München, Garching, Germany, December 2015.

[AZ89] L. Alting and H. Zhang. Computer aided process planning: the state-of-the-art survey. *International Journal of Production Research*, 27(4):553–585, 1989.

[BAT97] P. Borst, H. Akkermans, and J. Top. Engineering ontologies. *International Journal of Human-Computer Studies*, 46(2–3):365–406, February 1997.

[BD93] A.-P. Bröhl and W. Dröschel. *Das V-Modell: Der Standard für die Softwareentwicklung mit Praxisleitfaden.* Oldenbourg Wissenschaftsverlag, Munich, Germany, 1993.

[Bec02] K. Beck. *Test-driven development: by example.* The Addison-Wesley Signature Series. Addison-Wesley, Boston, MA, USA, November 2002.

[Ber08] D. M. Berry. *Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs: 14th Monterey Workshop 2007. Monterey, CA, USA, September 2007. Revised Selected Papers*, volume 5320 of *Lecture Notes in Computer Science*, chapter Ambiguity in Natural Language Requirements Documents, pages 1–7. Springer, Berlin, Heidelberg, Germany, 2008.

[BGT05] S. Burmester, H. Giese, and M. Tichy. *Model Driven Architecture: European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26–27, 2003 and Linköping, Sweden, June 10–11, 2004. Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, chapter Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML, pages 47–61. Springer, Berlin, Heidelberg, Germany, 2005.

[BH10] J. Botaschanjan and B. Hummel. *Theoretical Aspects of Computing – ICTAC 2010: 7th International Colloquium, Natal, Rio Grande do Norte, Brazil, September 1–3, 2010. Proceedings*, volume 6255 of *Lecture Notes*

in Computer Science, chapter Material Flow Abstraction of Manufacturing Systems, pages 153–167. Springer, Natal, Rio Grande do Norte, Brazil, 2010.

[BHHL09] J. Botaschanjan, B. Hummel, T. Hensel, and A. Lindworsky. Integrated behavior models for factory automation systems. In *IEEE Conference on Emerging Technologies Factory Automation*, ETFA'09, pages 1–8. IEEE, September 2009.

[BHJP05] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson. *Formal Approaches to Software Testing: 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*, chapter Specifying and Generating Test Cases Using Observer Automata, pages 125–139. Springer, Berlin, Heidelberg, Germany, 2005.

[BJR+96] G. Booch, I. Jacobson, J. Rumbaugh, et al. The unified modeling language. *Unix Review*, 14(13):5, 1996.

[BK04] D. M. Berry and E. Kamsties. Ambiguity in requirements specification. In J. H. Doorn J. C. Sampaio do Prado Leite, editor, *Perspectives on software requirements*, volume 753 of *Springer International Series in Engineering and Computer Science*, pages 7–44. Springer, New York, NY, USA, 2004.

[BME+07] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and K. A. Houston. *Object oriented analysis and design with application.* Addision-Wesley Object Technology Series. Addison-Wesley, Boston, MA, USA, 3rd edition, April 2007.

[BN06] T. Bhat and N. Nagappan. Evaluating the efficacy of test-driven development: Industrial case studies. In *ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE'06, pages 356–363, Rio de Janeiro, Brazil, September 2006. ACM.

[BOA+11] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, H. Elmqvist, A. Junghanns, J. Mauß, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and C. Wolf, S. Clauß. The functional mockup interface for tool independent exchange of simulation models. In *8th International Modelica Conference*, number 63 in Linköping Electronic Conference Proceedings, pages 105–114, Dresden, Germany, March 2011. Linköping University Electronic Press.

[BR87] C. Beck and G. Roepstorff. Effects of phase space discretization on the long-time behavior of dynamical systems. *Physica D: Nonlinear Phenomena*, 25(1–3):173–180, March–April 1987.

[Bro01] T. R. Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering Management*, 48(3):292–306, August 2001.

[Bro07] M. Broy. *SOFSEM 2007: Theory and Practice of Computer Science: 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20–26, 2007. Proceedings*, volume 4362 of *Lecture Notes in Computer Science*, chapter Interaction and Realizability, pages 29–50. Springer, Berlin, Heidelberg, Germany, 2007.

[Bro10] M. Broy. A logical basis for component-oriented software and systems engineering. *The Computer Journal*, 53(10):1758–1782, December 2010.

[BS96] R. H. Bracewell and J. E. E. Sharpe. Functional descriptions used in computer support for qualitative scheme generation – schemebuilder. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing*, 10(4):333–345, September 1996.

[BS01] M. Broy and K. Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Monographs in Computer Science. Springer, New York, NY, USA, May 2001.

[BVZ95] A. Del Bimbo, E. Vicario, and D. Zingoni. Symbolic description and visual querying of image sequences using spatio-temporal logic. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):609–622, August 1995.

[CFGR02] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of xml documents with xpath expressions. *The VLDB Journal*, 11(4):354–379, December 2002.

[CGHJ14] A. Campetelli, M. Gleirscher, G. Hackenberg, and M. Junker. Konzepte und werkzeug-prototypen für die mechatronische modellierung von embedded systems. Project report, Fakultät für Informatik, Technische Universität München, Garching, Germany, 2014.

[CGI93] B. Chandrasekaran, A. K. Goel, and Y. Iwasaki. Functional representation as design rationale. *Computer*, 26(1):48–56, January 1993.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. Mit University Press Group, Cambridge, MA, USA, December 1999.

[CH15] A. Campetelli and G. Hackenberg. Performance analysis of adaptive runge-kutta methods in region of interest. In *2nd International IFIP Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems*, EITEC'15, Seattle, WA, USA, April 2015.

[CHJ13] A. Campetelli, G. Hackenberg, and M. Junker. Modelling of spes xt case studies with focus components. Project report, Fakultät für Informatik, Technische Universität München, Garching, Germany, 2013.

[Cho78] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, 4(3):178, May 1978.

[Clo60] R. W. Clough. The finite element method in plane stress analysis. In *2nd ASCE Conference on Electronic Computation*, pages 345–378, Pittsburgh, Pennsylvania, September 1960. A.S.C.E. Structural Division, American Society of Civil Engineers.

[Cox61] H. S. M. Coxeter. Introduction to geometry. *Journal of Philosophy*, 60(1):19–21, 1961.

[CRS+13] S. K. Chandrasegaran, K. Ramani, R. D. Sriram, I. Horvth, A. Bernard, R. F. Harik, and W. Gao. The evolution, challenges, and future of knowledge representation in product design systems. *Computer-Aided Design*, 45(2):204 – 228, February 2013.

[CT98] J. V. Camahan and D. L. Thurston. Trade-off modeling for product and manufacturing process design for the environment. *Journal of Industrial Ecology*, 2(1):79–92, January 1998.

[CW02] L.-K. Chan and M.-L. Wu. Quality function deployment: A literature review. *European Journal of Operational Research*, 143(3):463–497, December 2002.

[CWW05] T.-C. Chang, R. A. Wysk, and H.-P. Wang. *Computer-aided manufacturing*. Prentice Hall International Series in Industrial and Systems Engineering. Prentice Hall, Upper Saddle River, NJ, USA, 3rd edition, July 2005.

[Dav92] A. M. Davis. Operational prototyping: a new development approach. *IEEE Software*, 9(5):70–78, September 1992.

[DHT00] C. H. Damm, K. M. Hansen, and M. Thomsen. Tool support for cooperative object-oriented design: Gesture based modelling on an electronic whiteboard. In *SIGCHI Conference on Human Factors in Computing Systems*, CHI'00, pages 518–525, The Hague, The Netherlands, April 2000. ACM.

[DJK+99] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *21st International Conference on Software Engineering*, ICSE'99, pages 285–294, Los Angeles, CA, USA, May 1999. ACM.

[dMB08] L. de Moura and N. Bjørner. *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, chapter Z3: An Efficient SMT Solver, pages 337–340. Springer, Berlin, Heidelberg, Germany, 2008.

[DMG07] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. The Addision-Wesley Signature Series. Addison-Wesley, Boston, MA, USA, July 2007.

[DMRT79] A. M. Davis, T. J. Miller, E. Rhode, and B. J. Taylor. Plp: an automated tool for the processing of requirements. In *3rd IEEE Computer Society's International Computer Software and Applications Conference*, COMPSAC'79, pages 289–299, Chicago, IL, USA, November 1979. IEEE.

[DOJ+93] A. Davis, S. Overmyer, K. Jordan, J. Caruso, F. Dandashi, A. Dinh, G. Kincaid, G. Ledeboer, P. Reynolds, P. Sitaram, A. Ta, and M. Theofanos. Identifying and measuring quality in a software requirements specification. In *1st International Software Metrics Symposium*, METRICS'93, pages 141–152, Baltimore, MD, USA, May 1993. IEEE.

[EFH13] S. Eder, H. Femmer, and G. Hackenberg. ifedit ermöglicht innovative schnittstellenspezifikationen. *VDW Branchenreport*, March 2013.

[Ein14] S. Einwang. Entwurf, implementierung und demonstration eines 3d animationsframework für das explorationswerkzeug xtream. Bachelor's thesis, Fakulät für Informatik, Technische Universität München, Garching, Germany, December 2014.

[EKvB+08] M. S. Erden, H. Komoto, T. J. van Beek, V. D'Amelio, E. Echavarria, and T. Tomiyama. A review of function modeling: Approaches and applications. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 22(2):147–169, March 2008.

[ElM93] H. A. ElMaraghy. Evolution and future perspectives of capp. *CIRP Annals – Manufacturing Technology*, 42(2):739–751, 1993.

[EN16] H. ElMaraghy and A. Nassehi. *CIRP Encyclopedia of Production Engineering*, chapter Computer-Aided Process Planning, pages 266–271. Springer, Berlin, Heidelberg, Germany, February 2016.

[Est07] J. A. Estefan. Survey of model-based systems engineering (mbse) methodologies. Technical Report Rev. B, INCOSE, San Diego, CA, USA, May 2007.

[Fal90] B. Faltings. Qualitative kinematics in mechanisms. *Artificial Intelligence*, 44(1–2):89–119, July 1990.

[Fär14] J. Färber. Entwicklung eines visualsierungsframeworks für das verhalten intelligenter energiesysteme auf basis von xtream. Bachelor's thesis, Fakulät für Informatik, Technische Universität München, Garching, Germany, May 2014.

[FB02] P. Fritzson and P. Bunus. Modelica – a general object-oriented language for continuous and discrete-event system modeling and simulation. In *35th Annual Simulation Symposium*, ANSS'02, pages 365–380, San Deigo, CA, USA, April 2002. IEEE.

[FKV91] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Informal and formal requirements specification languages: bridging the gap. *IEEE Transactions on Software Engineering*, 17(5):454–466, May 1991.

[FMS14] S. Friedenthal, A. Moore, and R. Steiner. *A practical guide to SysML: the systems modeling language*. Mk/Omg Press. Morgan Kaufmann, Burlington, MA, USA, 3rd edition, October 2014.

[For84] K. D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24(1–3):85–168, December 1984.

[FvBK+91] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[GBC+02] X. Gonze, J.-M. Beuken, R. Caracas, F. Detraux, M. Fuchs, G.-M. Rignanese, L. Sindic, M. Verstraete, G. Zerah, F. Jollet, M. Torrent, A. Roy, M. Mikami, P. Ghosez, J.-Y. Raty, and D. C. Allan. First-principles computation of material properties: the abinit software project. *Computational Materials Science*, 25(3):478–492, November 2002.

[GDT06] T. Gutowski, J. Dahmus, and A. Thiriez. Electrical energy requirements for manufacturing processes. In *13th CIRP International Conference on Life Cycle Engineering*, volume 31 of *LCE'06*, pages 623–638, Lueven, Belgium, May 2006.

[Ger90] J. S. Gero. Design prototypes: a knowledge representation schema for design. *AI magazine*, 11(4):26, 1990.

[GF94] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. In *1st IEEE International Conference on Requirements Engineering*, RE'94, pages 94–101, Colorado Springs, CO, USA, August 1994. IEEE.

[Gon70] G. Gonenc. A method for the design of fault detection experiments. *IEEE Transactions on Computers*, 19(6):551–558, June 1970.

[Gro15] M. P. Groover. *Automation, production systems, and computer-integrated manufacturing*. Pearson Education, London, UK, 4th edition, March 2015.

[GTD13] J. Gausemeier, C. Tschirner, and R. Dumitrescu. Der weg zu intelligenten technischen systemen. *Industrie Management*, 29(1):49–52, January 2013.

[GW03] B. George and L. Williams. An initial investigation of test driven development in industry. In *ACM Symposium on Applied Computing*, SAC'03, pages 1135–1139, Melbourne, FL, USA, March 2003. ACM.

[Hac15] Georg Hackenberg. Energyfocus – modellbasierte entwicklungsmethode für die informations- und kommunikationstechnologie (ikt) intelligenter energiesysteme. Project report, Fakultät für Informatik, Technische Universität München, Garching, Germany, 2015.

[Hau06a] N. C. Haugen. An empirical study of using planning poker for user story estimation. In *Agile Conference*, AGILE'06, pages 9–34, Minneapolis, MN, USA, July 2006. IEEE.

[Hau06b] M. Hause. The sysml modelling language. In *5th European Systems Engineering Conference*, EuSEC'06, Edinburgh, UK, September 2006. INCOSE.

[HB08] B. Hummel and P. Braun. Towards an integrated system model for testing and verification of automation machines. In *International Workshop on Models in Software Engineering*, MiSE'08, pages 51–56, Leipzig, Germany, May 2008. ACM.

[HB12] G. Hackenberg and D. Bytschkow. Towards early emergent property understanding: Merging behavior space exploration and model-based software engineering. In *Extreme Modeling Workshop*, XM'12, pages 39–44, Innsbruck, Austria, October 2012. ACM.

[HC88] J. R. Hauser and D. Clausing. The house of quality. *Harvard business review*, 66(3), May 1988.

[HCL+14] G. Hackenberg, A. Campetelli, C. Legat, J. Mund, S. Teufl, and B. Vogel-Heuser. Formal technical process specification and verification for automated production systems. In D. Amyot, P. Fonseca i Casas, and G. Mussbacher, editors, *8th International Conference on System Analysis and Modeling: Models and Reusability*, SAM'14, pages 287–303, Valencia, Spain, September 2014. Springer.

[HGS+16] G. Hackenberg, M. Gleirscher, T. Stocker, C. Richter, and G. Reinhart. Macon: Consistent cross-disciplinary conception of manufacturing systems. In *8th IFAC Conference on Manufacturing Modelling, Management and Control*, MIM'16, pages 1175–1180, Troyes, France, June 2016. Elsevier.

[HHP02] M. Harman, R. M. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *4th Annual Conference on Genetic and Evolutionary Computation*, GECCO'02, pages 1351–1358, New York, NY, USA, July 2002. Morgan Kaufmann.

[HIKB12] G. Hackenberg, M. Irlbeck, V. Koutsoumpas, and D. Bytschkow. Applying formal software engineering techniques to smart grids. In *Interna-*

tional Workshop on Software Engineering Challenges for the Smart Grid, SE4SG'12, pages 50–56, Zurich, Switzerland, June 2012. IEEE.

[HIKB14] G. Hackenberg, M. Irlbeck, V. Koutsoumpas, and D. Bytschkow. A rapid prototyping approach for smart energy systems based on partial system models. In *IEEE International Computer Software and Applications Conference Workshops*, COMSACW'14, pages 596–601, Vasteras, Sweden, July 2014. IEEE.

[Hir16] A. Hirsch. *Werkzeugmaschinen: Anforderungen, Auslegung, Ausführungsbeispiele*. Springer Vieweg, Berlin, Heidelberg, Germany, 3rd edition, November 2016.

[HJL96] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering Methodology*, 5(3):231–261, July 1996.

[HM16] G. Hackenberg and J. Mund. Cyber-physical manufacturing systems development: A test-driven approach and exploratory case study. Technical Report TUM-I1664, Fakultät für Informatik, Technische Universität München, Garching, Germany, 2016.

[HPSvH03] I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From shiq and rdf to owl: the making of a web ontology language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):7–26, December 2003.

[HRZ13] G. Hackenberg, C. Richter, and M. F. Zäh. Durchgängig modellbasierte entwicklung von werkzeugmaschinen. *VDI-Z Integrierte Produktion*, 155(9):24–28, September 2013.

[HRZ14] G. Hackenberg, C. Richter, and M. F. Zäh. A multi-disciplinary modeling technique for requirements management in mechatronic systems engineering. In *2nd International Conference on System-Integrated Intelligence: Challenges for Product and Production Engineering*, SysInt'14, pages 5–16, Bremen, Germany, July 2014. Elsevier.

[HRZ15a] G. Hackenberg, C. Richter, and M. F. Zäh. From conception to refinement in mechatronics systems engineering. *International Journal of Materials, Mechanics and Manufacturing*, 4(1):66–73, February 2015.

[HRZ15b] G. Hackenberg, C. Richter, and M. F. Zäh. Imomesa – abschlussbericht. Project Report TUM-I1519, Fakultät für Informatik, Technische Universität München, Garching, Germany, 2015.

[HSM+02] J. Hirtz, R. B. Stone, D. A. McAdams, S. Szykman, and K. L. Wood. A functional basis for engineering design: Reconciling and evolving previous efforts. Technical Report 1447, National Institute of Standards and Technology, Gaithersburg, MD, USA, February 2002.

[Hum09] B. Hummel. A semantic model for computer-based spatio-temporal systems. In *16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ECBS'09, pages 156–165, San Francisco, CA, USA, April 2009. IEEE.

[Hum11] B. Hummel. *Integrated Behavior Modeling of Space-Intensive Mechatronic Systems.* Doctoral thesis, Fakultät für Informatik, Technische Universität München, Garching, Germany, January 2011.

[HW98] W. Hsu and I. M.Y. Woon. Current research in the conceptual design of mechanical products. *Computer-Aided Design*, 30(5):377–389, April 1998.

[IBHK13] M. Irlbeck, D. Bytschkow, G. Hackenberg, and V. Koutsoumpas. Towards a bottom-up development of reference architectures for smart energy systems. In *International Workshop on Software Engineering Challenges for the Smart Grid*, SE4SG'13, pages 9–16, San Francisco, CA, USA, May 2013. IEEE.

[IO05] M. G. Ilieva and Olga Ormandjieva. *Natural Language Processing and Information Systems: 10th International Conference on Applications of Natural Language to Information Systems, NLDB 2005, Alicante, Spain, June 15–17, 2005. Proceedings*, volume 3513 of *Lecture Notes in Computer Science*, chapter Automatic Transition of Natural Language Software Requirements Specification into Formal Presentation, pages 392–397. Springer, Berlin, Heidelberg, Germany, 2005.

[JP07] A. Jimeno and A. Puerta. State of the art of the virtual reality applied to design and manufacturing processes. *International Journal of Advanced Manufacturing Technology*, 33(9–10):866–874, July 2007.

[JSJ98] S. H. Jacobson, K. A. Sullivan, and A. W. Johnson. Discrete manufacturing process design optimization using computer simulation and generalized hill climbing algorithms. *Engineering Optimization*, 31(2):247–260, June 1998.

[JST81] A. Jameson, W. Schmidt, and W. Turkel. Numerical solutions of the euler equations by finite volume methods using runge-kutta time-stepping schemes. In *14th Fluid and Plasma Dynamics Conference*, page 1981, Palo Alto, CA, USA, June 1981. American Institute of Aeronautics and Astronautics.

[Kai15] L. Kaiser. Development of a model for the intelligent operation of a hydro power plant chain. Project thesis, Fakulät für Maschinenbau, Technische Universität München, Garching, Germany, 2015.

[Keu91] A. M. Keuneke. Device representation-the significance of functional knowledge. *IEEE Expert*, 6(2):22–25, April 1991.

[KKFM04] Y. Kitamura, M. Kashiwase, M. Fuse, and R. Mizoguchi. Deployment of an ontological framework of functional design knowledge. *Advanced Engineering Informatics*, 18(2):115–127, April 2004.

[KM04] Y. Kitamura and R. Mizoguchi. Ontology-based systematization of functional knowledge. *Journal of Engineering Design*, 15(4):327–351, March 2004.

[KS98] G. Kotonya and I. Sommerville. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Hoboken, NJ, USA, September 1998.

[KS03] A. L. Kapelevich and Y. V. Shekhtman. Direct gear design: Bending stress minimization. *Gear Technology*, 20(5):44–47, September/October 2003.

[KT04] M. Krichen and S. Tripakis. *Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1–3, 2004. Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, chapter Black-Box Conformance Testing for Real-Time Systems, pages 109–126. Springer, Berlin, Heidelberg, Germany, 2004.

[Kul08] B. M. Kulfan. Universal parametric geometry representation method. *Journal of Aircraft*, 45(1):142–158, January/February 2008.

[LB02] B.-S. Lee and B. R. Bryant. Automated conversion from requirements documentation to an object-oriented formal specification language. In *ACM Symposium on Applied Computing*, SAC'02, pages 932–936, Madrid, Spain, March 2002. ACM.

[LDK03] B. Lauwers, P. Dejonghe, and J.P. Kruth. Optimal and collision free tool posture in five-axis machining through the tight integration of tool path generation and machine simulation. *Computer-Aided Design*, 35(5):421–432, April 2003.

[LFB96] J. Lin, M. S. Fox, and T. Bilgic. A requirement ontology for engineering design. *Concurrent Engineering*, 4(3):279–291, September 1996.

[LFVH13] C. Legat, J. Folmer, and B. Vogel-Heuser. Evolution in industrial plant automation: A case study. In *39th Annual Conference of the IEEE Industrial Electronics Society*, IECON'13, pages 4386–4391, Vienna, Austria, November 2013. IEEE.

[LMC+14] C. Legat, J. Mund, A. Campetelli, G. Hackenberg, J. Folmer, D. Schütz, M. Broy, and B. Vogel-Heuser. Interface behavior modeling for automatic verification of industrial automation systems' functional conformance. *at – Automatisierungstechnik*, 62(11):815–825, November 2014.

[LP04] V.-C. Liang and C. J. J. Paredis. A port ontology for conceptual design of systems. *Journal of Computing and Information Science in Engineering*, 4(3):206–217, September 2004.

[LSS94] O. I. Lindland, G. Sindre, and A. Solvberg. Understanding quality in conceptual modeling. *IEEE Software*, 11(2):42–49, March 1994.

[LSV03] N. Lynch, R. Segala, and F. Vaandrager. Hybrid i/o automata. *Information and Computation*, 185(1):105–157, August 2003.

[LT89] N. Lynch and M. Tuttle. An introduction to input/output automate. *CWI quarterly*, 2(3):219–246, September 1989.

[Man99] C. D. Manning. *Foundations of statistical natural language processing*. Mit University Press Group, Cambridge, MA, USA, May 1999.

[MDN09] P. Mohagheghi, V. Dehlen, and T. Neple. Definitions and approaches to model quality in model-based software development – a review of literature. *Information and Software Technology*, 51(12):1646–1669, December 2009.

[ME98] T. D. Miller and P. Elgard. Defining modules, modularity and modularization. In *13th IPS Research Seminar on Design for Integration in Manufacturing*, IPS'98, Fuglsø, Denmark, April 1998.

[MEO98] S. E. Mattsson, H. Elmqvist, and M. Otter. Physical system modeling with modelica. *Control Engineering Practice*, 6(4):501–510, April 1998.

[MLK12] T. Mayerhofer, P. Langer, and G. Kappel. A runtime model for fuml. In *7th Workshop on Models@Run.Time*, MRT'12, pages 53–58, Innsbruck, Austria, October 2012. ACM.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *38th Annual Design Automation Conference*, DAC'01, pages 530–535, Las Vegas, Nevada, USA, June 2001. ACM.

[Mon03] P. Monk. *Finite element methods for Maxwell's equations*. Numerical Analysis and Scientific Computation Series. Oxford University Press, Oxford, UK, April 2003.

[Mul16] T. Mulenko. Advanced experiment data collection and analysis for the macon approach. Master's thesis, Fakulät für Informatik, Technische Universität München, Garching, Germany, November 2016.

[MW03] E. M. Maximilien and L. Williams. Assessing test-driven development at ibm. In *25th International Conference on Software Engineering*, ICSE'03, pages 564–569, Portland, OR, USA, May 2003. IEEE.

[MWZ03] S. Merz, M. Wirsing, and J. Zappe. *Fundamental Approaches to Software Engineering: 6th International Conference, FASE 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings*, volume 2621 of *Lecture Notes in Computer Science*, chapter A Spatio-Temporal Logic for the Specification and Refinement of Mobile Systems, pages 87–101. Springer, Berling, Heidelberg, Germany, 2003.

[NMBW08] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3):289–302, February 2008.

[OSSJ09] L. Olsen, F. F. Samavati, M. C. Sousa, and J. A. Jorge. Sketch-based modeling: A survey. *Computers & Graphics*, 33(1):85–103, February 2009.

[Pay60] H. M. Paynter. *Analysis and design of engineering systems.* Mit University Press Group, Cambridge, MA, USA, June 1960.

[PBFG06] G. Pahl, W. Beitz, J. Feldhusen, and K.-H. Grote. *Engineering design: a systematic approach.* Springer, London, UK, 3rd edition, December 2006.

[PHO+01] S. G. Psakhie, Y. Horie, G. P. Ostermeyer, S. Y. Korostelev, A. Y. Smolin, E. V. Shilko, A. I. Dmitriev, S. Blatnik, M. Špegel, and S. Zavšek. Movable cellular automata method for simulating materials with mesostructure. *Theoretical and Applied Fracture Mechanics*, 37(1–3):311–334, December 2001.

[Pie91] L. Piegl. On nurbs: a survey. *IEEE Computer Graphics and Applications*, 11(1):55–71, January 1991.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[RBG14] G. Rehage, F. Bauer, and J. Gausemeier. Specification technique for the consistent description of manufacturing operations and resources. In M. F. Zäh, editor, *Enabling Manufacturing Competitiveness and Economic Sustainability: Proceedings of the 5th International Conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV 2013), Munich, Germany, October 6th–9th, 2013*, pages 47–53. Springer, Berlin, Heidelberg, Germany, 2014.

[RC86] A. A. G. Requicha and S. Chan. Representation of geometric features, tolerances, and attributes in solid modelers based on constructive geometry. *IEEE Journal on Robotics and Automation*, 2(3):156–166, September 1986.

[Ren82] T. Rentsch. Object oriented programming. *ACM Sigplan Notices*, 17(9):51–57, September 1982.

[Req80] A. A. G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, December 1980.

[RH01] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ECBS'01, pages 83–91, Washington, DC, USA, April 2001. IEEE, IEEE.

[RHAS00] W. C. Regli, X. Hu, M. Atwood, and W. Sun. A survey of design rationale systems: Approaches, representation, capture and retrieval. *Engineering with Computers*, 16(3–4):209–235, December 2000.

[RHSR15] C. Richter, G. Hackenberg, P. Stich, and G. Reinhart. Modellbasierte konzeption von benutzerschnittstellen im entwicklungsprozess von mechatronischen systemen. In *Tag des Systems Engineering: Verteiltes Arbeiten mit ganzheitlicher Kontrolle*, TdSE'15, pages 81–90, Herzogenaurach, Germany, October 2015. Carl Hanser Verlag.

[RHZ13] C. Richter, G. Hackenberg, and M. F. Zäh. Modellbasierte entwicklungsmethode für modulare maschinen und anlagen. *Zeitschrift für wirtschaftlichen Fabrikbetrieb (ZWF)*, 108(11):818–822, November 2013.

[RHZ14a] C. Richter, G. Hackenberg, and M. F. Zäh. Interdisziplinäre funktionsmodellierung für die generierung von robustem steuerungscode für fehlertolerantes systemverhalten. In *15. Branchentreff der Mess- und Automatisierungstechnik: Smart X – Powered by Automation*, AUTOMATION '14, pages 107–128, Düsseldorf, Germany, July 2014. VDI-Verlag.

[RHZ14b] C. Richter, G. Hackenberg, and M. F. Zäh. Mit interdisziplinärer modellierungstechnik mechatronische systeme entwickeln. *VDW Branchenreport*, May 2014.

[RHZR15] C. Richter, G. Hackenberg, M. F. Zäh, and G. Reinhart. Integrated requirements and systems modeling in the mechatronic development process. In *International Conference on Developments of E-Systems Engineering*, DeSE'15, pages 324–331, Dubai, UAE, December 2015.

[RJB10] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Pearson Education, London, UK, 2nd edition, April 2010.

[Roa76] P. J. Roache. *Computational fluid dynamics*. Hermosa publishers, Socorro, NM, USA, revised edition, June 1976.

[Rob06] C.M. Roberts. Radio frequency identification (rfid). *Computers & Security*, 25(1):18–26, February 2006.

[RRE91] J. Rumbaugh, J. Rumbaugh, and F. Eddy. *Object-oriented modeling and design*. Prentice Hall, Upper Saddle River, NJ, USA, December 1991.

[RV77] A. A. G. Requicha and H. B. Voelcker. Constructive solid geometry. *CADline*, 36(3):31–33, November 1977.

[RV83] A. A. G. Requicha and H. B. Voelcker. Solid modeling: current status and research directions. *IEEE Computer Graphics and Applications*, 3(7):25–37, October 1983.

[RV85] A. A. G. Requicha and H. B. Voelcker. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proceedings of the IEEE*, 73(1):30–44, January 1985.

[RW99] G. Reinhart and M. Weissenberger. Multibody simulation of machine tools as mechatronic systems for optimization of motion dynamics in the design process. In *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, AIM'99, pages 605–610, Atlanta, GA, USA, September 1999. IEEE.

[RW07] G. Reinhart and G. Wünsch. Economic application of virtual commissioning to mechatronic production systems. *Journal of Production Engineering - Research and Development*, 1(4):371–379, December 2007.

[RZH14] C. Richter, M. F. Zäh, and G. Hackenberg. Modellbasierte entwicklungsmethodik hilft mechatronische systeme zielgerichtet zu entwickeln. *VDW Branchenreport*, December 2014.

[SB02] K. Schwaber and M. Beedle. *Agile software development with Scrum*. Prentice Hall, Upper Saddle River, NJ, USA, February 2002.

[SCC01] W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and quasi-experimental designs for generalized causal inference*. Cengage Learning, Boston, MA, USA, January 2001.

[Sch97] W. Schiehlen. Multibody system dynamics: Roots and perspectives. *Multibody System Dynamics*, 1(2):149–188, June 1997.

[SCL98] N. P. Suh, D. S. Cochran, and P. C. Lima. Manufacturing system design. *CIRP Annals - Manufacturing Technology*, 47(2):627 – 639, 1998.

[SG00] E. Schweikardt and M. D. Gross. Digital clay: deriving digital models from freehand sketches. *Automation in Construction*, 9(1):107–115, January 2000.

[Sha97] A. A. Shabana. Flexible multibody dynamics: Review of past and recent developments. *Multibody System Dynamics*, 1(2):189–222, June 1997.

[SPLK01] R. Sinha, C. J. J. Paredis, V.-C. Liang, and P. K. Khosla. Modeling and simulation methods for design of engineering systems. *Journal of computing and information science in engineering*, 1(1):84–91, November 2001.

[Ste81] D. V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management*, 28(3):71–74, August 1981.

[Sto15] T. Stocker. Implementierung und evaluation des entwicklungsprozesses "imomesa" für den maschinen- und anlagenbau. Master's thesis, Fakulät für Informatik, Technische Universität München, Garching, Germany, September 2015.

[Suh95] N. P. Suh. Axiomatic design of mechanical systems. *Journal of Mechanical Design*, 117(B):2–10, June 1995.

[Suh98] N. Suh. Axiomatic design theory for systems. *Research in Engineering Design*, 10(4):189–209, December 1998.

[TH15] S. Teufl and G. Hackenberg. Efficient impact analysis of changes in the requirements of manufacturing automation systems. In *15th IFAC Symposium on Information Control Problems in Manufacturing*, INCOM'15, pages 1482–1489, Ottawa, Canada, May 2015. IFAC.

[Thr05] K. Thramboulidis. Model-integrated mechatronics - toward a new paradigm in the development of manufacturing systems. *IEEE Transactions on Industrial Informatics*, 1(1):54–61, February 2005.

[TMR13] S. Teufl, D. Mou, and D. Ratiu. Mira: A tooling-framework to experiment with model-based requirements engineering. In *21st IEEE International Requirements Engineering Conference*, RE'13, pages 330–331, Rio de Janeiro, Brazil, July 2013. IEEE.

[Tre99] J. Tretmans. *International Conference on Concurrency Theory: CONCUR 1999: CONCUR99 Concurrency Theory pp 46-65*, volume 1664 of *Lecture Notes in Computer Science*, chapter Testing Concurrent Systems: A Formal Approach, pages 46–65. Springer, Berlin, Heidelberg, Germany, 1999.

[UIY+96] Y. Umeda, M. Ishii, M. Yoshioka, Y. Shimomura, and T. Tomiyama. Supporting conceptual design based on the function-behavior-state modeler. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing*, 10(4):275–288, September 1996.

[Ulr94] K. Ulrich. *Management of Design: Engineering and Management Perspectives*, chapter Fundamentals of Product Modularity, pages 219–231. Springer, Dordrecht, The Netherlands, 1994.

[UTTY90] Y. Umeda, H. Takeda, T. Tomiyama, and H. Yoshikawa. Function, behaviour, and structure. *Applications of artificial intelligence in engineering V*, 1:177–194, July 1990.

[vBET10] T. J. van Beek, M. S. Erden, and T. Tomiyama. Modular design of mechatronic systems with function modeling. *Mechatronics*, 20(8):850–863, December 2010.

[VEH+14] A. Vogelsang, S. Eder, G.. Hackenberg, M. Junker, and S. Teufl. Supporting concurrent development of requirements and architecture – a model-based approach. In *International Conference on Model-Driven Engineering and Software Development*, MODELSWARD'14, pages 587–595, Lisbon, Portugal, January 2014. IEEE.

[VHLFF14] B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann. Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit. Technical report, Fakultät für Maschinenbau, Technische Universität München, Garching, Germany, 2014.

[VN92] N. Viswanadham and Y. Narahari. *Performance modeling of automated manufacturing systems*. Prentice Hall Information and System Sciences Series. Prentice Hall, Upper Saddle River, NJ, USA, February 1992.

[vON99] D. von Oheimb and T. Nipkow. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, chapter Machine-Checking the Java Specification: Proving Type-Safety, pages 119–156. Springer, Berlin, Heidelberg, Germany, 1999.

[WK99] J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml*. Addison-Wesley Object Technology Series. Addison-Wesley, Boston, MA, USA, March 1999.

[WMV03] L. Williams, E. M. Maximilien, and M. Vouk. Test-driven development as a defect-reduction practice. In *14th International Symposium on Software Reliability Engineering*, ISSRE'03, pages 34–45, Denver, CO, USA, November 2003. IEEE.

[WNST06] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety prooffor multiple inheritance in c++. *ACM SIGPLAN Notices*, 41(10):345–362, October 2006.

[WSX+02] L. Wang, W. Shen, H. Xie, J. Neelamkavil, and A. Pardasani. Collaborative conceptual design – state of the art and future trends. *Computer-Aided Design*, 34(13):981–996, November 2002.

[WW10] H.-J. Warnecke and E. Westkämper. *Einführung in die Fertigungstechnik*. Vieweg + Teubner, Wiesbaden, Germany, 8th edition, July 2010.

[ZHM97] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

[ZOM04] M. F. Zäh, T. Oertli, and J. Milberg. Finite element modelling of ball screw feed drive systems. *CIRP Annals - Manufacturing Technology*, 53(1):289–292, 2004.

[ZRH13a] M. F. Zäh, C. Richter, and G. Hackenberg. Herausforderungen im mechatronischen entwicklungsprozess: Anforderungsanalyse bei ausgewählten werkzeugmaschinenherstellern. *Newsletter der Wissenschaftlichen Gesellschaft für Produktentwicklung (WiGeP)*, 1:4–5, June 2013.

[ZRH13b] M. F. Zäh, C. Richter, and G. Hackenberg. Integrierte modellbasierte entwicklung mechatronischer systeme macht fortschritte. *VDW-Branchenreport*, June 2013.

[ZS07] M. F. Zäh and D. Siedl. A new method for simulation of machining performance by integrating finite element and multi-body simulation for machine tools. *CIRP Annals - Manufacturing Technology*, 56(1):383–386, 2007.