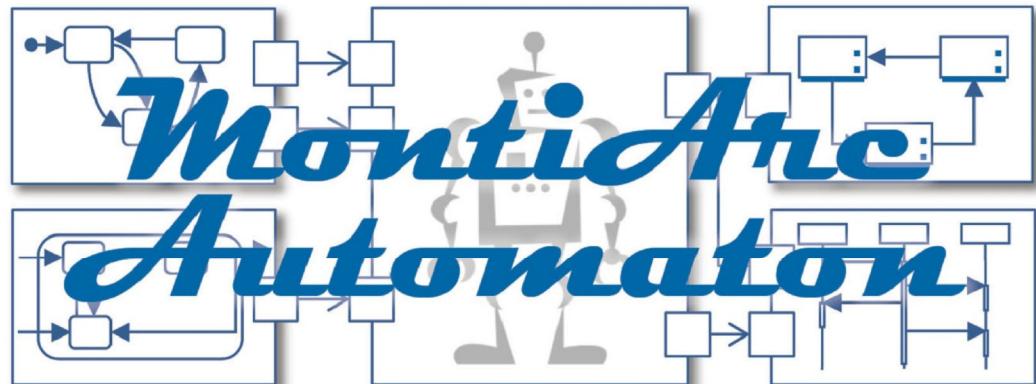


Andreas Wortmann

An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling



Aachener Informatik-Berichte,
Software Engineering

Hrsg: Prof. Dr. rer. nat. Bernhard Rümpe

Band 25

An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

**Diplom Informatiker Diplom-Wirtschaftsinformatiker
Andreas Wortmann
aus Joinville, Brasilien**

Berichter: Universitätsprofessor Dr. Bernhard Rümpe
Professor Benoit Combemale, Ph. D.

Tag der mündlichen Prüfung: 12. Juli 2016

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.



[Wor16] A. Wortmann:
An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling.
Shaker Verlag, ISBN 978-3-8440-4724-0. Aachener Informatik-Berichte, Software Engineering, Band 25. 2016.
www.se-rwth.de/publications/

*I was born not knowing
and have had only a little time
to change that here and there.*

Richard P. Feynman

Abstract

Efficient software engineering for complex systems requires abstraction, expertise from multiple domains, separation of concerns, and reuse. Domain experts are rarely software engineers and should be enabled to formulate solutions using their domain's vocabulary instead of general-purpose programming languages (GPLs). The successful integration of domain-specific languages (DSLs) into a software system requires a separation of concerns between domain issues and integration issues while retaining a loose enough coupling to support reusing a DSL in different contexts.

Component-based software engineering (CBSE) aims to increase software reuse and separation of concerns by encapsulating functionalities in components. This enables domain experts to develop solutions separated from integration concerns. Usually components are artifacts of GPLs, which gives rise to accidental complexities [FR07] and ties these to specific target platforms.

Model-driven engineering (MDE) abstracts from programming by lifting models to primary development artifacts. Models can be more abstract and better comprehensible by using domain vocabulary instead of a GPL. Furthermore, they can be platform-independent and translated into GPLs for different target platforms.

Component & connector (C&C) architecture description languages (ADLs) combine CBSE and MDE to enable composition of software architectures from component models. Such models define stable interfaces required to separate domain concerns from integration concerns. They can also employ the most appropriate DSLs to describe component behavior and support translation into GPL artifacts specific to different target platforms. Current research in MDE with ADLs focuses on structural modeling and requires component behavior either in terms of GPL artifacts or fixed component behavior languages. The former gives rise to accidental complexities, the latter demands that domain experts learn modeling languages foreign to their domain.

This thesis presents concepts for engineering complex software systems with exchangeable component behavior languages that enable contribution of domains experts using the most appropriate DSLs. The concepts are realized in a software architecture modeling infrastructure that comprises multiple modeling languages to develop applications based on C&C software architectures with exchangeable component behavior languages. It supports model-to-model transformations from platform-independent to platform-specific software architectures and compositional code generation. With this, it enables domain experts to (re-)use the most appropriate component behavior languages and facilitates composition of domain solutions through encapsulation in components.

It also enables reusing a single platform-independent software architecture with multiple platforms. To this effect, it combines results from software language engineering, model transformations, and code generator development to C&C ADLs.

The main contributions of this thesis are:

- Concepts to integrate domain-specific languages into component & connector software architectures to reduce accidental complexities, separate concerns, and facilitate their reuse.
- Methodical guidance to transform platform-independent into platform-specific architectures minor effort to increase reuse of components and architectures.
- Concepts of reusable compositional code generators for specific system aspects.
- A family of modeling languages to support architecture development with exchangeable behavior DSLs.
- A model-driven infrastructure, based on an extensible component & connector architecture description language that realizes these concepts.
- An evaluation of presented concepts in multiple contexts.

Employing these methodologies facilitates engineering of complex software systems by abstracting from programming issues, separating concerns, and reusing components, domain-specific languages, as well as code generators.

Kurzfassung

Die effiziente Softwareentwicklung komplexer System bedarf hoher Abstraktion, der Beteiligung von Experten aus verschiedenen Domänen, der Trennung von Belangen und eines hohen Grades an Wiederverwendung. Domänenexperten sind selten Softwareexperten und sollten daher befähigt werden Lösungen im Vokabular ihrer Domänen zu entwickeln. Um dies zu erreichen wurden in der modellgetriebenen Softwareentwicklung eine Vielzahl von domänenspezifischen Sprachen entwickelt. Die erfolgreiche Integration domänenspezifischer Sprachen in Softwaresysteme bedarf einer angemessenen Trennung von Domänen- und Integrationsbelangen, wobei die Kopplung dieser Sprachen lose genug sein muss um deren Wiederverwendung in anderen Kontexten zu ermöglichen.

Komponentenbasierte Softwareentwicklung versucht die Wiederverwendung von Software Kapselung von Funktionalitäten in Komponenten zu erhöhen. Dies ermöglicht Domänenexperten Lösungen unabhängig von Integrationsbelangen zu entwickeln. In komponentenbasierter Softwareentwicklung werden Komponenten üblicherweise durch Allzweck-Programmiersprachen beschrieben. Dies führt zu “unbeabsichtigten Komplexitäten” [FR07], welche darin bestehen Programmierdetails anstelle von Domänenproblemen zu lösen und führt dazu, dass die Lösungen nur zu bestimmten Zielplattformen kompatibel sind.

Modellgetriebene Softwareentwicklung abstrahiert von Programmierdetails durch die Verwendung von Modellen als primäre Entwicklungsaartefakte. Diesen können abstrakter und, durch Verwendung von Domänenvokabular, besser verständlich sein. Weiterhin können sie plattformunabhängig sein und durch Übersetzung in mehrere Allzweck-Programmiersprachen mit mehreren Plattformen wiederverwendet werden.

Komponenten und Konnektor Architekturbeschreibungssprachen kombinieren komponentenbasierte Softwareentwicklung mit modellgetriebener Softwareentwicklung zur Komposition von Softwarearchitekturen aus Komponentenmodellen. Diese Modelle verfügen über stabile Schnittstellen zur Trennung von Belangen, können Komponentenverhalten in angemessen domänenspezifischen Sprachen ausdrücken und ermöglichen eine automatisierte Übersetzung in plattformspezifische Artefakte. Gegenwärtige Forschung in der modellgetriebenen Entwicklung mit Architekturbeschreibungssprachen untersucht strukturelle Systemaspekte und erwartet Komponentenverhalten entweder in Form von Artefakten von Allzweck-Programmiersprachen oder in Form apriori festgelegter Modellierungssprachen. Ersteres führt zu unbeabsichtigten Komplexitäten, zweites erfordert dass Domänenexperten domänenfremde Sprachen lernen.

Diese Dissertation präsentiert Konzepte für die Entwicklung komplexer Softwaresysteme mit austauschbaren Komponentenverhaltenssprachen. Diese Konzepte sind in einer Infrastruktur zur Modellierung von Software-Architekturen realisiert welche mehrere Modellierungssprachen umfasst. Sie unterstützt die agile Einbettung angemessener Verhaltenssprachen, Modell-zu-Modell Transformationen von plattformunabhängigen zu plattformspezifischen Softwarearchitekturen, und kompositionale Code Generatoren. Dies ermöglicht Domänenexperten die angemessensten Komponentenverhaltenssprachen zu verwenden und eine einzige, plattformunabhängige, Softwarearchitektur mit verschiedenen Plattformen wieder zu verwenden. Um dies zu erreichen, kombiniert diese Infrastruktur Erkenntnisse aus der Entwicklung von Software-Sprachen, Modell-zu-Modell Transformationen, und aus der Code Generator Entwicklung mit Komponenten- und Konnektor Architekturbeschreibungssprachen.

Die wichtigsten Beiträge dieser Arbeit sind somit:

- Konzepte für die die Integration domänenpezifischer Sprachen in Komponenten- und Konnektor Softwarearchitekturen zur Reduktion unbeabsichtigter Komplexitäten, Trennung von Belangen und Erleichterung von deren Wiederverwendung.
- Eine Methodik zur Transformation plattformunabhängiger Softwarearchitekturen in plattformabhängige Softwarearchitekturen zur Erhöhung der Wiederverwendbarkeit von Komponenten und Architekturen.
- Ein Konzept zur Wiederverwendung kompositionaler Code Generatoren für bestimmte Systemaspekte.
- Eine Familie von Modellierungssprachen für die Architekturmödellierung mit austauschbaren Verhaltenssprachen.
- Eine modellgetriebene Werkzeugkette die diese Konzepte realisiert.
- Eine Evaluierung vorgestellter Konzepte in verschiedenen Kontexten.

Die Anwendung dieser Methodiken erleichtert die Entwicklung komplexer Softwaresysteme durch Abstraktion von Programmierungsdetails, durch eine gründliche Trennung von Belangen und durch Wiederverwendung von Komponenten, Architekturen, domänenpezifischen Sprachen und Code Generatoren.

Danksagung

Während meiner Promotion wurde ich von vielen Menschen unterstützt die damit zu dem Erfolg dieser Dissertation beigetragen haben und denen ich dafür sehr dankbar bin.

Größter Dank gebührt meinem Doktorvater Prof. Dr. Bernhard Rumpe, welcher diese Dissertation durch Unterstützung meiner Promotion erst möglich gemacht hat. Neben der Leitung einer großartigen Arbeitsgruppe haben viele fruchtbare Diskussionen mit ihm und seine Ratschläge diese Dissertation und meine wissenschaftlichen Tätigkeiten mitgeformt.

Ich danke außerdem Prof. Benoit Combemale, Ph.D., dem Zweitgutachter dieser Arbeit, für die sehr gute Zusammenarbeit und seine Unterstützung meiner Promotion. Mein Dank gebührt weiterhin Prof. Dr. Joost-Pieter Katoen dafür mein Promotionskomitee zu leiten und Prof. Dr. Ulrik Schroeder dafür in diesem mitzuarbeiten.

Außerdem danke ich den wundervollen Kollegen und Freunden am Lehrstuhl für Software Engineering der RWTH Aachen, welche die letzten fünf Jahre zu einer spannenden Zeit gemacht haben. Ohne das hervorragende Arbeitsklima, die fruchtbaren Anregungen und Diskussionen, und die gemeinsamen Arbeit an akademischen und organisatorischen Herausforderungen wäre diese Zeit kaum derart interessant gewesen. Besonders dankbar bin ich Dr. Jan Oliver Ringert, dessen Motivation, Unterstützung und Bereitschaft zu intensiven Diskussionen diese Dissertation stark beeinflusst haben. Besonderer Dank gilt auch Markus Look, welcher mich in vielen Dingen unterstützte und immer für aufschlussreiche Diskussionen da war, so wie Dr. Arne Haber, der geduldig für viele Fragen und Ideen zur Verfügung stand, und Andreas Horst, ohne dessen Hilfe manche Herausforderungen ungelöst wären. Weiterhin bedanke ich mich bei Robert Heim, dessen Unterstützung in verschiedenen Tätigkeiten den Endspurt der Promotion erleichtert hat.

Außerdem bedanke ich mich bei Prof. Dr. Christian Berger und Prof. Dr. Ulrike Thomas, welche mir die Möglichkeit gaben mich mit weiteren spannenden Forschungsfragen zu befassen und an Ihrer Erfahrung teilhaben ließen. Ich danke auch Dr. Stefan Schiffer, Dr. Martin Schindler und Prof. Dr. Christian Schlegel, deren Erfahrungen auf dem Weg zur Dissertation und darüber hinaus sehr hilfreich waren. Weiterer Dank gebührt Kai Adam, Marita Breuer, Arvid Butting, Angelika Fleck, Timo Greifenberg, Sylvia Gunder, Lars Hermerschmidt, Dr. Christoph Herrmann, Gabriele Heuschen, Katrin Hölldobler, Steffi Kaiser, Oliver Kautz, Dennis Kirch, Carsten Kolassa, Evgeny Kusmenko, Thomas Kurpick, Achim Lindt, Klaus Müller, Antonio Navarro Pérez, Jerome Pfeiffer, Prof. Dr. Manfred Nagl, Pedram Mir Seyed Nazari, Dr. Claas Pinkernell, Dimitri Plotnikov, Deni Raco, Holger Rendel, Dirk Reiss, Alexander Roth, Christoph

Schulze, Galina Volkova, Michael von Wenckstern, Dr. Ingo Weisemöller und Dr. Steven Völkel ohne deren Unterstützung in den vielen Herausforderungen dieser Promotion diese nicht derart möglich gewesen wäre. Nicht zuletzt danke ich meiner Familie, meiner Partnerin und meinen Freunden für ihre Unterstützung während dieser Zeit und für ihr Verständnis, wenn ich mich für die Arbeit rar gemacht habe. Besonders danke ich meinen Eltern für Ihre durchgängige Unterstützung aller Schritte die zu dieser Arbeit geführt haben.

Trademarks appear throughout this thesis without any trademark symbol; they are the property of their respective trademark owner. There is no intention of infringement; the usage is to the benefit of the trademark owner.

Contents

1	Introduction and Motivation	1
1.1	Motivation	2
1.2	Main Goals and Results	4
1.3	Thesis Organization	5
1.4	Related Publications	5
2	Preliminaries for Architecture Modeling	7
2.1	Model-Based Software Engineering	7
2.2	MontiCore	11
2.2.1	Symbol Table Framework	14
2.2.2	Language Integration Mechanisms	15
2.2.3	Code Generation Framework	18
2.2.4	Related Language Workbenches	19
2.3	Architecture Description Languages	21
2.4	The MontiArc Architecture Description Language	23
3	Scope and Methodology	29
3.1	Scenario	30
3.2	Requirements	33
3.2.1	Modeling Requirements	34
3.2.2	Model Transformation Requirements	36
3.3	Methodical Guidance	37
3.3.1	Extension with Behavior Languages	41
3.3.2	Architecture Modeling	41
3.3.3	Composed Code Synthesis	43
4	C&C Architectures with Application-Specific Behavior	45
4.1	MontiArcAutomaton ADL	46
4.1.1	Language Elements	47
4.1.2	Symbol Table	51
4.1.3	MontiArcAutomaton Symbol Table	51
4.1.4	Context Conditions	53
4.1.5	Transformations on the MontiArcAutomaton ADL AST	65

4.2	Embedding Component Behavior Languages	68
4.2.1	Syntactic Behavior Language Embedding	69
4.2.2	Symbolic Language Integration	73
4.2.3	Language Integration Infrastructure	76
4.2.4	Language Integration Semantics	81
4.3	Discussion	82
4.4	Related Modeling Languages	83
5	A Behavior Language with I/Oω Automata.	87
5.1	Language Elements	89
5.1.1	Automaton Declaration	90
5.1.2	Inputs, Outputs, and Local Variables	91
5.1.3	Values	91
5.1.4	State Declarations, Initial States, and Initial Outputs	92
5.1.5	Transitions	93
5.1.6	Alternative Stimuli	94
5.2	Symbol Table	95
5.3	Context Conditions	97
5.3.1	Uniqueness Conditions	97
5.3.2	Convention Conditions	98
5.3.3	Referential Integrity Conditions	102
5.3.4	Type Correctness Conditions	104
5.4	A Transformation on the AUTOMATA AST	107
5.5	Integrating AUTOMATA into MontiArcAutomaton	107
5.5.1	Semantics of Integrated AUTOMATA Models	109
5.5.2	Integration Infrastructure	109
5.6	Discussion	110
6	Reusable Architectures through Bindings and Libraries	111
6.1	Modeling Platform-Independent Architectures	113
6.2	Interface Libraries and Implementation Libraries	118
6.2.1	BumperBot Interface Library	121
6.2.2	JavaNXT Implementation Library	123
6.2.3	Python ROS Implementation Library	125
6.3	Deriving Platform-Specific Architectures	128
6.4	Discussion and Related Approaches	130
7	Compositional Code Generation	135
7.1	Code Generator Kinds	137
7.2	Code Generator Description Language	141
7.2.1	Language Elements	141

7.2.2	Symbol Table	146
7.2.3	Context Conditions	147
7.3	Code Generator Composition	152
7.3.1	Developing MontiArcAutomaton Generators	154
7.3.2	Instantiating and Executing Composable Generators	156
7.4	Two Compositional Code Generator Families	159
7.4.1	A Code Generator Family for Java Systems	160
7.4.2	A Code Generator Family for ROS Python Systems	166
7.5	Discussion and Related Work	173
8	Describing Component & Connector Applications	175
8.1	Application Configuration Language	176
8.1.1	Language Elements	176
8.1.2	Symbol Table	179
8.1.3	Context Conditions	180
8.2	Processing MontiArcAutomaton Applications	191
8.3	Modeling MontiArcAutomaton Applications	193
8.4	Discussion and Related Work	196
9	Experiments	197
9.1	Evaluations	197
9.1.1	NXT Java Coffee Delivery	198
9.1.2	Robotino ROS Python Transport Services	200
9.1.3	Robotino SmartSoft Java Transport Services	205
9.2	Case Studies	208
9.2.1	Lego NXT Distributed Toast Service	208
9.2.2	Multi-Platform BumperBot	209
9.2.3	The iserveU Hospital Logistics Project	212
9.3	Discussion	214
10	Conclusions and Future Work	217
10.1	Contributions	217
10.2	Potential for Future Research	219
10.3	Conclusion	220
Bibliography		221
A	Modeling Language Grammars	249
A.1	MontiArcAutomaton ADL Grammars	249
A.1.1	MontiArc Grammar for Human Comprehension	249
A.1.2	MontiArcAutomaton ADL Grammar for Human Comprehension .	250

A.1.3	MontiArcAutomaton ADL Grammar for MontiCore	251
A.2	AUTOMATA Grammars	254
A.2.1	AUTOMATA Grammar for Human Comprehension	254
A.2.2	AUTOMATA Grammar for MontiCore	255
A.3	Generator Description Grammar	255
A.4	Application Configuration Grammar	259
B	Survey Materials	261
B.1	NXT Java Coffee Delivery	261
B.2	Robotino ROS Python Transport Services	263
B.3	Robotino SmartSoft Java Transport Services	268
C	Kinds of Names in MontiArcAutomaton	275
D	Diagram and Listing Tags	277
E	Curriculum Vitae	279
List of Figures		281
List of Listings		285

Chapter 1

Introduction and Motivation

Wisdom begins in wonder.

Socrates

Software engineering for modern application domains, such as cloud-based systems, cyber-physical systems (CPS), or robotics is a complex endeavor due to the heterogeneous and distributed nature of their applications. While component-based software engineering has been applied to such domains successfully, the application of model-driven engineering (MDE) is still emerging. The resulting variety of languages covers from geometric, spatial, and kinematic properties (CPS, robotics) to load distribution, system replication, and database access (cloud) to behavior descriptions and software architectures. Software engineering with C&C software architecture description languages facilitates distributed development and evolution by separating functionality into exchangeable components with stable interfaces, thus increasing reuse and maturity of the software. Describing component behavior by the most appropriate modeling languages liberates domain experts from becoming software engineering experts. Instead, model processing tools embody the knowledge required to transform problem domain models into solution domain source code. The approaches to MDE are diverse and the necessity of reusable, compositional modeling languages has been argued for [KRV08b, JMD⁺14]. Nonetheless, current architecture modeling languages and frameworks do not consider extensibility and reuse sufficiently. Software language engineering (SLE) investigates such traits and has provided solutions realized in form of language workbenches for the compositional development of modeling languages.

This thesis contributes to the application of MDE to complex domains by providing concepts for architecture engineering with exchangeable component behavior languages and code generators. These concepts are realized in the extensible MontiArcAutomaton software architecture modeling infrastructure based on the SLE principles realized in the language workbench MontiCore [KRV08b, Kra10] and the architecture description language (ADL) MontiArc [HRR12]. We extend the MontiCore language MontiArc with concepts to integrate exchangeable component behavior languages, introduce concepts and modeling languages to describe the use of architectures for model transformation

and code generation. The concepts and modeling languages are integrated into an infrastructure to increase reuse of components, architectures, languages, and code generators while separating concerns of participating experts. This infrastructure reduces the effort for integration of the most appropriate component behavior languages into components, enables black-box composition of code generators, and facilitates transformation of software architecture models into executable systems for multiple platforms.

1.1 Motivation

Our work on software architecture modeling with exchangeable component behavior languages, transformations from platform-independent to platform-specific architectures, and composable code generators is driven from experiences with robotics. As modern robotics systems are distributed and concurrent, connected to cloud-systems and data bases, perform complex on-board calculations based on input fused from multiple sensors, and require participation of experts from multiple domains, the results should be generalizable to other domains as well.

Robotics is a heterogeneous domain in which successful deployment of even simple applications requires tremendous effort in crafting solutions specific to participating domains (such as path planning, vision, grasping) and in integration of the resulting software modules. Software engineering for robotics applications thus is a sophisticated endeavor which poses many challenges and successful robotics applications usually are the joint effort of teams of domain experts. This domain-intrinsic complexity forces robotics experts to become software engineering experts as well [BBC⁺07, SSL11] or leads to monolithic and hardly reusable applications [BS06, Mos09, SSL11]. A study on the RoboCup@Home competition of service robots [Mue13] analyzed 31 teams and found that more than 50% develop their own robots instead of reusing existing hardware and software. Of the 31 teams, less than 25% used the same middleware, and at least six different programming languages were used.

Robotics has turned to software engineering, to adopt methodologies, technologies, and tools to tackle these complexities via component-based software engineering [Bru01, ASK⁺05, BBC⁺07, QGC⁺09, NFBL10], and, recently, model-based software engineering [BGBK08, BDHN10, MAHR10, SSL11, ASH⁺12, BBH13, BKH⁺13, OAR⁺14]. With the help of component-based software engineering, robotics researchers aim to separate system complexity behind stable component interfaces, which enables exchanging and reusing components. Such components are developed with general-purpose programming languages (GPLs) and thus are bound to specific paradigms, concepts, and platforms.

From this conceptual gap [FR07] between challenges of the problem domains (e.g., path planning) and answers from the solution domain software engineering - in form of software programs - complexities arise. These are not tied to the challenges itself, but to hand-crafting and integrating solutions with specific implementation details. Such accidental complexities [FR07] increase risks and costs of software projects and require domain experts to become software engineering experts. This conceptual gap can be alleviated by solving domain challenges abstracted from implementation details.

Model-driven engineering [Sel03, SVEH05] aims to reduce this gap by introducing models as development artifacts which can be domain-specific, concise, platform-independent, and better comprehensible than source code programs [MHS05, SMTS09, WHR14]. Such models can be used for communication, documentation, and generation of actual implementations [WHR14]. They can capture domain-specific information (such as kinematics [SCS07, FBC11], force-controlled motions [KSBDS11], geometric relations [LSGB12], manipulation plans [THR⁺13, Van13], or perception information [Hoc13]). They also can capture information specific to software engineering (for instance in form of UML diagrams [OMG10] or software architecture models [MT00, SSL11, BKH⁺13]). Especially software architecture modeling has been applied successfully to robotics [NFBL10, SSL11, Tro11, DKS⁺12, NW12, BKH⁺13], as it combines component-based software engineering with the benefits of model-driven engineering: Architecture models allow identification and separation of logically and physically independent components and to structure the application under development hierarchically.

In contrast to GPL artifacts, architecture and component models can be agnostic of the target programming language, provided by respective experts, and thus can be highly reusable. Therefore, architecture modeling can reduce the conceptual gap by describing large parts of the overall application under development with greater abstraction - for instance via component models, component behavior models, data type models, or deployment models. It furthermore facilitates reuse by enabling exchange of component models with stable interfaces and enables transformation of executable models into multiple target systems via software engineering expertise embodied in code generation tools. Thus it can liberate robotics experts from becoming software engineering experts and integration experts from becoming robotics experts.

Current software architecture description languages (ADLs) lack expressiveness and extensibility with respect to generative software engineering (GSE). Either component behavior is omitted [GMW97, Tro11, BKH⁺13], required as GPL artifacts [MRT99, VVKM00, ACN02, MCWF02, JBCG05, CKS11, SSL11, FG12], or as fixed component behavior languages [BDC02, NFBL10, DKS⁺12]. The former two approaches are of limited use and give rise to accidental complexities. The third approach requires to use any of the prescribed languages. These are usually little abstract and lack the clarity of domain-specific languages (DSLs) [vDKV00, MHS05, FR07, Fow10].

Similar issues hold for employed data type modeling languages: either types are omitted [Tro11, BKH⁺13], GPL types [NFBL10, SSL11], or modeled using a type modeling language (such as UML [OMG10] class diagrams) [Tro11]. The two former approaches are of limited use and tied to GPLs again, while the latter restricts developers to a certain language with specific properties (such as complexity or underspecification). Only few ADLs support integration of arbitrary component behavior languages [NDZR04, FG12] and these are either overly complicated or do not consider integrated code generation for language aggregates.

1.2 Main Goals and Results

This thesis' research objective is to enable seamless model-driven engineering of applications based on C&C architectures with exchangeable component behavior languages. This entails a number of subsequent research questions regarding modeling language integration, model transformation, and code generator composition. We therefore investigate how a C&C architecture description language, its concepts, and infrastructure can be extended to enable language integration, how their models can be enriched with platform-specific information, and how to generate code for combinations or exchangeable languages. For such, we introduce modeling languages, methods, and tools for different aspects of model-driven robotics software engineering with C&C software architectures.

The contributions presented in this thesis are:

- Requirements for an extensible architecture description language that allows integration of exchangeable component behavior languages.
- An investigation and modeling concepts for the distinction of platform-specific and platform-independent component models.
- An architecture description language realizing these concepts.
- Concepts of behavior language integration exploiting the C&C nature of that ADL and a mechanism to realize their integration.
- A state-based component behavior language based on I/O^ω automata and its integration.
- Concepts of platform-specific and platform-independent software architecture models and their relation to another.
- A model-to-model transformation from platform-specific to platform-independent software architecture models.
- Concepts for black-box code generator composition exploiting structural properties of C&C software architectures and its realization.
- The MontiArcAutomaton architecture modeling infrastructure comprising the concepts, ADL, state-based behavior language, model transformation and code generation framework to enable multi-platform generative software engineering.
- Methodical guidance describing the application of the MontiArcAutomaton infrastructure with its features to software engineering projects.

1.3 Thesis Organization

We present concepts, a methodical guidance, and their realization to overcome the limitations presented in Section 1.1. With these, software engineers can model platform-independent logical software architectures and gradually transform these into platform-specific implementations. For this, the remainder of this thesis is organized as follows. First, Chapter 2 introduces the concepts of model-driven development with C&C software architectures and software language engineering. To this effect, it also introduces the language workbench MontiCore [KRV10] and the ADL MontiArc [HRR12]. Chapter 3, presents a usage scenario from which we derive requirements for model-driven engineering of C&C architectures with exchangeable component behavior languages, and describe the resulting software engineering process for MontiArcAutomaton. Chapter 4 presents the MontiArcAutomaton ADL with its language elements, well-formedness rules, and AST transformations. Furthermore, it describes how component behavior languages are integrated into MontiArcAutomaton and the MontiArcAutomaton ADL. Afterwards, Chapter 5 presents a state-based behavior language based on the I/O ω automata paradigm [Rum96, Rin14] and its integration into MontiArcAutomaton. Subsequently, Chapter 6 explains the concepts of multi-platform generative software engineering with MontiArcAutomaton. To this end, it introduces interface components, interface libraries, and implementation libraries. It also introduces the modeling languages and model transformations realizing the aforementioned concepts. Chapter 7 describes the code generator composition with its concepts, constituents, and infrastructure. It also introduces code generator kinds, the code generator description modeling language, and two code generator families for different platforms. Chapter 8 introduces the notion of applications and their configuration for concrete combinations of languages and code generators. Chapter 9 subsequently presents evaluations and case studies using MontiArcAutomaton in various scenarios ranging from academic to industrial projects. Finally, Chapter 10 concludes.

1.4 Related Publications

The results presented in this thesis were produced within five years of research. As such, various parts have been published prior to this thesis. This section describes related publications.

- In the first version of MontiArcAutomaton [RRW12], the ADL consists of a single language featuring language elements for software architectures and automata. Furthermore, the code generation infrastructure is limited to monolithic code generators producing a specific form of Java source code.
- Subsequently, MontiArcAutomaton is extended with a code generation infrastructure for usage with multiple code generators and results on the effort of developing code generators with it are reported [RRW13b].

- That state of MontiArcAutomaton was used in a lab course to develop a system of distributed Lego NXT robots with Java. Results of this course are presented in [RRW13a].
- That state of the MontiArcAutomaton language is also presented in detail a technical report [RRW14a], which omits Java expressions to describe transition actions.
- The first extension of MontiArcAutomaton with other behavior languages is presented in [RRW13c], where automata are a fixed part of the MontiArcAutomaton architecture description language. There, a rules language is manually integrated into MontiArcAutomaton to describe component behavior statelessly.
- Concepts for modeling language integration of MontiArcAutomaton are based on the language workbench MontiCore and are introduced in [LNPR⁺13] and explained in greater detail in [HLMSN⁺15].
- First results on multi-platform generative software engineering with MontiArcAutomaton are presented in [RRRW14]. The notion of code libraries and interface components are extensively revised.
- Preliminary results on code generator composition are reported in [RRW14b]. The presented generator kinds and their interfaces have been modified for this thesis.
- Integration of handcrafted GPL artifacts into generative development is examined in [GHK⁺15].
- The tailoring required to configure a MontiArcAutomaton application is introduced in [RRW15b, RRW15a].
- Translating platform-independent to platform-specific architectures is sketched in [RRW15c].
- An intermediate version of the MontiArcAutomaton software architecture employed in the iserveU research project presented in the case studies is discussed in [HLMSN⁺15].
- An example for language integration and generator composition with MontiArcAutomaton is presented in [RRRW15]

Chapter 2

Preliminaries for Architecture Modeling

The scientific man does not aim at an immediate result. He does not expect that his advanced ideas will be readily taken up. His work is like that of the planter - for the future. His duty is to lay the foundation for those who are to come, and point the way.

Nikola Tesla

The work presented in this thesis builds upon foundations laid by other researchers. Its prime foundations are in model-driven engineering, software language engineering, and architecture description languages. This chapter presents these foundations. First, it introduces foundations in model-based software engineering and related concepts. Afterwards, it describes the language workbench MontiCore [KRV08b, Kra10] and the architecture development framework MontiArc [HRR12], which MontiArcAutomaton relies upon. Subsequent chapters describe contributions based on these and will elucidate details where necessary.

2.1 Model-Based Software Engineering

Within the last decades, amount and importance of software and software-based systems have increased significantly and with this, their complexity has increased as well [FR07]: current state-of-the-art systems are multi-platform¹ distributed heterogeneous systems, which often include physical parts. This increasing complexity of modern software requires methods, concepts, tools, and infrastructures to develop software systems more efficiently.

A significant reason for the complexity of modern software systems lies in the “wide conceptual gap” [FR07] between the problem domains or business domains and the solution domain software engineering. Overcoming this gap with handcrafted solutions requires tremendous effort and gives rise to so-called accidental complexities [FR07], i.e., problems of the solution domain, such as programming language peculiarities, networking communication issues, persistence, deployment, which are not conceptually relevant in the problem domain. These accidental complexities, however, increase software development risks and need to be reduced.

¹In this context, “platform” refers to computers that can execute GPL programs and describes the hardware and software necessary to provide certain functionality.

Model-based software engineering (MBSE) is an umbrella term for software development methodologies which employ domain-specific models to reduce the conceptual gap and with it the accidental complexities. Therefore, MBSE utilizes models as development artifacts in various stages of a software development process, ranging from requirements modeling to implementation to deployment. Computer science, however, does not have a common notion of a “model” [Sei03, Kü05], which is reflected in the numerous definitions produced:

- “A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.” [BG01]
- “A model is a set of statements about some system under study.” [Sei03]
- “A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made.” [Küh06]

Similar definitions are proposed in [HBB⁺94, BD99, Bal00] where most define a model as a simplified abstraction of a system, which can replace the system for certain form of use. Some of these definitions leave only little more room for interpretation regarding what is modeled. We thus follow [Sta73], in which a model is characterized by three features. While the definition is available in German only, it can be translated as presented in [MFBC12]:

“A model needs to possess the following three features:

- Mapping feature: A model is based on an original.
- Reduction feature: A model only reflects a (relevant) selection of an original’s properties.
- Pragmatic feature: A model needs to be usable in place of an original with respect to some purpose.”

We also distinguish between “model-based” and “model-driven” approaches as proposed by [BCW12] and [Sch12], where “model-based” characterizes approaches employing models for documentation, requirements engineering [HPB11], and system design, and “model-driven” [VSB⁺13] characterizes approaches where models are the primary development artifacts used for automated analysis and synthesis (cf. [Sel03, Sel06]). Model-driven architecture (MDA) [OMG03, PM06] can be considered a specialization of MDE with focus on the Unified Modeling Language (UML) [OMG10] standardization efforts proposed by the Object Management Group. A key idea of MDA, driven by the rationale that business concepts outlive their technical realizations, is the separation of platform-independent (but domain-specific) business models and their subsequent application to specific platforms. Model-driven architecture therefore divides an application into different layers representing the application’s computation independent model (CIM), platform-independent model (PIM), platform-specific model (PSM), and GPL

code. In these terms, we propose a pervasive approach to MDE with C&C architectures that employs software architectures on the PIM layer, application models on the PSM layer, and produces GPL code automatically.

Describing models requires corresponding notations, which can have the form of “meta models” [Kü05, SRVK10] or grammars [GKR⁺08] to describe corresponding modeling languages. As modeling aims to increase abstraction, many successful applications of MDE [WWM⁺07, Rai05, KR05, Sta06, HRW11, EvdSV⁺13, WHR14, BCOR15] have been in the context of software engineering for certain domains, such as aviation or automotive. Therefore, the term domain-specific language (DSL) has become popular for such languages. Popular DSLs are, for example, HTML [HTM] (structured documents), Matlab Simulink [Sim] (control systems engineering algorithms), SQL [DD96] (database management), or Verilog [TM02] (hardware description of electronic systems). Employing DSLs promises benefits regarding productivity, quality, validation and verification as they may provide a platform-independent “thinking and communication tool” [Vö11] with minimal overhead. As for the term “model”, computer science has produced a number of definitions for DSLs, such as:

- “A domain-specific language is a programming or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.” [vDKV00]
- “By focusing on a problem domain’s idioms and jargon, DSLs avoid the “notational noise” [Wil01] required when using overly general constructs of a general-purpose language to express the same thing. Moreover, DSLs are not necessarily programming languages: they are languages tailored to express something about the solution to a problem.” [Wil01]
- A “DSL is a language designed to be useful for a limited set of tasks, in contrast to general-purpose languages that are supposed to be useful for much more generic tasks, crossing multiple application domains.” [JB06]

We consequently do not distinguish between “technical DSLs” and “application domain DSLs” [VBD⁺13], distinguish between modeling languages and general-purpose programming languages. The latter are characterized by imposing “notational noise by using overly general constructs” [Wil01], which results in the accidental complexities we aim to avoid. There is, however, a certain consensus regarding the components of a modeling language, which, following [HR04b], are a *concrete syntax* describing models graphically or textually, an *abstract syntax* describing the language’s “structural essence” [Kra10], well-formedness rules restricting the valid models, and the language’s dynamic semantics.

Model-driven engineering yields many benefits ranging from abstraction from accidental complexities to providing comprehensible means to communication and documentation to a formality allowing translation of models into executable systems [VSB⁺13, WHR14]. Avoiding the accidental complexities of the manual translation of models into

executable systems requires tools that translate such models into GPL artifacts automatically while employing codified software engineering expertise.

Tools for generative software engineering [Jéz07, RSVW10] either employ model-to-model (M2M) transformations or model-to-text (M2T) transformations to translate models into GPL artifacts [CH03, EvdSV⁺13]. Model-to-model transformations translate models of the input modeling language into models of the output modeling language (usually based on their abstract syntax). The transformed models' syntax afterwards looks exactly like target language code and, after printing it to plain text, can simply be processed by tooling of the target GPL. While M2M transformations yield various benefits, for instance well-formedness checking of properties of the generated GPL code via checking properties of the target language's model might require less effort than analyzing target language artifacts, engineering and maintaining various target languages requires considerable effort. Furthermore, M2M transformations require transformation languages [JAB⁺06, HHRW15], which usually are little domain-specific and hence impose similar efforts to learning a GPL.

Model-to-text transformations process models of the input language to produce text representing source code of the target GPL. This process does not require a representation of the target language and such transformations usually are implemented employing template languages [CH03, TRMS09]. Usually, these languages allow generator developers to produce templates closely resembling the intended target language artifacts. Thus, M2T facilitates generator development but complicates checking properties of the generated result. Regardless of the nature of performed transformations, we refer to such tools as code generators in this thesis. This conforms the IEEE's definition of a "code generator", which, according to [CoEEB90], is as follows:

Definition 1 (Code Generator). *(1) A routine, often part of a compiler, that transforms a computer program from some intermediate level of representation (often the output of a root compiler or parser) into a form that is closer to the language of the machine on which the program will execute. (2) A software tool that accepts as input the requirements or design for a computer program and produces source code that implements the requirements or design.*

For the scope of this thesis we consider code generators as software tools that transform models conforming to a modeling language into GPL artifacts. Regardless of the generated artifacts, we use the terms "generator" and "code generator" interchangeably throughout this thesis.

Using modeling languages to automatically generate executable systems requires appropriate tool support for language engineering and generator development. MontiCore [KRV08b, Kra10] is a language workbench [EvdSV⁺13] for the development of modular modeling languages and we employ it to engineer the modeling languages presented throughout this thesis. The MontiArcAutomaton ADL architecture description language presented builds upon the C&C ADL MontiArc [HRR12] which is developed with MontiCore as well. The subsequent sections present both and the following chapters will particularize where necessary.

2.2 The MontiCore Language Workbench

The MontiCore [GKR⁺06, GKR⁺08, KRV08b, KRV08a, KRV10] language workbench comprises a modeling language and a toolchain for the efficient and modular development [KRV08a] of modeling languages [RSVW11]. MontiCore employs context-free grammars (CFGs) for integrated definition of abstract and concrete syntax [KRV07, KRV10] of modeling languages. The grammars describe which models are principally possible and well-formedness rules restrict these. The resulting languages are augmented with language configuration files that describe additional language properties (regarding language composition and tooling).

From these CFGs, MontiCore generates model processing infrastructure to parse textual models [GKR⁺07] into abstract syntax trees (ASTs) [Kra10, Vö11]. The ASTs store the content of models, such as their elements and their relations.

```

1 grammar ARC {
2   Component = "comp" Name "{" (Port | Connector | Body)* "}";
3   Port = "port" Type Name? ";";
4   Connector = "connect" src:Name "to" tgt:Name ";";
5   external Body;
6 }
```

MCG

Listing 2.1: MontiCore grammar of the ARC language to define software components with ports and connectors.

Listing 2.1 displays the grammar for a simplified software component modeling language. A grammar begins with the MontiCore keyword “grammar” followed by its name and comprises EBNF-like productions. The displayed grammar begins with the grammar-level keyword **grammar**, has the name **ARC** (l. 1) and contains four productions: **Component** (l. 2) models the structure of software components; **Port** (l. 3) defines a typed, named connection point between components; **Connector** (l. 4) describes a connection between components; and the **external** production **Behavior** (l. 5) serves as grammar extension point and must be implemented by MontiCore language configuration files.

Everything enclosed in quotation marks is considered a model-level keyword (such as “**comp**” in l. 2) and is part of the concrete syntax of this language. Everything else is part of both concrete and abstract syntax. The production **Component** starts with the keyword **component**, followed by a name, and a body in curly brackets. The body can consist arbitrary many instance of **Port**, **Connector**, **Behavior** (denoted by the disjunction operator “|” in combination with the star operator “*”). A **Port** element begins with the model-level keyword **port** and is followed by a type and an optional (denoted by “?”) name. Please note that both productions **Type** and **Name** are provided by MontiCore. A **Connector** consists of the model-level keyword **connect**, followed by a name, the model-level keyword **to** and another name. To distinguish both properties of type **Name**, they are assigned names **src** and **tgt**, respectively. Whenever

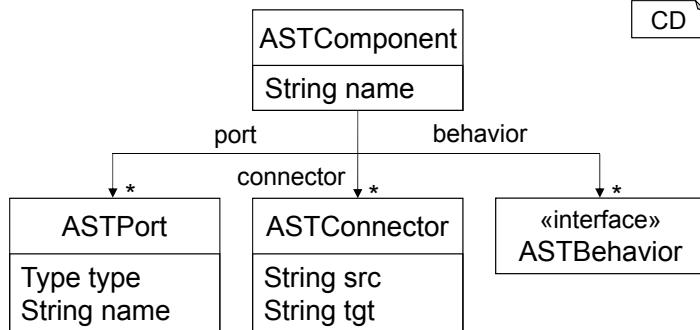


Figure 2.1: The AST node classes resulting from the ARC grammar (cf. Listing 2.1).

this is omitted, such as for the component property of type Name (l. 2), its name is derived from the type, i.e., the property name of Name is derived as name. The external production Behavior has no definition itself. A detailed explanation of the MontiCore grammar language is presented in [Kra10].

For each grammar, MontiCore generates model processing infrastructure, such as parser and lexer [Wir96, ALSU06, Kra10] to create and instantiate the AST node classes. In case of the ARC grammar, this comprises the four classes depicted in Figure 2.1, which represent the respective AST node types. For each production of the grammar, MontiCore creates a class of the production's name with prefix AST. Similarly, MontiCore produces a parser for each production of the grammar. It also translates lists into one-to-many relations, alternatives into multiple fields, and various literals (such as Name into primitives). Thus, MontiCore translates the production component into the class ASTComponent, which has a String property name and three one-to-many relations - one for each alternative. Similarly, it translates the productions Port and Connector. External productions are translated into interfaces. Given a model corresponding to the ARC grammar, the MontiCore parser instantiates the AST node classes to store model properties. The resulting AST instances are basis for subsequent model processing.

Checking properties not expressible with context-free grammars, for instance whether a component contains two ports of the same name, requires additional mechanisms. For such well-formedness checks, MontiCore comprises a compositional context condition framework [Vö11]. Context conditions are well-formedness rules formulated in Java to check model properties. MontiCore distinguishes intra-language context conditions from inter-language context-conditions. The former check properties of models of a single language, the latter check model properties across languages. MontiCore further comprises a model-to-text (M2T) code generation framework [Sch12] to transform ASTs into arbitrary target representations. The actual transformation is described in form of FreeMarker [Fre, TRMS09] templates with additional data management structures [Sch12, RRW13b]. MontiCore also supports compositional language integration [HR13] in form of language extension, language embedding, and language aggregation [LNPR⁺13, HLMSN⁺15].

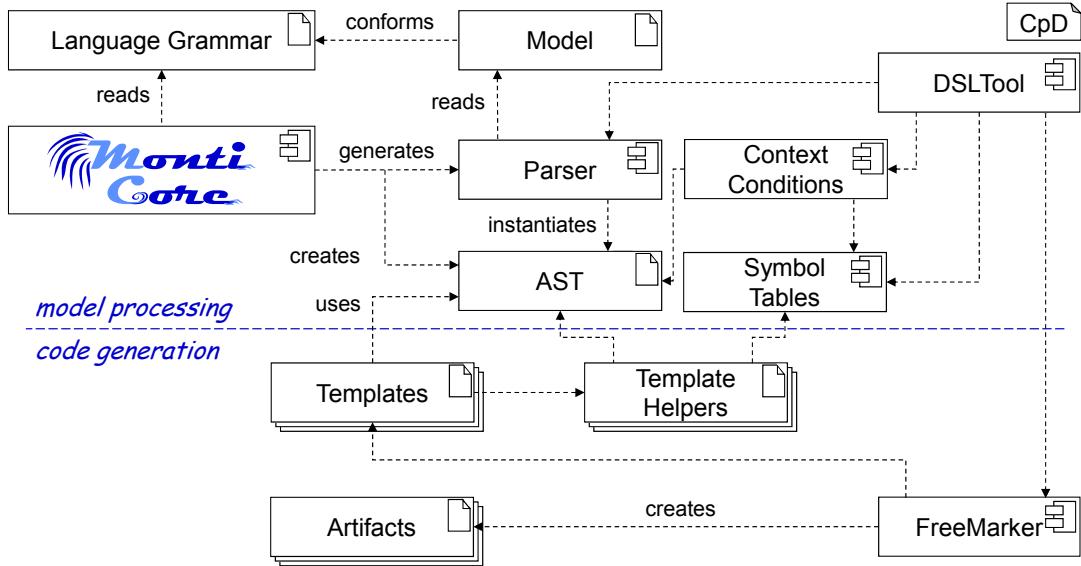


Figure 2.2: Important components and artifacts of the MontiCore toolchain.

Figure 2.2 illustrates parts of the MontiCore toolchain relevant to this thesis and relates produced and required artifacts. Given the context-free grammar of a language, the MontiCore generator creates the model processing infrastructure (abbreviated to “Parser”) to translate conforming textual models into an AST. The language developer also develops intra-language context conditions and inter-language context conditions [Sch12]. Intra-language context conditions check properties of models of a single language. In case models of the language under development are related to models of other languages, inter-language context conditions check related models, which are considered an important source for errors [Tvt+13]. MontiCore also generates a framework to create and manage a language’s *symbol table entries*. These entries encapsulate a model’s essence without the technical necessities of the AST and are stored in so-called *symbol tables* [Vö11, LNPR+13, HLMSN+15], which take care of creating and resolving entries. Context conditions may rely on AST and symbol table. The FreeMarker-based code generation framework of MontiCore [Sch12] comprises templates and template helpers. The former describe production of target language artifacts and the latter perform complex calculations. Templates only rely on the AST, but may invoke template helpers which can access AST and symbol table. Both, the model processing frameworks and the code generation framework of MontiCore are controlled by so-called *DSLTools* [Kra10, Vö11] that configure and execute model processing.

The activities from a textual model to an AST to the symbol table and their entries required for model well-formedness checking are depicted in Figure 2.3, where each activity corresponds to one or more workflows (depending on the processed language). Workflows in MontiCore 3.1.1 are visitors [GHJV95] that are registered with a certain name

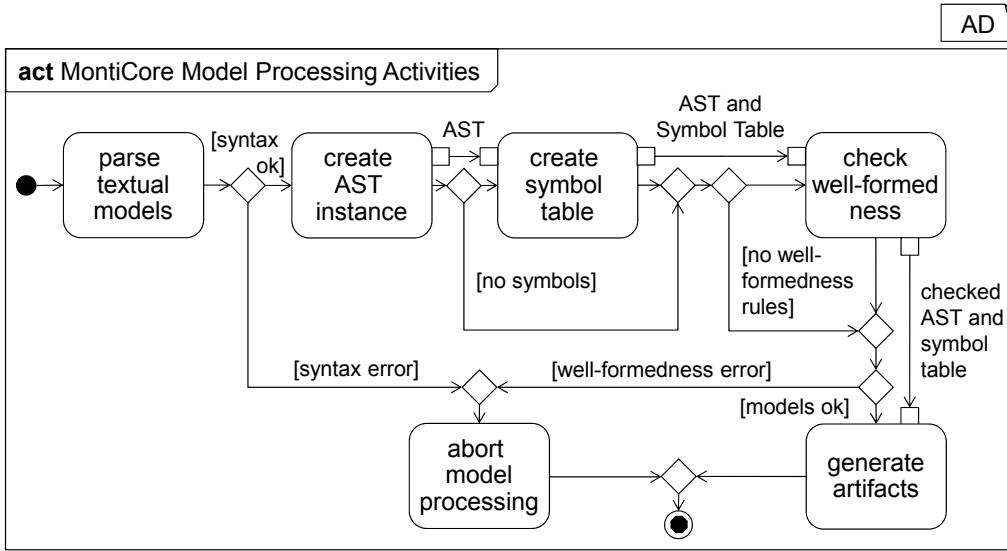


Figure 2.3: A typical MontiCore execution consists of multiple activities to parse models into an AST representation, create their symbol tables, check their well-formedness, and ultimately generate artifacts.

and process certain models. For each language, MontiCore generates workflows to parse models and to prepare their symbol table entries. At start, MontiCore is configured with the workflows to execute by passing their names to it. At first, MontiCore uses the parser generated from the language's grammar to parse the input models. If their syntax conforms to the grammar, their AST instances are created next. Otherwise, model processing is aborted. After creating the AST instances, optional language-specific workflows create *symbol table* data structures that store improved representations of the parsed models based on their ASTs. Afterwards, MontiCore checks optional well-formedness rules using the ASTs and, if available, the symbol table. If successful, toolchain-specific generators may produce (GPL) artifacts. Otherwise, model processing aborts.

2.2.1 Symbol Table Framework

Symbol tables enable compositional language engineering by providing a data structure to store, manage, and retrieve entries of models. Entries encapsulate the “interface” of models in terms of important names and information relevant to their use (such as types and signatures). Entries are free from the technical details of the AST and, hence, provide a greater composition stability regarding grammar (and AST) changes. Thus, they serve as interfaces between interacting models of different languages and their combination is independent from the languages’ actual AST’s. Intra-language well-formedness checks utilize symbol tables to reason over the well-formedness of integrated models and changing the language aggregation thus requires proper adaptation only [Vö11, LNPR⁺13, HLMSN⁺15].

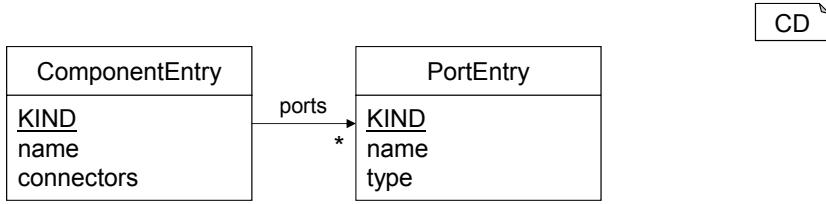


Figure 2.4: Example symbol table entries for the ARC language.

Each symbol of a symbol table has a name and is of a certain *kind*. Further properties are optional. For the ARC grammar, the symbol table *entries* could be as depicted in Figure 2.4. The class `ComponentEntry` encapsulates all information on components required for further processing. This includes a static symbol `KIND`, a name, connectors, and an arbitrary number of ports. The latter are represented by class `PortEntry`, which has a different `KIND`, and members `name` and `type`. Every symbol table provides additional infrastructure not depicted here, such as an entry creator and entry-specific qualifiers, resolvers, and deserializers. The entry creator processes AST nodes and produces symbol table entries of specific kinds from these. These entries are persisted by MontiCore. Whenever a name, for instance of a port type, is looked up, first the qualifier translates the port type's unqualified name (e.g., `Number`) into a qualified name (e.g., `types.Number`) based on the processed model's import statements. Afterwards, the resolver uses this name and the kind of the symbol table entry to look up (e.g., `POR`). The resolver looks up this combination of name and kind in all relevant namespaces and can invoke the deserializer to load matching persisted entries. The implementation of related artifacts, namespace construction, and resolving strategies are explained in different degrees of detail in [Vö11, LNPR⁺13, HLMSN⁺15].

After creating the symbol tables and, with their help, validating the well-formedness of models, they are considered correct. Such models can be used by MontiCore's code generation framework, which comprises templates and template helpers [Sch12] to transform models into target language artifacts. The whole process is invoked and orchestrated by a language-specific DSLtool [Vö11] (cf. Figure 2.2) that serves as interface to language users and orchestrates model processing.

The following subsections present the language integration mechanisms and the code generation framework of MontiCore in greater detail. Afterwards, we highlight related language workbenches.

2.2.2 Language Integration Mechanisms

Effective MDE of complex software systems requires appropriate descriptions of different system aspects. To this effect, efficient integration of modeling languages for these aspects is essential. Integration of modeling languages is tedious and requires profound knowledge on multiple levels. Traditional approaches to modeling language integration require language engineers to compose languages to monolithic aggregates, which are hardly adaptable and reusable in different contexts.

MontiCore languages can be developed independently, are syntactically composable, and ultimately reusable. The enabling concepts are language aggregation, language embedding, and language inheritance [KRV10, LNPR⁺13, HLMSN⁺15], which support generalizable, systematic, syntax-oriented language composition. With these, MontiCore supports the language integration mechanisms as distinguished in [EGR12]. Differences arising from the notion of language embedding identified in [EGR12] (based on [Hud98]) and the notion of language embedding in MontiCore [Vö11] are discussed in [HLMSN⁺15].

Language aggregation denotes the combination of multiple independent languages into a collection (called *language family*), which enables formulating models for different system aspects in separate artifacts that can be interpreted together. This, for instance, is essential when using a domain model formulated in a type definition language (such as UML/P class diagrams [Rum11]) with different behavior or structure modeling languages. The type of a ARC port, for instance, could refer to a class diagram type. Language families describe a loose coupling where participating languages can reference each other's elements by name via their symbol tables. Conceptually, the symbol table entries describe the essence of AST nodes independent of the AST structure and, thus, invariant to language evolution. To enable modular language components and ultimately decouple participating languages the symbol table framework employs adapters [GHJV95] between entries of different languages. For instance, using CD data types with MontiArc requires to provide an adapter from MontiArc type entries to CD type entries, such that resolving a port type Number of ARC type returns this adapter.

Language embedding combines languages such that their elements can be used in a single integrated model. Such embedding is useful where embedding languages require different levels of expressiveness depending on the target platform, for example in form of different SQL [DD96] dialects embedded into class diagram domain models. To this effect, embedding languages provide distinguished extension points where *language fragments* from other languages can be embedded (such as the external production of the ARC grammar illustrated in Listing 2.1). This is realized by mapping productions of other languages to these extension points. Integration is defined in language configuration files (cf. [Kra10, Vö11]), omitting a tight coupling between the languages' grammars.

Language inheritance enables extending, refining, or even restricting existing languages. To this end, MontiCore allows definition of new modeling languages based on existing modeling languages by altering, reusing, and overriding their grammars' productions. Inheritance is especially useful for extending existing languages with new concepts - for instance to engineer restricted versions of complex languages for specific domains (i.e., to add replicating components for load balancing in cloud-based variants of ARC [NPR13]). Language inheritance thus requires both inheriting and inherited language to be conceptually similar.

Figure 2.5 illustrates the technical realization of (a) language aggregation, (b) language embedding, and (c) language inheritance on grammar and parser level. Each language's CFG is accompanied by a language configuration file that describes additional language properties. For instance, such models describe which productions of the embedded lan-

guage are mapped to the embedding language's extension points (cf. the external production Body of Listing 2.1). For language aggregation, the language configuration files (L_1 , L_2) reference each language's own grammar (G_1 , G_2) only. From this, MontiCore generates the independent parsers P_1 and P_2 . The languages' symbol tables S_1 and S_2 rely on these parsers to produce symbol table entries used for well-formedness checking. The language family combining both languages contains not only their symbol tables, but also adapters between symbols and inter-language context conditions. The language families' `DSLTool` class relies its symbol table to process the individual languages' models and to interpret these together. For language embedding, the grammar of the embedding language must provide extension points in form of external productions. The language configuration file L_2 of the embedding language defines how these are implemented. To this effect, it maps productions from the embedded language's grammar G_1 to external productions of the embedding language's grammar G_2 . The resulting combined parser P_2 delegates parsing of embedded productions to the embedded P_1 where applicable. Each language maintains its own symbol table and the symbol table of the embedding language may reference symbols table entries of the embedded language. The embedding language's `DSLTool` class uses the symbol table S_2 only. Language configuration files are unaware of language inheritance. Here, the grammar G_2 inherits from the grammar G_1 . MontiCore generates parsers for both and the language family `DSLTool` uses the parser generated for the inheriting language P_2 which uses P_1 whenever a production of the inherited language is processed. Both languages provide their own symbol tables, although the inheriting language's symbol table S_2 may extend the inherited language's symbol table S_1 . MontiCore processes language embedding and language inheritance similarly, however, embedding enables use of selected productions from the embedded language only while extension enables use of all productions of the inherited language.

The language integration mechanisms of MontiCore affect the resulting AST nodes differently: for language aggregation, the languages' ASTs are not affected at all as the languages are integrated on symbol table level only. For language embedding, the integrated language's AST has subtrees of the embedded language at the external production extension points. For language inheritance, the AST contains individual nodes of the extended language. Only the latter enables to arbitrary mix nodes of both languages.

Figure 2.6 illustrates the effects on the ASTs of integrated languages. Language aggregation (a) integrates independent models of different modeling languages into a language family: first the models are parsed individually and their ASTs are created, afterwards, the symbol table framework resolves and manages references between independent ASTs of such models. Their actual integration, for instance to check inter-language properties, is performed via their symbol table entries. The language family's symbol table references both languages' symbol tables and performs entry adaptation where necessary. For language aggregation, existing tooling can be reused or easily combined. Instances of embedded languages (b) are defined in integrated models, from which MontiCore produces combined ASTs. Here, subtrees of the embedded language replace external productions of the embedding language. Nevertheless, context conditions for subtrees can easily

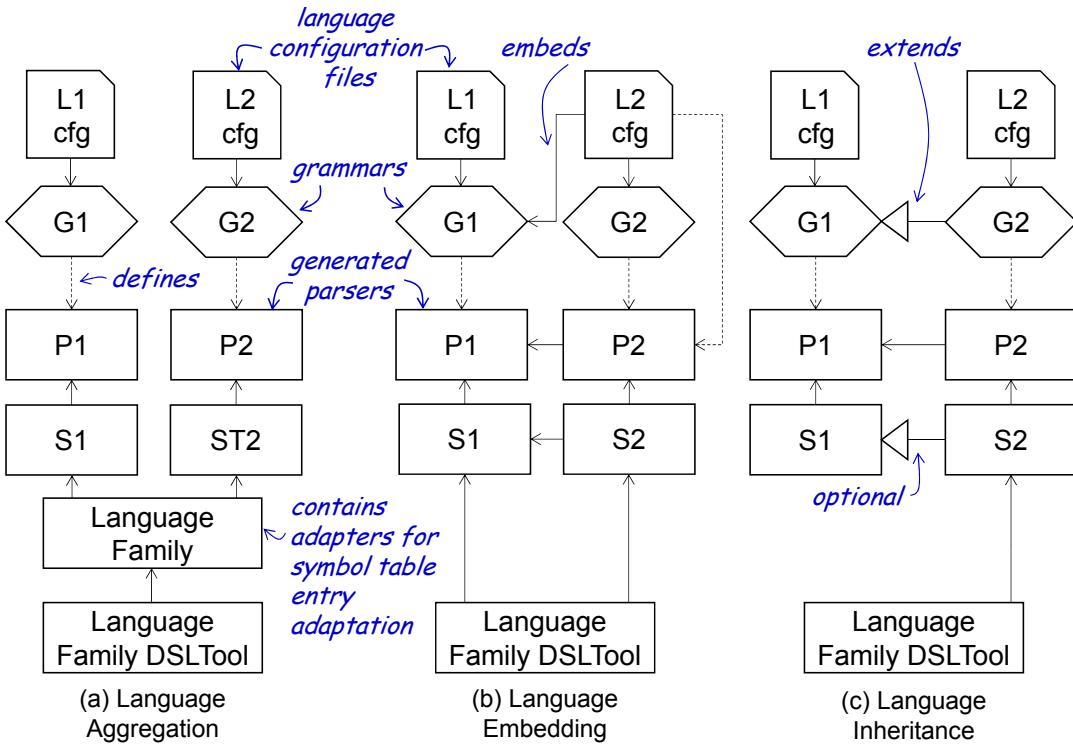


Figure 2.5: Effects of the language integration mechanisms on the generated parsers, symbol tables, and language processing tools.

be reused. In case inter-language context conditions between embedded language and embedding language are required, the context condition and symbol table frameworks enable integration of such. Using language inheritance (c) also yields integrated models, but allows reusing all productions and context conditions of the super grammar.

2.2.3 Code Generation Framework

The MontiCore code generation framework [Sch12] employs the FreeMarker [Fre] Java template engine to facilitate development of code generators. A MontiCore code generator consists of a set of FreeMarker templates, which can access the currently processed models' AST, and additional data management and computation infrastructure specific to MontiCore. The latter comprises *template operators* and *template helpers* [Sch12, RRW13b]. Template operators are generic to MontiCore generators and provide data management and template control methods, such as blackboard-like [AZ05] storage and persistence, as well as access to AST, template helpers, and sub templates. Templates contain FreeMarker expressions, control structures, and target language elements. Template helpers are specific to individual code generators and encapsulate complex calculations not conveniently expressible with FreeMarker. To this effect, they have access to template operators, AST nodes, and the symbol table.

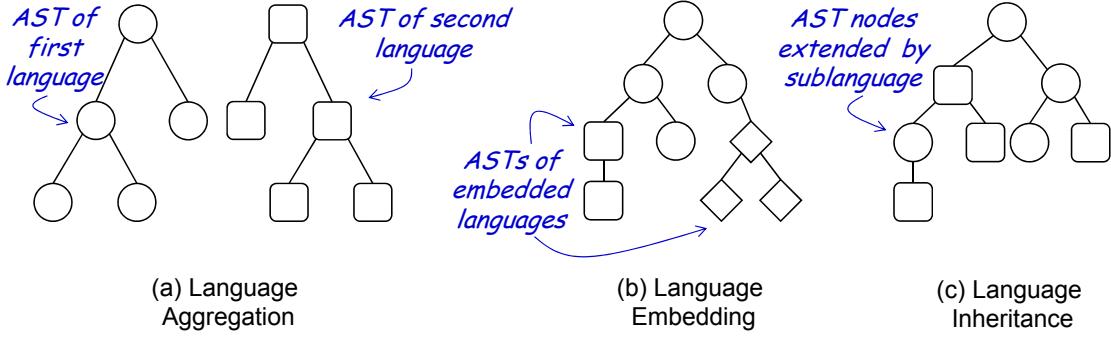


Figure 2.6: Effects of language aggregation, language embedding, and language inheritance on the ASTs of the participating languages: aggregation produces separate ASTs for each model, embedding produces a single AST with subtrees at leaves of the embedding language, and inheritance also produces a single AST containing extended nodes of the inheriting language.

Listing 2.2 illustrates part of a MontiCore FreeMarker template usable to translate ARC models into Java classes. The template uses FreeMarker expressions and the AST node of the currently processed model part (accessible via variable `ast`) to produce a class declaration with the AST node's name (l. 1). Here, `public` and `class` are plain text and will be printed to the resulting artifact as stated. The subsequent FreeMarker loop `<#foreach>...</#foreach>` (ll. 2-10) iterates over the ports of the current AST node (l. 2) and utilizes methods of their AST nodes to print corresponding Java members to the generated artifact (ll. 3-8). As the name of ports is optional (cf. Listing 2.1, l. 3), the template uses the FreeMarker conditional `<#if>...</#if>` to determine the port's name and assigns it to the template variable `name` (ll. 4,6). In case the name was omitted (l. 6), `name` holds the port's type name starting with a lower-case letter (via the FreeMarker built-in function `lower_case`). Finally, the member is printed (l. 8) and another template to produce the member's `get()` method (l. 9) is included. The transformations of `Connector` and `Body` AST nodes (cf. Listing 2.1, l. 5) employ similar mechanisms and are omitted for clarity.

MontiCore code generation is invoked with a map from AST types to templates and will apply the specified templates to each instance of the corresponding AST type. With this, invoking different code generators for (different parts of) a model and thus generating code for different aspects of the target system requires little effort.

2.2.4 Related Language Workbenches

A detailed review of different language workbenches has been conducted in [EvdSV⁺13]. The authors examined the language workbenches Ensō [vdSCL14], Más [Má], MetaEdit+ [KLR96], MPS [VBD⁺13], Onion [EvdSV⁺13], Rascal [KvdSV09], Spoofax [KV10], SugarJ [ERKO11], Whole Platform [Sol05], and Xtext [EB10] regarding syntax, validation, semantics, and editor services. All workbenches support syntactical composition, but

```
1 public class ${ast.getName()} {  
2 <#foreach p in ast.getPorts()>  
3   <#if p.getName() ??>  
4     <#assign name = p.getName()>  
5   <#else>  
6     <#assign name = p.getType()?lower_case>  
7   </#if>  
8   private ${p.getType()} ${name};  
9   ${op.includeTemplates(getter, p)}  
10 </#foreach>  
11 }
```

FM

Listing 2.2: FreeMarker template for transformation of ARC (cf. Listing 2.1) models to Java code.

compositional validation of integrated models, for instance for naming and type checking, similar to MontiCore is supported by MPS, SugarJ, and Xtext only.

The language design concepts of MontiCore are similar to Xtext [EB10] which also employs parser generation from context-free grammars to produce ASTs of DSLs that can be subject of further processing. Xtext supports language inheritance and language aggregation similar to MontiCore but does not provide support for language embedding. Code generation with Xtext is template-based as well and employs the Xtend [Bet13] language to define templates.

The Kermeta [JBF11, JCB⁺13] language workbench focuses on meta-language composition for the different concerns of software systems. To this effect, the workbench employs a modeling language based on EMOF [OMG06] for metamodel definition, a variant of OCL [OMG10] to describe well-formedness rules, and the Kermeta language to define dynamic semantics. In contrast to MontiCore, Kermeta focuses on model interpretation, for which it utilizes dynamic semantics specified in the Kermeta language.

The Meta Programming System (MPS) [VS10] employs a parser-less, projectional, approach to language engineering. Here, language engineers directly specify the AST and editors for specific AST nodes. Language users directly manipulate the model’s AST, which omits the need for parsing. However, the lack of an abstraction such as the symbol table, ties language integration to the participating languages’ ASTs and hence breaks whenever the AST changes sufficiently. Code generation with MPS is realized with model-to-model (M2M) transformations, which requires ASTs of the target languages as well. Furthermore, due to the projectional editing, MPS is tied to the MPS IDE and cannot be integrated into other development environments easily.

Another language integration approach are “domain-specific embedded languages” (DSELs) [Hud96, vDKV00, Fow10] or “internal DSLs”, which employ GPLs to design APIs resembling DSLs [HO10, Gho10]. While circumventing the need for DSL aggregation and enabling reusing existing infrastructure, integration is limited to the host language’s expressiveness and there are no established concepts for language embedding or inheritance yet. Furthermore, DSELs are subject to notational noise [Wil01].

Attribute grammars [Knu68] are grammars enriched with computation rules, which can be used to add semantic information to the corresponding AST. Multiple inheritance of attribute grammars is an interesting approach [Mer13] to language integration as well. While research led to further promising results on language integration, such as Forwarding [WdMBK02], and produced apt language workbenches [WBGK08], we focus on syntactic language integration. Further discussions of related language workbenches have been conducted earlier and detailed arguments for the development of modeling languages [Kra10], their integration [Vö11], and development of code generators [Sch12] with MontiCore have been raised accordingly.

2.3 Architecture Description Languages

Engineering complex software systems requires expertise from various domains: on one hand are problem domains with specific concepts, knowledge, and requirements - on the other hand are solution domain concepts and technologies, such as specific programming languages, network communication, or data management. Following this dichotomy, sufficiently sophisticated software projects would require domain experts to become software engineering experts or vice versa [BBC⁺07, SSL11]. As this usually is unfeasible, resulting software systems are often monolithic and hardly reusable [BS06, Mos09, SSL11].

Computer science, and software engineering in particular, have brought forth concepts of software modularization to alleviate this issue. The level of abstraction ranges from the encapsulation of object-oriented software construction [Sny86] to component-based software engineering (CBSE) [McI68], where software *components* can be full-fledged applications, to MDE, where primary development artifacts focus on domain issues instead of technical issues. Component-based software engineering is a software development paradigm which aims to increase reuse of software by encapsulating functionality behind well-defined, stable interfaces of software components. Such software components are, following [SGM02], “a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition”. The contractually defined stable interfaces enable a distributed development of components by their respective experts, increase reuse and ultimately maturity of software components. Isolated software components are insufficient to describe the system aspects required to achieve executable systems. They may, however, serve as building blocks for a system’s *software architecture*. The software architecture of a system is the set of its principal design decisions [MDT07, TMD09], which “involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns” [SG96]. This may include, for instance, the connections between components, their behavior and configuration, or their deployment onto target platforms. The notion of component & connector architectures (C&C) [MT00, BS01, TMD09] is a popular approach to describe the logical decomposition of a system’s software architecture. Components of such architectures perform computations and communicate by exchanging messages via connectors between ports of their interfaces.

Many domains have adopted notions of software architectures with GPL or binary components successfully [HK00, Bru01, BHH02, NTN⁺04, ASK⁺05, LTR05, BBC⁺07, QGC⁺09, TRMS09, NFBL10, MAK11, SSL11, NPR13]. Such components limit exchange and reuse to systems that can process the same GPL components and fulfill the same software dependencies. Facilitating reuse of components requires describing components and their composition abstract from technological dependencies. This conforms to the definition of software components contributed by [HC01], where a software component “conforms to a component model and can be independently deployed and composed without modification according to a composition standard”. Such conformance requires explicating rules for component creation, composition, and deployment, but ultimately facilitates reuse. However, neither the definition of [SGM02] nor the definition of [HC01] entail that a software component is executable, instead [HC01] introduces the notion of a “component model implementation [as] the dedicated set of executable software elements required to support the execution of components that conform to the model”.

While component models may be expressed in GPLs, this distinction between component models and their implementation helps to understand the conceptual gap between abstract, high-level models and their GPL implementations full of accidental complexities. Employing component behavior languages, this gap can be reduced by describing the intended behavior or the component implementation on model level as proposed by [BGM10]. This also enables utilizing MDE methods and tools that embody software engineering expertise to produce executable component implementations.

Component & connector architecture description languages (ADLs) [MT00, MDT07] are languages to express software architectures in terms of components, connectors, and configurations. Popular examples for such languages are the architecture analysis & design language AADL [BFBFR07, FG12], ACME [GMW97, GMW00], Arch-Java [ACN02], AutoFocus [BHS99], Darwin [MDEK95], EAST-ADL [DSL05], Fractal [BCL⁺06], Koala [VVKM00, ASM04], Mechatronic UML [BGT05], MontiArc [HRR12], UML [OMG10] component diagrams, SysML [FMS11] internal block diagrams, and xADL [DVdHT01, NDZR04].

A recent survey investigating industrial demands on architectural languages [MLM⁺13], the authors interviewed 48 software engineering practitioners from 40 companies. Thereby, they identified that the most important properties of such are well-defined semantics and support for iterative development, proper tooling, extensibility, versioning, and views. Similar to [MT00], the survey finds that ADLs furthermore should be intuitive and should avoid notional noise [Wil01]. Conforming to [WHR14], the survey also finds that abstraction and comprehensibility are more important than code generation.

MontiArc [HRR12] is a C&C ADL that employs the messaging semantics of Focus [BS01, BDD⁺93] as foundational theory and realizes these in form of a MontiCore modeling language. The work presented in this thesis builds upon MontiArc, wherefore the next section introduces the language.

2.4 The MontiArc Architecture Description Language

MontiArc [HRR10, HRR12] is a MontiCore language for intuitive modeling of comprehensible C&C software architectures. As such, versioning and model-differencing are available and foundations for language extension are provided. It features the core elements of C&C ADLs as identified by [MT00], i.e., components with interfaces, types, and semantics, connectors, and architectural configurations. MontiArc extensions for variability [HKR⁺11], refinement [Rin14], and views [MRR13, MRR14] are available.

With MontiArc, a system's functionality is decomposed into hierarchical components, which provide interfaces as sets of typed and directed ports. Types of ports are either defined in terms of UML/P class diagrams [Rum11, Sch12] or in terms of Java/P types. The latter is a modeling language resembling Java 1.5 [Sch12]. Hence, the basic data types of MontiArc are models of the primitive data types of Java 1.5. As the Java/P and UML/P class diagrams are integrated, complex types can be composed from combinations of both. Components encapsulate subsets of the systems functionality and are either composed or atomic. Composed components contain a hierarchical topology of subcomponents (their *subcomponent hierarchy*) that exchange messages via unidirectional, logical connectors between the typed ports of their interfaces and their behavior emerges solely from the subcomponents and their interaction. Connectors may connect one outgoing port to many incoming ports but not vice versa. Atomic components perform calculations via handcrafted GPL behavior implementations integrated via naming conventions. Furthermore, MontiArc also employs a powerful type system featuring component configuration parameters and generic type parameters. The latter can characterize port types as well. As MontiArc does not provide languages to describe the input-output behavior of components explicitly, behavior of atomic components must be defined in Java classes related to components by name conventions (i.e., for a component `Robot`, MontiArc expects the Java class `RobotImpl` in the same package). As such, similar to other ADLs [MT00], its components are tied to a GPL.

MontiArc distinguishes the definition of component types from their instantiation, which allows reusing component types where desired. Component types can be subject to inheritance similar to classes in object-oriented programming languages: a component type extending another inherits the super component type's ports, configuration parameters, and generic type parameters, but not its behavior. MontiArc also features a sophisticated type system to describe the types of ports and their relations. The type system resembles Java in being object-oriented, supporting inheritance, interfaces, multi-dimensional types, and generic types. Instead of using the Java type system, using its own type system enables adapting MontiArc to other type systems with little effort.

Figure 2.7 illustrates the most important modeling elements of MontiArc by example of a software architecture for a simple robot. The robot comprises of a front-mounted ultrasonic sensor to detect obstacles, and two parallel motors to propel the robot. Once an obstacle is detected, the robot backs up, rotates, and continues to drive forward.

The software architecture consists of components of the types `UltraSonic`, `Timer`, `BumpControl`, `Navigation`, `Translator`, `Motor`, and `MontiArcBumperBot`. The

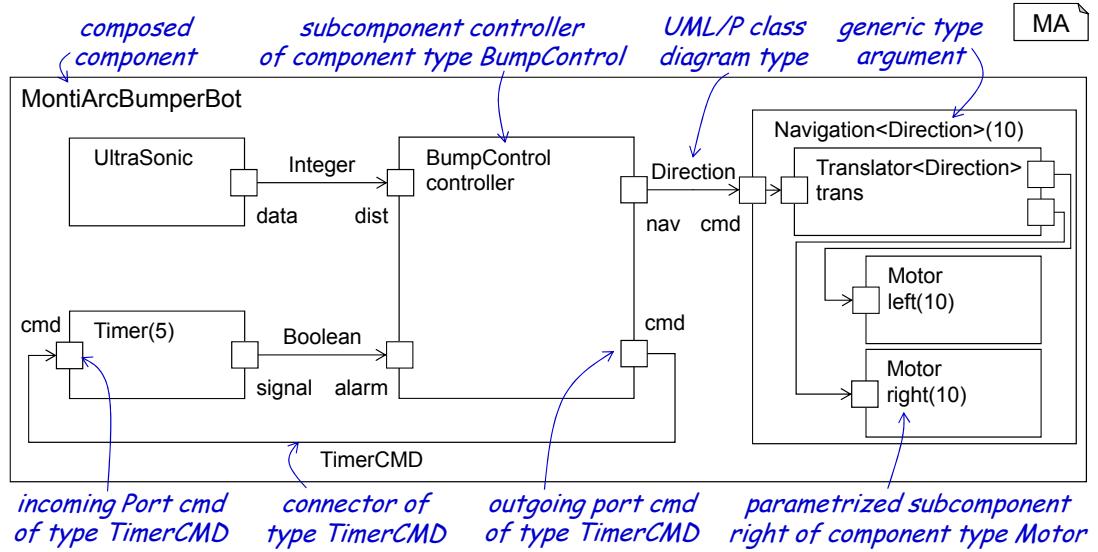


Figure 2.7: A MontiArc software architecture of a simple robot. The composed component **MontiArcBumperBot** contains four subcomponents of different types to read sensor data, interpret it, and actuate two motors.

component **MontiArcBumperBot** is the top level component of the software architecture and contains subcomponents of the types **UltraSonic**, **Timer**, **BumpControl**, and **Navigation**. We denote the top-level component of an architecture as *architecture root* or *root* when the context is unambiguous. The component **Navigation** is composed again and contains three subcomponents **trans**, **left**, and **right**. It also yields a generic type parameter that needs to be passed at instantiation to define the type of its incoming port **cmd**. In Figure 2.7 the component **trans** is instantiated with the type argument **Direction** for its generic type parameter. This argument passed to subcomponents of type **Navigation** is also passed down to its subcomponent **trans**, which uses it to define the type of its incoming port. Furthermore, **Navigation** expects a numerical configuration argument (here instantiated with the value 10) that is passed down to **left** and **right**.

Whenever the subcomponent instance name is omitted, as for example with the component of type **UltraSonic**, the name is derived automatically to be the type name with uncapitalized first letter (e.g., **ultraSonic**). The resulting subcomponent **ultraSonic** uses the outgoing port **data** to emit **Integer** messages to subcomponent **controller** of type **BumpControl**. The subcomponent **controller** also receives **Boolean** input from subcomponent **timer** of type **Timer** and sends messages to **navigation**. The subcomponent **timer** is parametrized with the value 5 and starts intervals of that length whenever triggered. The subcomponent **navigation** passes incoming **Direction** messages to its subcomponent **trans**, which translates these into messages for two connected subcomponents **left** and **right** of type **Motor**. Figure 2.7 does not reveal whether the component types **UltraSonic**, **Timer**, and **BumpControl**

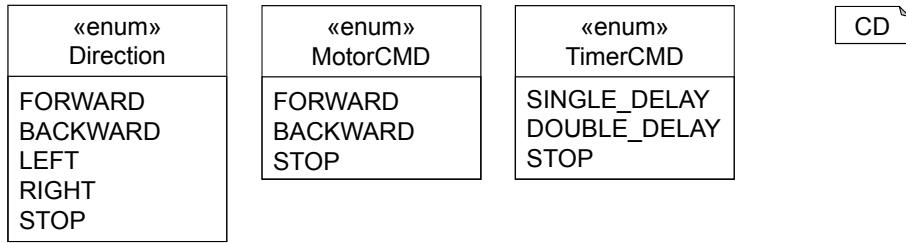


Figure 2.8: The data types `Direction`, `MotorCMD`, and `TimerCMD` as used by the MontiArcBumperBot software architecture depicted in Figure 2.7.

are composed or atomic. This is hidden in component interfaces and facilitates modular and iterative development of architectures.

The data types of the ports `BumpControl.nav`, `BumpControl.cmd`, `Motor.cmd`, and `Timer.signal` are modeled as UML/P class diagrams (CDs) [Rum11, Sch12]. Figure 2.8 shows the corresponding CD, where the types `Direction`, `MotorCMD`, `TimerCMD` are implemented as enumeration types. Messages of data type `Direction` describe the direction the robot should move. `MotorCMD` messages describe how the motor is supposed to rotate and are be translated by component `Motor` into a format the underlying software understands. Messages of type `TimerCMD` control the component `Timer` and start a countdown of the interval length, the subcomponent was parametrized with (in this case 5), of twice the interval length, or stop it.

```

1 package robots;
2
3 import datatypes.*;
4 import robots.sensors.*;
5 import robots.actuators.*;
6
7 component MontiArcBumperBot {
8     component UltraSonic;
9     component Timer(5);
10    component BumpControl controller;
11    component Navigation<Direction>(10);
12
13    connect ultraSonic.data -> controller.dist;
14    connect timer.signal -> controller.alarm;
15    connect controller.nav -> navigation.cmd;
16    connect controller.cmd -> timer.cmd;
17 }
```

Listing 2.3: Textual model of the MontiArcBumperBot component depicted in Figure 2.7. The names for the subcomponents of types `UltraSonic`, `Timer`, and `Navigation` are derived from their types' names.

MontiArc models are textual and each model contains at least one component type definition. Listing 2.3 presents the textual model for the component MontiArcBumperBot as depicted in Figure 2.7. First, the package is declared (l. 1) and the required data types (l. 3) and component types (ll. 4-5) are imported. Afterwards, the component type definition begins (ll. 7-17) which starts with the keyword `component` followed by the component's name. A component definition may contain subcomponent declarations (ll. 8-11) and connectors (ll. 13-16). Each subcomponent declaration references a component type and may specify a name and further arguments to describe how the subcomponent of the referenced type should be instantiated. The declaration of subcomponent navigation (l. 11) shows the application of type arguments in angle brackets and configuration arguments in round brackets. This follows the notation of Java and similar languages. Connectors connect one incoming port of either the containing component definition or one of its subcomponents to multiple outgoing ports of the containing component definition or its subcomponents. Modeling generic type parameters, configuration arguments, and subcomponents with multiple instances is illustrated with component type Navigation as depicted in Listing 2.4.

```
1 package robots;
2
3 import datatypes.*;
4
5 component Navigation<T>[int maxSpeed] {
6   port
7     in T cmd;
8
9   component Translator<T> trans;
10  component Motor(maxSpeed) left, right;
11
12  connect cmd -> translator.input;
13  connect translator.a -> left.cmd;
14  connect translator.b -> right.cmd;
15 }
```

MA

Listing 2.4: Textual model of the component type Navigation as depicted in Figure 2.7 and Listing 2.3.

The signature (l. 5) of component type Navigation declares the type parameter `T` in angle brackets followed by the configuration parameter `maxSpeed` of data type `int` enclosed in square brackets. The type parameter `T` defines the type of incoming port `cmd` (l. 7) and is passed to its subcomponent `trans` (l. 9). Similarly, the configuration parameter `maxSpeed` is passed down to both `Motor` instances defined by a single subcomponent declaration (l. 10).

MontiArc employs context conditions to check the well-formedness of models, such as uniqueness of names, existence of referenced model elements, and naming conventions. The language elements and context conditions of MontiArc are available [HRR12].

Communication of MontiArc components is based on the FOCUS [BS01, BR07, RR11] framework. Components send and receive sequences of messages via unidirectional channels (represented by connectors). A channel transports messages of its type in order of their transmission. Its semantics over time is formalized as an ordered *stream* of messages $\langle m_1, m_2, m_3, \dots \rangle$. While streams preserve the order of messages, no assumptions on the time lag between two messages can be derived. FOCUS allows simulating the progress of time using *time slices* delimited by *tick* messages (\checkmark). For messages contained within a time slice, only the order of transmission is preserved. FOCUS provides three different timing paradigms: *untimed*, *timed*, and *time-synchronous*: untimed streams contain messages only, timed streams contain an arbitrary but finite number of messages in each time slice, and time-synchronous streams contain up to one message per time-slice. The effects of different time paradigms on MontiArc components are illustrated in [HRR12]. The code generation and simulation framework of MontiArc generates Java code for timed and time-synchronous communication [Mon]. Furthermore, FOCUS distinguishes components that produce output immediately from components that produce output in the next time slice. The former are called *weak-causal* and may lead to issues with component cycles (see [HRR12] for a discussion). The latter are called *strong-causal* and automatically produce a delay, hence, cycles with at least one strong-causal component are unproblematic.

However, the MontiArc code generation framework is available in Java only and atomic components always require GPL behavior implementations. Consequently, the framework assumes that for each generated implementation of an atomic component a Java behavior implementation following certain naming conventions exists. This prohibits usage with platforms that cannot support Java (which, for instance, in robotics is the majority [Mue13]). Furthermore, this also implies that domain experts, who contribute components, know Java as well. MontiArc also does not distinguish between platform-independent and platform-specific components. As this link is established for generated code, the software architecture cannot disclose whether a certain component has an implementation in the required platform GPL. Replacing the GPL behavior implementation of a component with another of the same GPL (e.g., to use a different type of sensor) requires error-prone handcrafting and expertise of coed generator details. With MontiArc, the software architecture is the single development model. This prohibits to explicate properties of the generated implementations, or the architectures usage without narrowing its applicability further.

Chapter 3

Scope and Methodology

*The game of science is, in principle, without end.
He who decides one day that scientific statements do
not call for any further test, and that they can be
regarded as finally verified, retires from the game.*

Karl Popper

Efficient model-driven engineering of reusable multi-platform applications with C&C software architectures requires modeling languages to express all “principal” design decisions of such architectures. This does not only comprise structural decisions, such as component topology, message data types, or architectural configurations, but also decisions regarding system behavior, and interaction. The principality of such decisions depends on the system goals and thus varies between applications. Hence, the modeling elements required to describe such decisions vary as well. This can be condensed to: “In architecture modeling, one size will never fit all” [MDT07].

To achieve acceptance and reuse of an ADL for multiple applications, it must be applicable to many challenges. Thus, “extensibility is a key property of modeling notations” [MDT07]. We therefore propose an extensible architecture modeling infrastructure, which is supported by the powerful language definition and integration mechanisms of the MontiCore language workbench and supports extension of its C&C ADL with exchangeable component behavior languages, development of platform-specific implementations from platform-independent architecture models, and automated translation of such models with compositional code generators into artifacts for arbitrary platforms.

The following section illustrates the intended usage of MontiArcAutomaton by the example of developing a robotics application with a specific component behavior language and code generation for two different platforms. Afterwards, Section 3.2 presents the requirements for efficient development of platform-independent C&C software architectures with exchangeable component behavior languages. Finally, Section 3.3 describes the important parts of the MontiArcAutomaton infrastructure and the corresponding software engineering process including tailoring MontiArcAutomaton to company-specific or application-specific requirements.

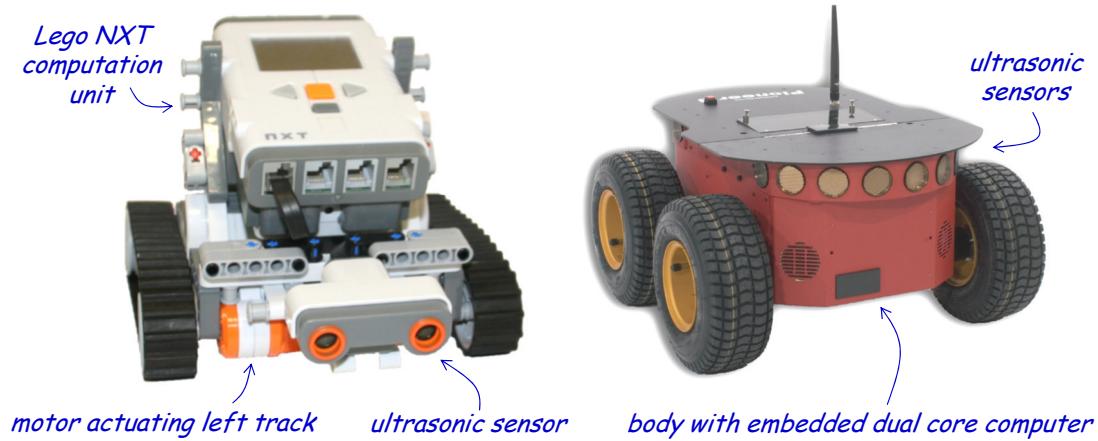


Figure 3.1: The hardware platforms to be used for exploration: a LEGO NXT robot running LeJOS [LeJ] and a Pioneer 3-AT (photo via www.activrobots.com) running ROS [QGC⁺09].

3.1 Scenario

Consider a company that is going to produce robots for exploration of unknown areas. These robots are supposed to drive around the area they are deployed into until they approach an obstacle, then they should back up, rotate, and start exploring again. The company is going to produce two versions of the system to serve different customer requirements: the first version should be affordable and mostly for indoor education purposes, while the second version should be robust and usable for outdoor all-terrain exploration. To reduce cost, the company is going to use off-the-shelf hardware and software. The educational system is supposed to use LEGO NXT robots as depicted on the left of Figure 3.1. These robots employ two parallel tracks, an ultrasonic distance sensor, a central computing unit (the LEGO NXT “brick”) with a CPU speed of 48 MHz, and 64 KB RAM, which has to support execution of the complete software architecture. Each track is powered by a single motor and all motors and sensors are connected to the computation unit. The company chooses the Java LeJOS NXT operating system [LeJ] as robot operating system, which provides interfaces to the NXT hardware in form of a reduced Java virtual machine with a concise API. For the second system the company chooses to use a Pioneer 3-AT four wheel drive robot with 8 front-mounted ultrasonic sensors and an embedded dual core computer with 2.26 GHz and 8 GB RAM. Interfaces to the robot’s hardware can be accessed using the robot operating system ROS [QGC⁺09] with multiple programming languages. As the ROS support for Java is less mature than for Python, the company is going to use the Python implementation of ROS.

While both platforms require different programming languages to provide the same functionality, the logical architecture and high-level functionality of both applications will be mostly identical. To reduce software engineering costs, the company is going to model the software architecture, which enables reusing the same architecture model with

both robots. As the behavior of both robots is identical as well, their central controller also is modeled to be reusable.

Now, a software engineer has to develop the common software architecture for both platforms. Based on the company's requirements, she decides to use a C&C ADL to model the architecture and class diagrams for the architecture's data types. This enables her to model the structure of the architecture platform-independently and to delegate development of component behavior for sensors and actuators to respective experts. She therefore decomposes the system's functionality into five component types:

1. A `DistanceSensor` component which emits the distance to the next obstacle. Whether this actually uses ultrasonic, laser, or infrared is left underspecified.
2. Components of type `Motor` propel either one of the tracks and or two wheels on one side of each robot. To this effect, their implementations must interface the motor drivers. How these are interfaced is independent of the component type's interface and transparent to component users.
3. The `Timer` component type manages countdowns to support timing operations.
4. The component type `ExplorationControl` realizes the robots behavior. To describe it platform-independently, the company's exploration behavior expert uses a behavior language that can be translated into platforms-specific GPL implementations, such as the company's specific variant of UML Statecharts [OMG10].
5. The top-level component `ExplorerBot`, which contains subcomponents of the other types and their connections.

Figure 3.2 depicts the resulting software architecture. Here, the central subcomponent controller of component type `ExplorationControl` receives messages from sub-components `sensor` of component type `DistanceSensor` and `timer` of component type `Timer`. Based on these messages, `controller` calculates the robot's next action and translates it into messages for components `left`, `right`, and `timer`.

The component types `DistanceSensor`, `Motor`, and `Timer` require platform-specific implementations to interface the ultrasonic sensors, motors, and operating system features of the underlying platform. Describing their component behavior requires interaction with the corresponding, platform-specific GPL APIs. Hence, modeling their behavior with abstract DSLs is beneficial only if these DSLs are designed to interface these specific APIs or are general enough to resemble the APIs' GPLs. The former DSLs are hardly reusable, the latter resemble GPLs. In lieu of developing such three DSLs for these component types, a LeJOS API DSL, or a reduced Java DSL to interface the required APIs, these components are developed as *interface components* and integrated into the software architecture.

Interface components may not describe component behavior in any form and need to be replaced by platform-specific components of the same interface later in the development process (at the latest prior to code generation). The component `ExplorationControl`

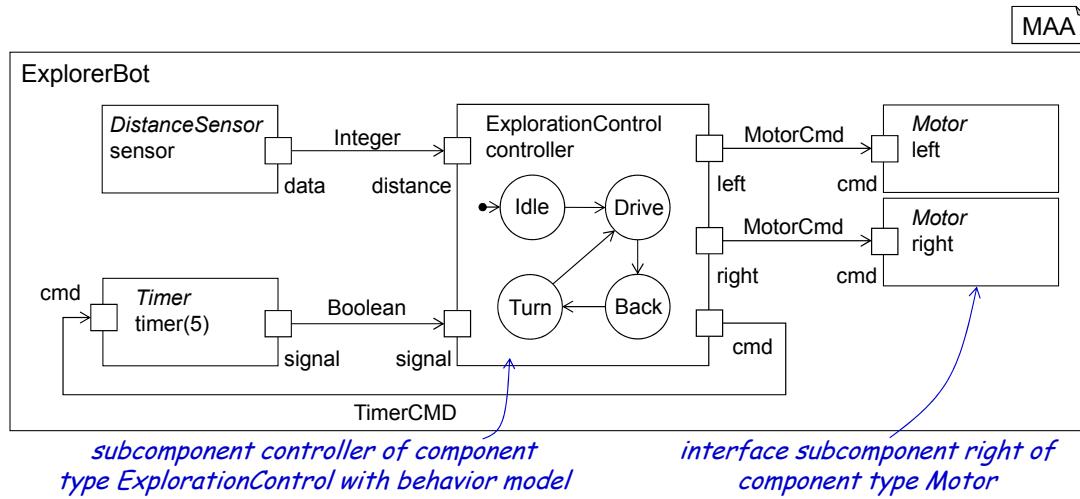


Figure 3.2: The software architecture model ExplorerBot describes the logical, structural software architecture of the system under development and uses interface components as extension points.

is atomic and its behavior is modeled using a variant of UML Statecharts [OMG10]. Thus, the component has no dependencies to the underlying platform, which allows generating implementations for – and to reuse it with – different target platforms. To enable the company’s exploration expert to model component behavior by reusing this language, she integrates it into the ADL without changing it.

Realizing the intended functionality requires the software engineer to develop platform-specific components to replace **DistanceSensor**, **Motor**, and **Timer** for each platform. Therefore, she describes the behavior of these components in a platform-specific GPL, i.e., Java for the Lego NXT robot components and Python for the Pioneer 3-AT robot components. To describe how generated components and their GPL implementations interact, she also develops two run-time environments (RTEs), one for each robot. These define the interfaces of components, realize message passing, and integration of handcrafted component behavior implementations in terms of the respective robot’s GPL. Given these run-time environments, she develops platform-specific component models and RTE-compatible implementations for each component models in the respective GPL. These are the component models **NXTUltrasonicSensor**, **NXTMotor**, and **JavaTimer** for the NXT robot and **Py3UltrasonicSensor**, **Py3Motor**, and **Py3Timer** for the Pioneer 3-AT robot. The behavior implementations of the components **NXTUltrasonicSensor** and **NXTMotor** use the LeJOS API to translate sensor readings into messages, and messages into actuator movement. The implementation of components **JavaTimer** uses Java API time functions to manage a timer and translates timer events into messages and vice versa. For the Pioneer 3-AT robot, the components **Py3Motor** and **Py3UltrasonicSensor** interface with ROS to translate sensor readings into messages, and messages into actuator movement. The implementation **Py3Timer** uses Python functions to manage a timer.

Given the interface components and platform-specific components required for the application as well as the run-time environments, the software engineer needs to provide two code generators: one to translate components (with integrated Statecharts) to Java and one to translate such components to Python. As all functions related to the robots' operating systems are encapsulated into components, the code generators do not have to consider LeJOS or ROS, but generate plain GPL representations of components with Statecharts for the respective languages only. While code generators for Java components and Python components already exist, neither generates code for components with embedded Statecharts. The software engineer thus develops two new code generators that generate Statechart implementations that conform to the interfaces of the respective run-time environment. Their integration into the existing code generation framework is application-specific and defined in an application configuration model.

So far, the software engineer can generate GPL-specific code for both target platforms from the platform-independent software architecture of `ExplorerBot` (cf. Figure 3.2). The resulting code does not interface the platform-specific components developed earlier, as the architecture does not know that these are to be used. Instead of integrating these components manually, which would lead to two new software architectures that have to be developed, evolved, and maintained separately, she describes these replacements in models of a concise application configuration language. These define how such components should be replaced and which code generators to apply. Based on these models, the ADL's model processing framework applies M2M transformations to translate the single platform-independent software architecture into two platform-specific software architectures. For the Lego NXT robot, the transformed software architecture is depicted in Figure 3.3. Here, the platform-independent, interface components of subcomponents `sensor`, `left`, `right`, and `timer` have been replaced with the platform-specific components `NXTUltrasonicSensor`, `NXTMotor`, and `JavaTimer` respectively.

From this platform-specific architecture, executable implementations of the systems can be generated and deployed. With another application configuration model specifying the replacement of interface components with ROS-specific components, the software engineer produces a platform-specific architecture for the Pioneer 3-AT robot. Reusing the same software architecture with other platforms requires minimal effort: if the new platform supports a GPL for which proper code generators exist, the software engineer needs to develop new platform-specific components and to define corresponding replacements only. If new component behavior languages should be integrated, for instance to describe timing functions, sensor data interpretation, or robot movement, these can be reused and integrated easily. They only require proper code generators for embedded models of this language. Finally, a new application configuration model must be created and properly configured.

3.2 Requirements

The software engineering process intended for the development of multi-platform applications with C&C ADLs and exchangeable component behavior languages imposes

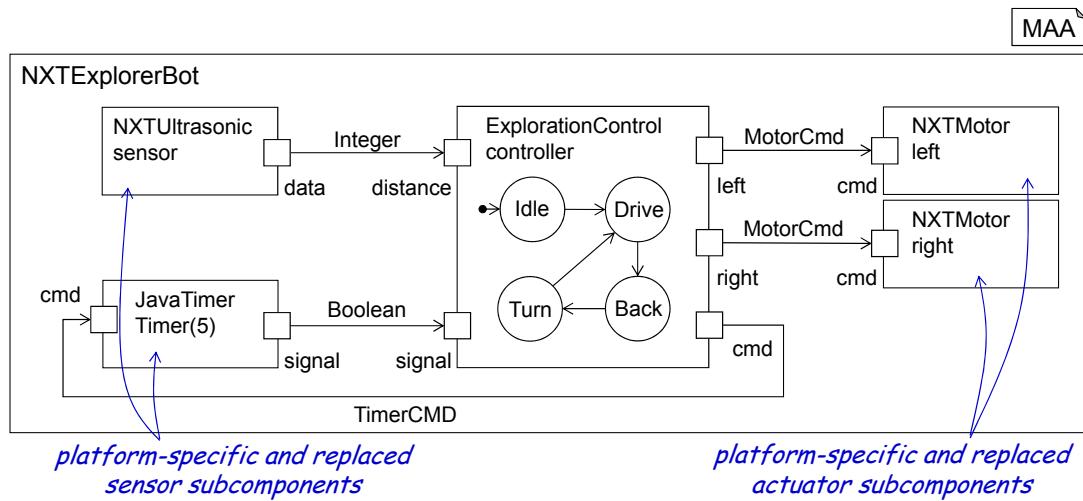


Figure 3.3: The intermediate, platform-specific architecture of the NXTExplorerBot exploration robot.

multiple requirements upon supporting infrastructures. These requirements can be separated into requirements on modeling capabilities (Section 3.2.1) and requirements on model transformation capabilities (Section 3.2.2).

3.2.1 Modeling Requirements

Obviously, most important constituents of model-driven engineering processes are the employed modeling languages. For the intended process, these are the languages to model software architectures, their configuration, and the participating infrastructure constituents. Most of the modeling requirements are derived from experience with describing service robotics applications as illustrated within the scenario.

MRQ-1 Platform-independent architectures: To foster reuse, the infrastructure to develop reusable software architectures supports modeling of logical, platform-independent software architectures.

MRQ-2 Platform-specific architectures: Executing platform-independent software architectures ultimately requires their transformation into platform-specific representations. While smart code generators could produce such, analysis of resulting GPL artifacts is complex and depends on the generator's translation. Transforming platform-independent architectures into valid platform-specific architectures prior to code generation enables such analyses. Hence, all participating modeling languages must support modeling of platform-specific architectures as well.

MRQ-3 Pervasive model-driven engineering: It is possible to model complete, executable applications without requiring handcrafted GPL artifacts.

MRQ-4 Exchangeable modeling languages: Employing the most-appropriate component behavior languages for the application under development is possible.

MRQ-5 Embeddable language fragments: Models usually exist in a well-defined context. For instance, they might expect to operate in the context of specific data sources. Embedded component behavior models may not need such infrastructure as they exist encapsulated in the context of the embedding component. Therefore, it is possible to define which language elements are embedded into the extension points of the ADL, instead of reusing complete languages only.

MRQ-6 Non-invasive language integration Reusing input-output modeling languages to describe component behavior is possible without changing these.

MRQ-7 Platform-independent data types: Reusing a software architecture model can be restricted by the referenced data types: a component relying on a port data type only available for a certain robot platform or in a certain GPL limits the options to reuse the surrounding architecture severely. Therefore, it is possible to describe a software architecture model solely with platform-independent data types.

MRQ-8 Platform-independent software architecture reuse: The concepts realized in structure, data types, and control logic of complex applications can often be formulated independently of the actual target platform. Instead of re-developing the required software architecture and its constituents for each target platform, it is possible to reuse the complete software architecture with different platforms without (platform-specific) modifications and with minimal effort. In detail, this requires:

MRQ-8.1 Additional parametrization: Platform-specific components might require additional configuration information, such as the hardware port a sensor is connected to. Introducing this information to the base software architecture would tie it to specific platforms. Hence, such information is defined besides the platform-independent software architecture.

MRQ-8.2 Parametrization stability: Platform-independent software architectures can specify arguments of subcomponents. These arguments characterize properties of all derivable platform-specific software architectures. Therefore, platform-specific arguments may not overwrite arguments of the platform-independent architecture.

MRQ-8.3 Behavior decomposition: Realizations of platform-specific components might be of arbitrary complexity. Consequently their decomposition is desired.

MRQ-8.4 Architecture validity: The resulting platform-specific architecture is a valid MontiArcAutomaton model, hence the platform-specific behavior replacements for interface components are compatible to their interfaces.

MRQ-8.5 Code generator compatibility: Retaining compatibility with existing code generators [RRW13b], requires architecture integration to be performed completely prior to code generation and may not rely on generator specifics.

MRQ-9 Structural completeness: Usage of partial software architecture structures (such as omitting components to describe extension points) reduces comprehensibility and is a source for errors. Thus, the modeling languages prohibit such structural incompleteness.

3.2.2 Model Transformation Requirements

Translating C&C architecture models with exchangeable component behavior languages also raises requirements regarding model transformations. For instance, the languages to be embedded into components are apriori unknown to code generator developers. Hence, there must be mechanisms to support composition of code generators for such features after deployment. As we furthermore aim at reusing a single software architecture with multiple target platforms, it must be possible to translate that software architecture into representations compatible to these platforms. This section introduces the code generation requirements derived from the scenario and the modeling requirements.

TRQ-1 Arbitrary GPLs and platforms: Complex application domains, such as robotics [Mue13], employ various programming languages and platforms. To support both, code generators for arbitrary GPLs and platforms can be used.

TRQ-2 Integration of handcrafted code: When it is not feasible to describe component behavior platform-independently (for instance when components need to access operating system functionality), handcrafted GPL code can be integrated to describe component behavior as well.

TRQ-3 Code generator reuse: Increasing reuse of code generators demands flexible composition of generator modules. Achieving this requires well-defined code generators and composition mechanisms.

TRQ-3.1 Explicit code generator interfaces: Composing code generators requires formalization of their interfaces in which code generators must explicate their expected inputs, constraints and execution information.

TRQ-3.2 Explicit code generator responsibilities: Each code generator is responsible for a well-defined set of languages and exchangeable without modifications to the other generators.

TRQ-4 Monolithic code generators: There are use cases where monolithic code generators, tied to a limited set of behavior languages and specific target platforms are more useful than compositional generators. Hence, employing such code generators is supported as well.

TRQ-5 Code generation for language fragments: As components may embed fragments of other languages' models (Req. *MRQ-5*), code generators may generate code for such language fragments as well.

TRQ-6 Non-invasive code generator composition: Composition of code generators requires no modification of the participating generators.

TRQ-7 Light-weight code generators: To avoid re-implementation of common M2M transformations for different code generators, the infrastructure supports execution of arbitrary M2M transformations prior to code generation.

TRQ-8 Efficient code generators: To reduce the number of generated artifacts, which, for instance, is crucial when using platforms with small memory, code generators may process component types and produce artifacts representing component types instead of individual subcomponents.

3.3 Methodical Guidance

Software development with the MontiArcAutomaton infrastructure relies on a separation of concerns between software engineering experts for infrastructure extension and domain experts for application engineering. This separation is reflected by its engineering process and constituents. Developing a MontiArcAutomaton application comprises modeling the system's platform-independent architecture with components developed in a variety of behavior languages, adding realizations of platform-specific components, and generating platform-specific GPL artifacts. MontiArcAutomaton differs from established C&C software architecture development frameworks in aiming for a flexible extensibility. To this effect, it allows integration of new component behavior languages into its ADL, automated translation of platform-independent into platform-specific architecture, and composition of code generators. While language engineering, extension, and code generator development require certain software engineering expertise, they need to be performed at different stages of the development process associated with MontiArcAutomaton. Engineering new languages and extending existing languages need to be performed prior to modeling. Code generators can be developed parallel to the software architecture model or subsequently and are integrated in a black-box fashion after deployment. Enabling such a form of software engineering requires concepts, methods, and an appropriate infrastructure that separates concerns not only logically, but also sequentially over the different development activities. To achieve this, the MontiArcAutomaton infrastructure comprises the following elements:

- An extensible ADL based on MontiArc [HRR12] that allows to describe structural properties of C&C software architectures, such as hierarchical components with typed interfaces and their interconnection.
- Concepts to distinguish platform-independent components models from platform-specific component models.

- The UML/P class diagram language [Sch12] for platform-independent data type description.
- An extension mechanism for exchangeable component behavior modeling languages, which enable platform-independent description of component behavior.
- An integrated, state-based, component behavior language based on the I/O^ω automata paradigm [Rin14].
- A model transformation translating platform-independent architectures with extension points into platform-specific architectures with supporting library infrastructure.
- A code generator composition framework to produce executable GPL implementations from component models with embedded behavior models.
- A mechanism to configure concrete applications with selected code generators and libraries.

Figure 3.4 illustrates these with the corresponding roles following [KRV06]. The quintessential elements of MontiArcAutomaton are its modeling languages, its language integration framework, and its model transformations. The infrastructure provides two extension points: its ADL provides an extension point for component behavior languages and its code generation framework supports extension with code generators for different platforms and combinations of behavior languages. This enables reusing both, languages and generators, in different contexts and applications.

Once the first extension point is implemented with languages provided by *language engineers*, *application modelers* can develop C&C software architectures with components embedding these languages. In case the architecture under development is supposed to be platform-independent, the architecture imports *interface components* from interface libraries provided by *interface library providers* to indicate its extension points. The application modeler then describes how these components are replaced with platform-specific implementation library components developed by *implementation library providers*. Implementation libraries realize interface libraries. Therefore, they contain specializations of interface components and corresponding, platform-specific, handcrafted GPL behavior implementations, as well as required data types.

The application modeler defines component replacement in *application configuration models*, which also reference the architecture to process. In case the architecture is supposed to be platform-specific, this is not necessary. Application configuration models also describe which code generators to apply. To this effect, they reference compatible code generators provided by *generator developers*. Generator description models define generator properties important to composition and execution, such as the *run-time environment* the produced artifacts are compatible to. This is achieved by agreeing on RTE interfaces for components and their behavior implementations. Hence, *run-time environment developers* must comprehend details programming of the target platform and

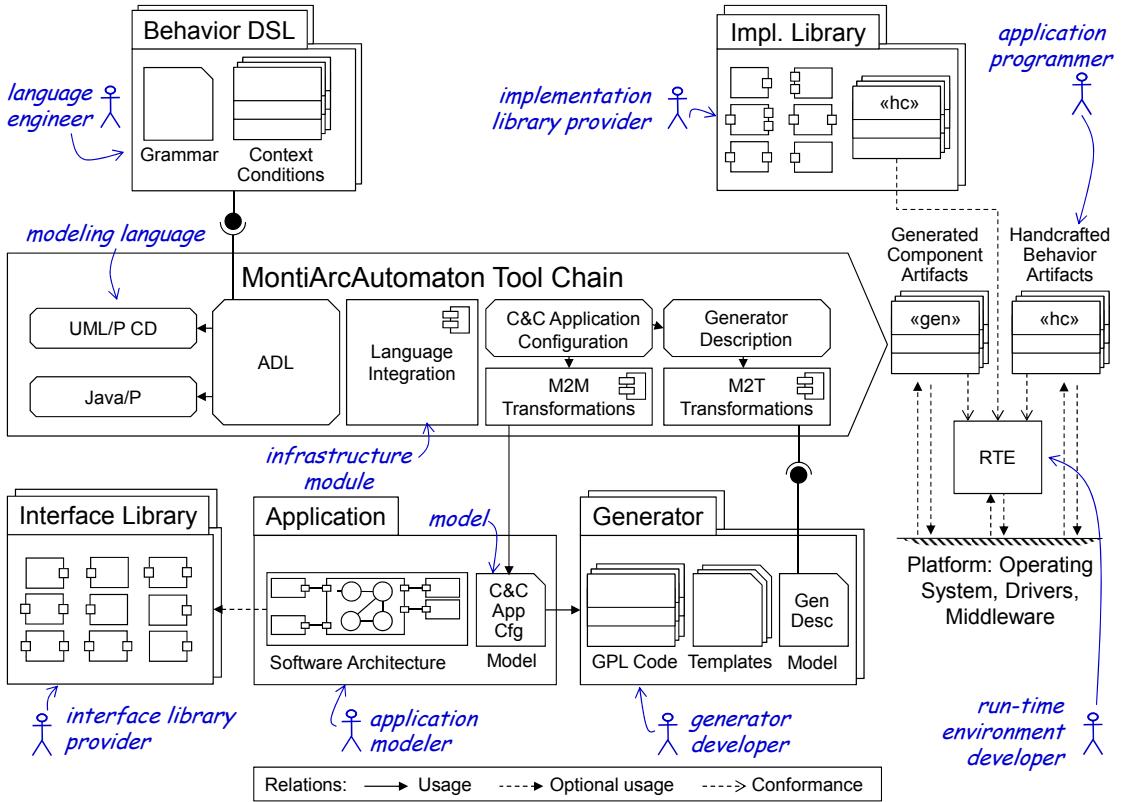


Figure 3.4: Quintessential MontiArcAutomaton infrastructure components, artifacts, and relations with responsible roles.

software engineering details as well. Run-time environments can also contribute solutions for component communication, integration of handcrafted artifacts, and arbitrary additional functionality.

The MontiArcAutomaton infrastructure processes configuration models, configures its component replacement M2M transformation according to it, and passes the selected code generators to its code generator composition framework. The latter combines the generators and uses the resulting aggregate to produce generated component artifacts. Per agreement on an RTE and its interfaces, the generators produce components that are compatible to it. This enables *application programmers* to develop handcrafted behavior implementations for atomic components without behavior models (i.e., to interface the target platform's GPL functionality) without considering the generated code. Instead, the application programmer must comprehend the RTE only.

With these extension points and modules, MontiArcAutomaton enables describing component behavior not conveniently expressible with existing modeling languages with more appropriate behavior languages. These languages must be provided by *language engineers* and can be stand-alone languages that describe some form of input-output behavior (such as Statecharts reading from and writing to data sources). Alternatively,

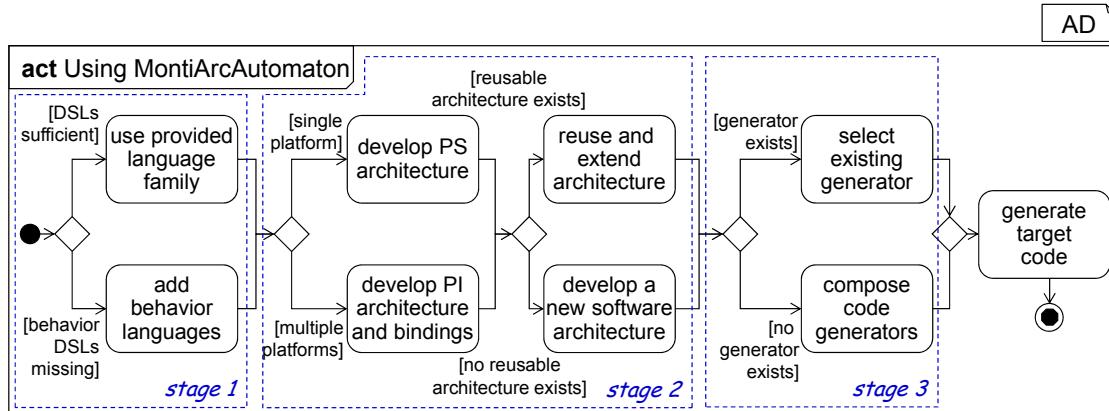


Figure 3.5: The three stages to configure and use MontiArcAutomaton. Each activity can result in failure (for instance due to modeling an invalid architecture or composing incompatible code generators). These transitions are omitted for clarity.

these languages can be designed for embedding into MontiArcAutomaton. In both cases, the language integration infrastructure of MontiArcAutomaton minimizes integration efforts. For each additional language, a *generator developer* then has to provide a code generator to produce executable component implementations according to the RTE of the component generator the new behavior generator should be combined with.

This modularization of MontiArcAutomaton and division of responsibilities to roles addresses the different skill sets of engineers: domain experts may enact the roles of *application modelers* and are liberated from becoming experts in software language engineering, code generator development, or the platform details. The latter expertise can be provided by *implementation library providers*, who produce implementation libraries that can be reused for multiple applications performing on the same platform, and *application programmers*, who produce application-specific components.

Developing an application for a new platform from scratch requires the enactment of the majority of roles defined above and can be separated into the three stages as depicted in Figure 3.5: In the first stage, language engineers extend the MontiArcAutomaton ADL and integrate required component behavior languages. In the second stage, the application modeler develops either a platform-specific software architecture or a platform-independent software architecture and the related model. In this stage, the application modeler also decides whether to reuse an existing architecture or to develop a new architecture from scratch. In the last stage, either a component generator for the specific language aggregate composed in the first stage is (re)used or an appropriate code generator family is composed. Finally, code for the target systems can be generated.

As the activities depicted on top of each stage are subsumed by the respective corresponding bottom activities, the following sections describe the only the bottom activities of each stage.

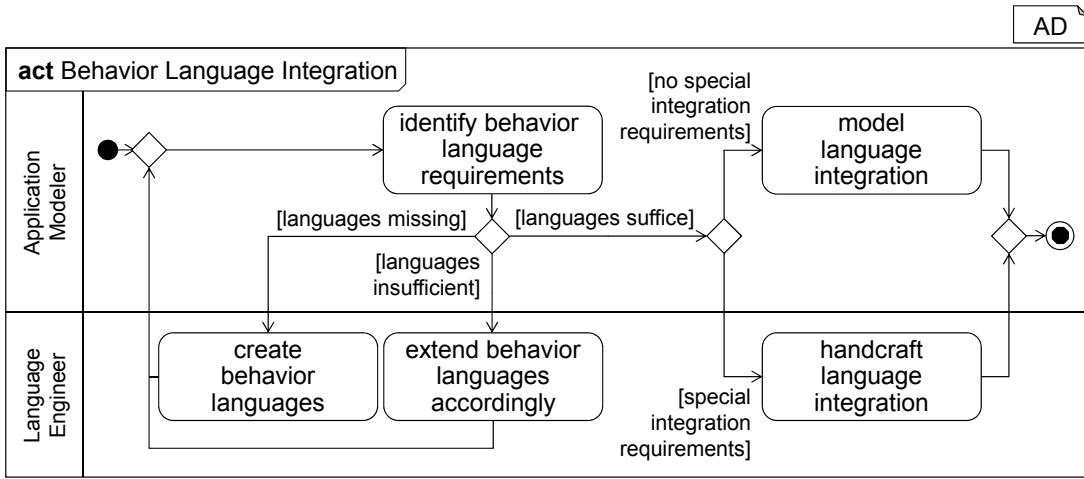


Figure 3.6: Instantiation and configuration of MontiArcAutomaton begins with integration component behavior languages.

3.3.1 Extension with Behavior Languages

MontiArcAutomaton combines at least two kinds of languages to model software architectures: a C&C architecture description language and a data type description language. To enable modeling of platform-independent software components with behavior, it further enables integration of component behavior languages. The configuration and integration efforts to integrate new component behavior languages are part of the behavior language extension stage of MontiArcAutomaton deployment (cf. Figure 3.6).

The application modeler begins this stage identifying the required component behavior languages. If any languages are missing or are insufficient to the requirements, the language engineers need to develop proper component behavior languages or extend existing languages accordingly. In case the languages to be integrated do not yield special integration requirements (such as parts crucial to integration missing), the application modeler can model their integration. Otherwise, the language engineers can handcraft integration using the language embedding features of MontiCore.

Chapter 4 presents the integration of component behavior languages by introducing the MontiArcAutomaton ADL and its extension mechanisms. Chapter 5 afterwards describes the state-based AUTOMATA behavior language and its integration into the MontiArcAutomaton ADL.

3.3.2 Architecture Modeling

After the language family has been extended with component behavior modeling languages, the actual software architectures can be modeled. For platform-specific architectures this is straightforward and the application modeler needs to provide the required components and data types only. For components with modeled behavior the application modeler also defines their behavior as well. In case a component requires a GPL

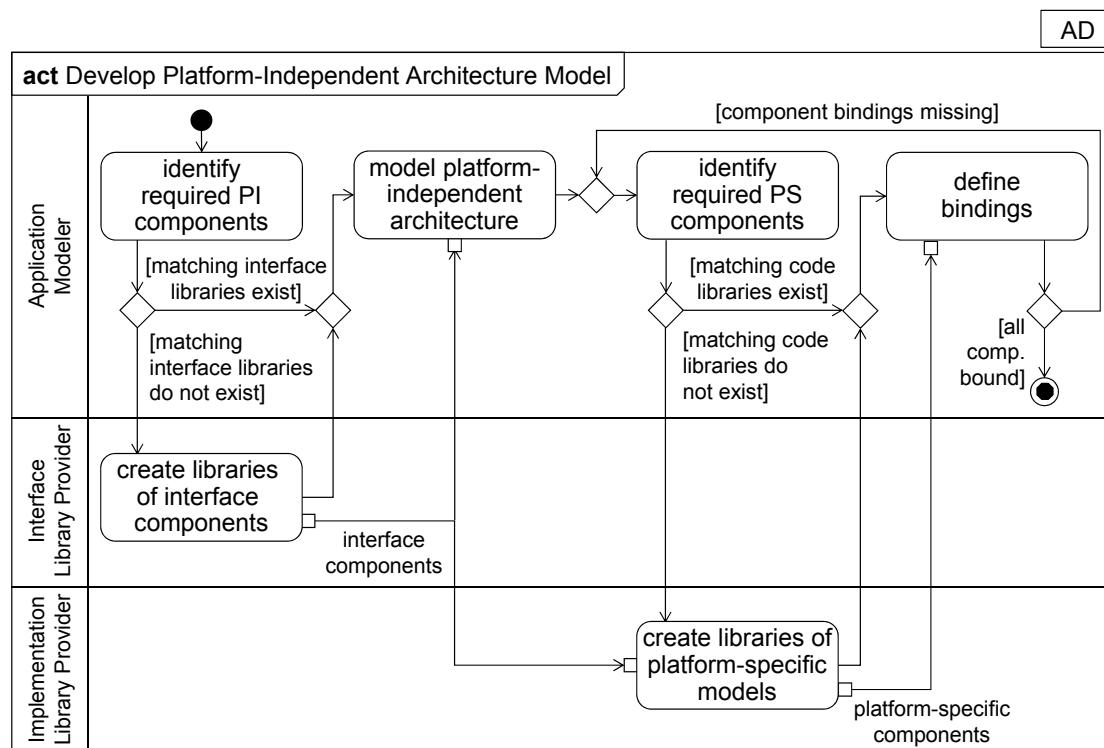


Figure 3.7: The steps involved in modeling a MontiArcAutomaton software architecture differ depending on the intended platform-independence: if the architecture should be platform-independent, it relies on interface components imported from interface libraries and later replaced with platform-specific components from implementation libraries.

behavior implementation, the application programmer implements it accordingly. Afterwards, the architecture can be processed by MontiArcAutomaton to parse it into an AST representation and perform well-formedness checks.

Reusable, platform-independent software architectures may not rely on behavior provided in form of GPL artifacts without restricting reuse to compatible platforms. Therefore, components with behavior not expressible in modeling languages are replaced by *interface components* (Figure 3.7). These components provide the required interfaces to describe the structure of the software architecture, but omit behavior implementations, i.e., they act as architecture extension points. To enable this, the application modeler defines the required interface components as part of interface libraries that are used by the software architecture model. The corresponding platform-specific components are modeled as part of platform-specific implementation libraries and implemented by the implementation library providers. After selecting implementation libraries for each target platform and creating an application configuration model (describing replacement of interface components and selected code generators), code generation can be started. To

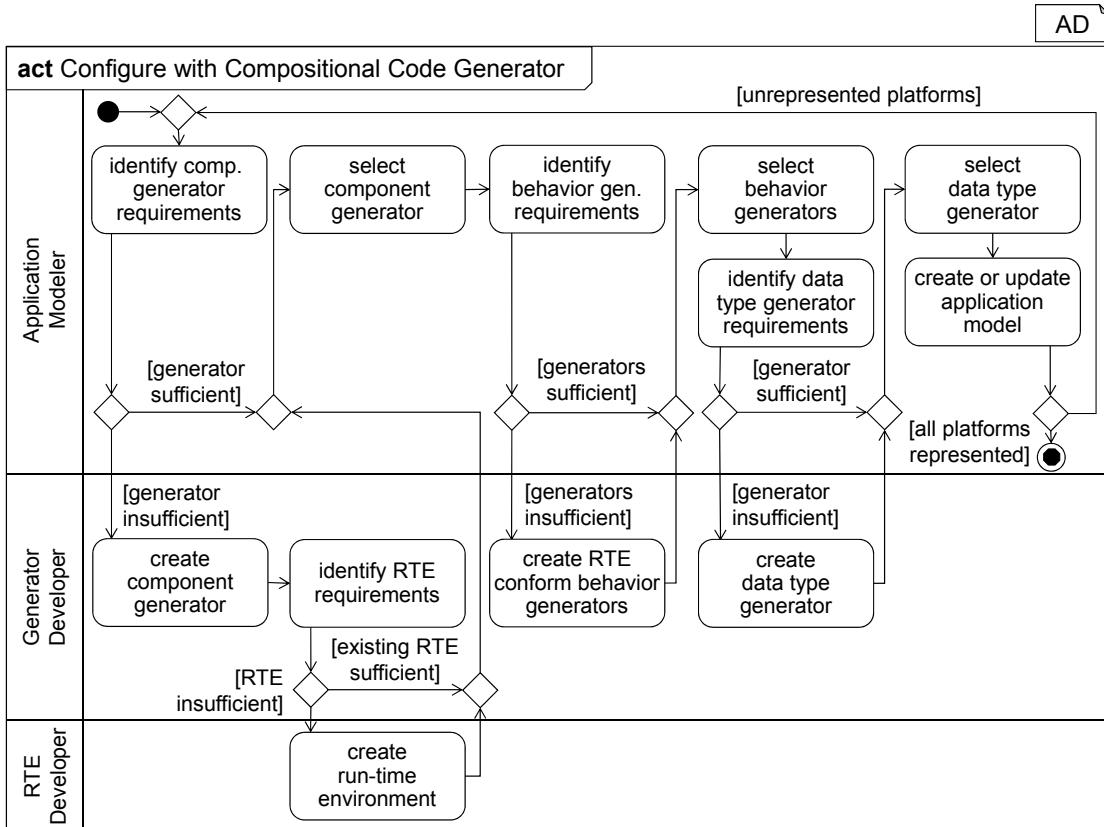


Figure 3.8: Code generators for MontiArcAutomaton either are monolithic (i.e., they contain translations for all participating component behavior languages) and, thus, hardly reusable or they are composed from modular *component generators*, *behavior generators*, and *data type generators*.

this end, MontiArcAutomaton replaces the interface components of the software architecture with corresponding platform-specific components as defined in the application configuration model prior to well-formedness checking and code generation. Chapter 6 describes this stage by introducing the interface libraries, implementation libraries, and the M2M transformation realizing the replacement.

3.3.3 Composed Code Synthesis

Producing executable systems automatically requires code generators that can process the language aggregate of the software architecture model. MontiArcAutomaton enables using monolithic code generators for specific language aggregates as well as composition of modular, reusable, code generators. Composition of modular code generators for C&C architectures requires at least three generator kinds for different concerns:

1. Component generators produce GPL artifacts representing component structure, i.e., ports, variables, messaging infrastructure, and the topologies of composed components.
2. Behavior generators produce GPL artifacts representing models of a single component behavior language.
3. Data type generators produce GPL artifacts representing data types.

Accordingly, this stage begins with the *application modeler* identifying required code generators suitable for the intended use. If a proper monolithic code generator exists, its usage has to be defined in the *application configuration model*.

In case compositional code generators are to be used (Figure 3.8), first a proper *component generator* has to be selected. If no such generator exists, a *generator developer* needs to provide a suitable one. To integrate generated component code with handcrafted and generated behavior code, the component generator relies on a *run-time environment*. If no suitable RTE for the target platform, its GPL, and required functionalities exists, the *run-time environment developer* provides one. Afterwards, for each component behavior language, target platform, and RTE, a suitable *behavior generator* has to be selected as well. This may include involving a *generator developer* to develop proper generators. Finally, data type generators (e.g., for data types of ports) are required as well. Again, if no proper data type generator exists, a *generator developer* needs to provide one for the target platform. After selection of proper code generators, transformation in executable systems can be invoked without effort. Chapter 7 describes this stage by introducing run-time environments, code generator kinds, and a generator description modeling language to describe compositional code generators.

Afterwards, Chapter 8 presents the application configuration modeling language, which enables defining component replacement and to select generators for a software architecture model.

Chapter 4

Component & Connector Architectures with Application-Specific Behavior

*Architecture starts when you carefully put two bricks together.
There it begins.*

Ludwig Mies van der Rohe

Component & connector architecture description languages enable developers to compose complex systems from component models. These models abstract from implementation details of GPLs and provide well-defined interfaces to hide component behavior complexity. Most ADLs, however, require component developers to describe component behavior in form of GPL artifacts tied to components by convention or explicit reference. Thus, although ADLs contribute abstraction to system integration, their usage entails coping with accidental complexities and notational noise nonetheless. That all intended target platforms must support the GPL featured by the ADL furthermore hampers reuse of components with different platforms. Enabling component developers to use modeling languages to describe component behavior facilitates abstraction and reuse.

MontiArcAutomaton is an infrastructure that realizes concepts to integrate (domain-specific) component behavior languages into a MDE process focused on C&C architecture modeling and model transformation. At its core, the infrastructure contains a C&C ADL that describes how structural architecture parts can be modeled and provides and extension points for exchangeable component behavior languages. This MontiArcAutomaton ADL inherits from MontiArc [HRR12], extends its symbol table, and introduces new well-formedness rules to reflect language integration and related modeling elements. As MontiArc enables modeling of platform-specific software architectures, inheriting from it enables MontiArcAutomaton to model such software architectures as well. This fulfills Req. *MRQ-2*. Furthermore, this allows modeling data types in terms of UML/P class diagrams and, hence, also fulfills Req. *MRQ-7*.

This chapter presents the MontiArcAutomaton ADL. To this effect, this section introduces the extensions of MontiArcAutomaton over MontiArc on the example of the software architecture ImprovedBumperBot as depicted in Figure 4.1.

The component type ImprovedBumperBot is a variant of the BumperBot robot (cf. Figure 2.7) that contains the subcomponents distance, clock, controller, and navigation to provide the same functionality. The components Distance, Clock,

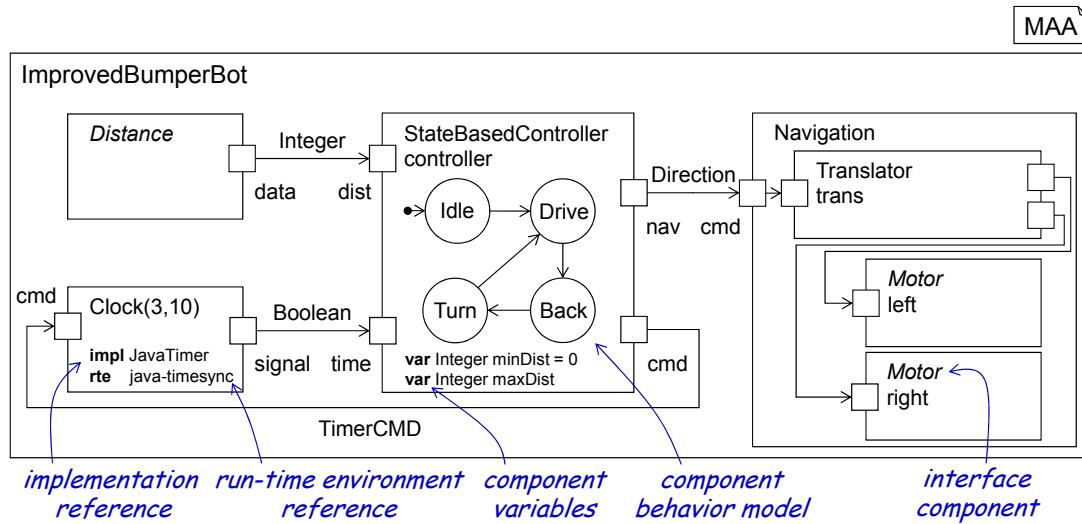


Figure 4.1: A variant of the BumperBot robot illustrating language elements of MontiArc and of MontiArcAutomaton.

StateBasedController, and Motor feature modeling elements specific to MontiArcAutomaton, while the component types Navigation and Translator feature MontiArc language elements only. The component Distance is of interface component kind, and must be replaced with a platform-specific realization prior to code generation. The component Clock employs new component properties to reference its GPL behavior implementation artifacts, as well as the RTE it expects its implementation to be compatible to. The behavior of component StateBasedController is defined in terms of an embedded component behavior model and defines two component variables.

The following Section 4.1 introduces the MontiArcAutomaton ADL with its modeling elements, symbol table, context conditions, and post-processing model transformations. Section 4.2 describes how to embed component behavior languages into the MontiArcAutomaton ADL. Afterwards, Section 4.3 discusses the MontiArcAutomaton ADL and Section 4.4 discusses related modeling languages.

4.1 MontiArcAutomaton Architecture Description Language

The MontiArcAutomaton ADL enables application modelers to describe software architectures as the hierarchical composition of components with encapsulated behavior models. This encapsulation allows logically distributed development and the composition of components separate from their behavior implementations. MontiArcAutomaton exploits this encapsulation and allows embedding of behavior languages into components. This enables the application modelers to use the most suitable behavior language per component. To this effect, the MontiArcAutomaton ADL adopts the structural language elements from MontiArc (cf. Section 2.4) and introduces additional language elements.

Thus, following the language design guidelines of [KKP⁺09], MontiArcAutomaton ADL inherits the elegance of MontiArc and remains “as simple as possible, [yet] as rich as needed” [Gli02] without the overwhelming language complexity of other ADLs.

This section introduces the new language elements and their concrete syntax via small examples in Section 4.1.1. The complete grammar of the MontiArcAutomaton ADL is available in Section A.1 in two versions: One version is meant for documentation and prepared for better comprehension by the reader. The other is the version processed by MontiCore. Afterwards, Section 4.1.2 describes the symbol table structure and symbol table entries of MontiArcAutomaton. Based on these, Section 4.1.4 presents the context conditions of MontiArcAutomaton. Subsequently, MontiArcAutomaton performs two transformations to facilitate further model processing. Section 4.1.5 describes these.

4.1.1 Language Elements

The modeling language elements defined by the MontiArcAutomaton ADL introduce new keywords to the component signature, default parameter values for component configuration parameters, component variables, platform-specific component properties, and an extension point for component behavior. As the MontiArcAutomaton ADL extends MontiArc’s grammar, the language elements to model ports, subcomponent declarations, and connectors that it inherits from MontiArc remain unchanged. Consequently, MontiArcAutomaton inherits the abstract syntax of MontiArc as well.

Interface Components

MontiArc does not enable to model component interfaces decoupled from their internals (behavior or subcomponents). However, the MontiArcAutomaton infrastructure requires such a notion to describe extension points of platform-independent software architectures. Implementing these extension points differently for specific platforms (cf. Section 3.1) facilitates to reuse a single software architecture with different platforms. MontiArcAutomaton thus introduces *interface components*. Interface components may not describe component behavior (neither via composition, nor embedded models, nor via handcrafted GPL behavior implementations) and must be replaced with platform-specific components prior to code generation. As they yield complete interfaces, their usage retains a valid software architecture. This enables modeling the structure of platform-independent software architectures completely and to reuse these as common base for subsequent, platform-specific software architecture as required by Req. *MRQ-1*. The component type `DistanceSensor` depicted in Figure 4.1 and Listing 4.1 is of such an interface component. In its component type definition (l. 1), this is indicated by the keyword `interface` at the beginning of its signature.

Using interface components in a software architecture requires replacing their types prior to well-formedness checking and code generation. MontiArcAutomaton supports replacing their types with component types extending the corresponding interface component to replaced (Chapter 6).

```
1 interface component DistanceSensor {  
2   port  
3     out Integer data;  
4 }
```

MAA

Listing 4.1: The *interface component* DistanceSensor.

Default Parameter Values

MontiArc components may declare configuration parameters to describe the information required for proper component instantiation. Complex component types might require dozens of configuration parameters and instantiation of all parameters is not always necessary. As MontiArc components may yield a single set of mandatory configuration parameters only, this requires engineering components for each required parameter combination. The MontiArcAutomaton ADL reduces this effort by borrowing the notion of *default parameter values* [Oli07] from the Python programming language.¹ Default parameter values enable omitting arguments for component configuration parameters that then use the default values instead. To this end, the component type may specify default values for each configuration parameter under the following restriction: if any configuration parameter yields a default value, all subsequent parameters must yield a default value as well. Otherwise, assigning arguments at component instantiation would require providing arguments for all parameters without default values and preceding parameters (even if they yield default values) or becoming non-deterministic.

```
1 package robots;  
2  
3 component Clock[Integer short, Integer long = 10] {  
4   port  
5     in TimerCMD cmd,  
6     out Boolean signal;  
7  
8   implementation robots.sensors.SecClockImpl;  
9  
10  rte java-timesync;  
11 }
```

MAA

Listing 4.2: The component type *Clock* defines two configuration parameters *short* and *long* of which the latter has the default value 10.

In Listing 4.2, the component *Clock* requires two configuration parameters *short* and *long*. The parameter *short* is mandatory and the parameter *long* is optional with a default value of 10 as indicated by the assignment *long = 10* following its name.

¹The Python 3.0 default parameter value documentation is available at <https://docs.python.org/3/reference/funcdef.html#index-21>

This value is applied whenever a subcomponent instance of `Clock` with only a single argument is declared. Hence, both subcomponent declarations depicted in Listing 4.3 (ll. 2-3) are valid.

```

1 component DoubleClock {
2   component Clock(1,2) clock0;
3   component Clock(1) clock1;
4   // ..
5 }
```

MAA

Listing 4.3: The composed component `DoubleClock` declares two instances `clock0` and `clock1` of component type `Clock`. The former applies two arguments to the parameters of `Clock` and the latter uses its default value for the parameter `long`.

Component Behavior Implementation Reference

For atomic components with handcrafted GPL behavior implementation, MontiArc expects the implementation to reside in the same logical package² as the component type under a name consisting of the components name with suffix `Impl`. As this assumption prohibits directly exchanging component implementations and hinders reuse, MontiArcAutomaton introduces the keyword `implementation`. With this, application programmers can develop component types that reference GPL behavior implementations existing in different packages or under different names. For instance, the component type `Clock` of Listing 4.2 references the GPL behavior implementation `robots.sensors.SecClockImpl` (l. 8) that performs its computations. References to component implementations are bequeathed to inheriting components and override the implicitly expected behavior following MontiArc's naming conventions.

Component Variables

A central feature of MontiArcAutomaton is the integration of component behavior modeling languages. Experimenting with various languages [RRW13c, RRW15b] has shown that most languages require some form of variables to store states for performing complex calculations. To facilitate development of behavior languages for the MontiArcAutomaton ADL, it introduces variables to components. Variables follow the notions of component configuration parameters and ports, i.e., they are defined by a type and a name and employ similar well-formedness rules. The component type `StateBasedController` (Listing 4.4) defines two component variables `min` and `max`, both of type `Integer` (ll. 8-9). The variable `min` is preceded by the keyword `var`, which is optional. Also, initialization of variables (l. 8) is optional as well. If omitted, a variable's value is undefined. The types of MontiArcAutomaton component variables

²Interpretation of packages with different GPLs is within the generator developers' duties.

rest on MontiArc's type system and may be of arbitrary complexity employing array dimensions and generic type parameters. Variables are locally visible in the defining component only and can be referenced by its behavior model.

```
1 component StateBasedController {
2   port
3     in Integer dist,
4     in Boolean time,
5     out Direction nav,
6     out TimerCmd cmd;
7
8   var Integer min = 0;
9     Integer max;
10
11  behavior automaton ControllerAutomaton {
12    // Embedded behavior language model
13  }
14 }
```

MAA

Listing 4.4: The atomic component `StateBasedController` contains the two component variables `min` (l. 8) and `max` (l. 9) of type `Integer`.

Run-Time Environment Reference

Proper integration of platform-specific atomic components (i.e., components tied to a GPL behavior implementation) requires integration of their model into the software architecture under development as well as integration of their GPL behavior implementation into the generated system. The latter requires means to describe which implementations of components, ports, and connectors the component GPL behavior implementation is compatible to.

We identify the set of artifacts that represent components and related concepts in a GPL as a run-time environment and platform-specific MontiArcAutomaton components may indicate which run-time environment their implementations conform to. Explicating this is crucial for code generator composition to ensure proper integration of generated code with handcrafted code. Declaration of the required run-time environment is part of the component body (delimited by curly brackets) and begins with the keyword `rte` followed by a name. Component `Clock` (Listing 4.2) declares that the required run-time environment for its implementation is `java-timesync` (l. 10). References to run-time environments are bequeathed to inheriting components also.

Component Behavior Model

The MontiArcAutomaton ADL provides an extension point for component behavior language models within atomic components. This extension point starts with the keyword `behavior`, followed by an identifier for the behavior's kind, an optional name, and the

behavior's content in curly brackets. The component type `StateBasedController` (Listing 4.4), for instance, contains an automaton model to describe its behavior. Hence, its body contains a behavior block (ll. 11-13) that starts with the keyword `behavior`, followed by the identifier `automaton`, the automaton's name `ControllerAutomaton`, and a block containing the actual automaton model. As multiple behavior modeling languages can be embedded, the identifier `automaton` is required to distinguish embedded models and improves parsing (cf. Section 2.2.2). The language engineer integrating a behavior language can assign an arbitrary identifier for her languages as long as it is unique. Specifying a behavior model overrides MontiArc's expectation of a GPL behavior artifact of the component's name.

Language Elements of Different Component Kinds

With the introduction of interface components and the distinction between platform-independent and platform-specific components, the MontiArcAutomaton ADL extends MontiArc's dichotomy of atomic and composed component kinds. Essentially, interface components correspond to component interfaces of other ADLs, i.e., they are atomic, do not describe behavior and are intrinsically platform-independent. Platform-independent component kinds can be atomic or composed. They must avoid references to GPL behavior implementations and run-time environments. Platform-specific components may actually be platform-independent, but do not guarantee this, hence they may be atomic or composed, refer to GPL behavior implementations, and run-time system, but must avoid the keyword `interface`. Overall, MontiArcAutomaton introduces two new component kinds: interface components and atomic components with behavior models. Table 4.5 describes the component kinds of MontiArcAutomaton.

4.1.2 Symbol Table

MontiArcAutomaton extends MontiArc with component behavior, component variables, default parameters, interface components, and references to component implementations as well as to run-time environments. To make this information amenable to well-formedness checking and language integration, the MontiArcAutomaton ADL symbol table must store this information. To this end, the MontiArcAutomaton ADL extends the MontiArc symbol table entries for components as depicted in Figure 4.2. These entries encapsulate the essence of C&C structures in form of entries for components, connector, ports, and related entities.

4.1.3 MontiArcAutomaton Symbol Table

The MontiArcAutomaton ADL inherits its grammar as well as symbol table infrastructure from MontiArc. The symbol table entries inherited from MontiArc describe important properties of structural modeling elements as depicted in Figure 4.3.

³In the transitive closure of its subcomponent relation.

Table 4.5: Component kinds of the MontiArcAutomaton ADL.

Component Kind	Required Properties	Prohibited Properties	Platform Dependence
Composed	Sub-components	Behavior models, GPL behavior implementation references, RTE references, component variables	Platform-specific if it contains a platform-specific subcomponent ³
Atomic with behavior model	Behavior model	Subcomponents, GPL behavior implementation references, RTE references	Platform-Independent
Atomic without behavior model	RTE reference	Subcomponents, behavior models, component variables	Platform-Specific
Interface component	Keyword “interface”	Subcomponents, behavior models, component variables, GPL behavior implementation reference, RTE references	Platform-Independent

Component entries represent component type definitions and store related information. This comprises the component type the represented component type inherits from as well as its subcomponents, connectors, ports, inner components, type parameters, and configuration parameters. References to the inherited component type and to subcomponents are stored in component reference entries that represent instantiated component types, i.e., contain the required configuration arguments and type arguments. Ports are stored in separate entries as well. Type entries represent MontiArc’s data types and resemble types of UML/P class diagrams [Sch12]. A type has a name and may feature an array dimensionality, generic type parameters, and fields. The latter are field entries with types again. Fields entries resemble members of UML/P classes and provide a multitude of qualifiers, such as, for instance, visibilities. These types can be adapted to types of other languages, such as the types of embedded behavior languages, to check the validity of embedded models. Value entries contain a name, the string representation of a value, and a TypeEntry.

MontiArcAutomaton extends MontiArc’s component entry to store additional information, introduces variable definitions to represent variables with their optional initial values, and behavior entries to represent information about component behavior models. Thus, the MontiArcAutomaton ADL extends the MontiArc symbol table with the entry types MAAComponentEntry and VariableEntry as depicted in Figure 4.3, where the entry kind of MAAComponentEntry entries is the same as for MontiArc component entries. MAAComponentEntry entries hold a map from default parameter names to their values (via qualifier defaultParameter), two strings representing the references to component implementation and run-time environment, a Boolean flag indicating whether the component type this entry represents is an interface component, and

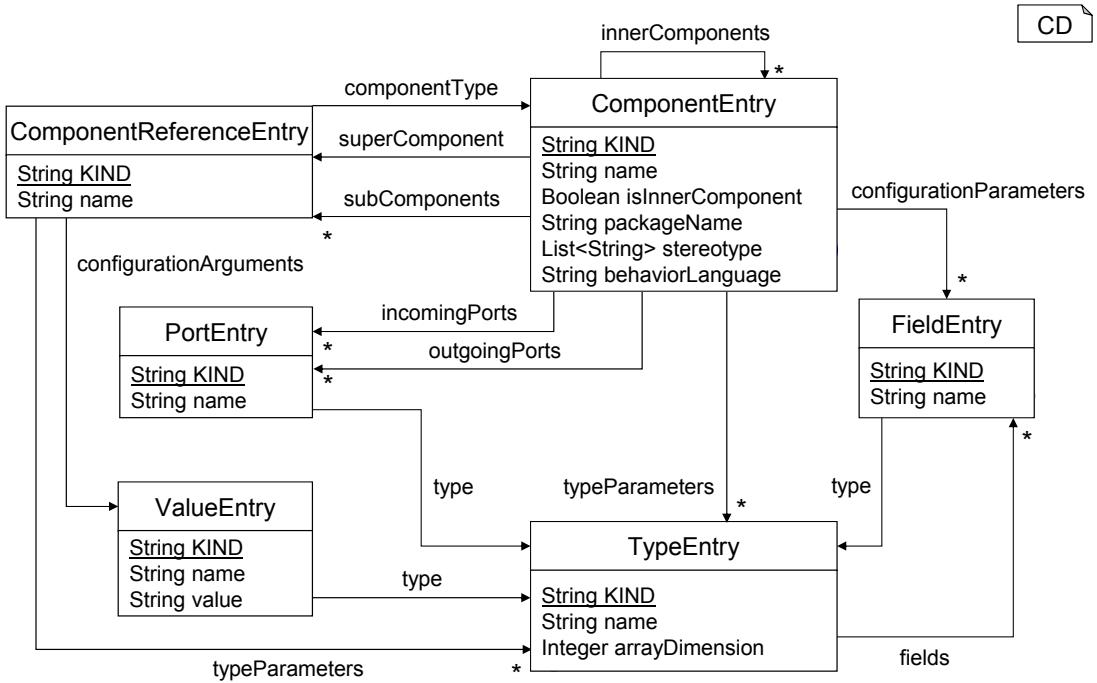


Figure 4.2: An excerpt of the entries of the MontiArc symbol table that represents structural aspects of C&C software architectures.

an optional⁴ string representing its behavior model’s language fragment. Additionally, it inherits all members of MontiArc’s **ComponentEntry**. Variable entries extend the field entries of MontiArc with a member `value` of MontiArc’s **ValueEntry** type, to store their values.

The MontiArcAutomaton ADL also provides a new workflow to create symbol table entries (cf. Figure 2.3). This workflow extends the symbol table creation workflow of MontiArc and produces **ComponentEntry** entries instead of MontiArc component entries. In consequence, resolving a component entry by kind and name yields a **MAAComponentEntry** instead of a **ComponentEntry**. Following Liskov’s substitution principle [Lis87], instance of **MAAComponentEntry** can substitute **ComponentEntry** instances wherever required and thus the complete well-formedness checking and the MontiArc symbol table are reused.

4.1.4 Context Conditions

The modeling elements of the MontiArcAutomaton ADL characterize the parseable models. MontiArcAutomaton, however, can produce executable systems only from well-formed models. The context conditions of the MontiArcAutomaton ADL restrict the MontiArcAutomaton architectures to well-formed MontiArcAutomaton ADL models

⁴Optionality is represented using `java.util.Optional` of Java 1.8.

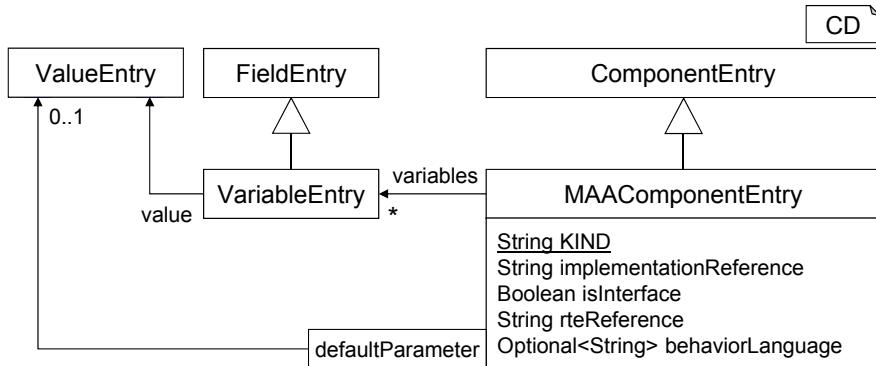


Figure 4.3: The most important MontiArcAutomaton symbol table entry types with their quintessential features.

(e.g., without references to missing component types). Therefore, after parsing and creating both AST and symbol table, MontiArcAutomaton checks the well-formedness of each input model and raises a warning or rejects the model. The MontiArcAutomaton ADL models are checked by *context conditions* that are either inherited from MontiArc or introduced with the MontiArcAutomaton ADL. This section presents four types of context conditions following [RRW14a] as introduced with the MontiArcAutomaton ADL. The context conditions inherited from MontiArc are presented in [HRR12] and all valid MontiArc models are MontiArcAutomaton ADL models as well.

Uniqueness Conditions

Uniqueness conditions ensure that language elements of a certain type only occur once per model and that the names of language elements are unique. MontiArc, for instance, ensures that the names of configuration parameters, ports, subcomponents are unique per component type definition [HRR12]. Ambiguous names produce ambiguous behavior. Hence, all uniqueness context conditions raise errors that prohibit further processing. MontiArcAutomaton inherits the uniqueness conditions of MontiArc presented in Section 3.1 of [HRR12].

MU1: The name of each component variable is unique among ports, variables, and configuration parameters.

All component variables coexist in the namespace of the component together with its ports and configuration parameters. Therefore, their names must be unique. Listing 4.5 shows four errors: The first two errors arise from defining a component configuration parameter and a port of the same name *distance* (ll. 1-3). The subsequent two errors arise from defining a port and a variable of the same name *min* (ll. 4-6).

```

1 component Recorder[int distance] { ✗ // Ambiguous name.
2   port
3     in int distance; ✗ // Ambiguous name.
4     in int min; ✗ // Ambiguous name.
5
6   var int min; ✗ // Ambiguous name.
7 }
```

MAA

Listing 4.5: The component Recorder defines a configuration parameter (l. 1) and a port (l. 3) of name distance, as well as port (l. 4) and variable (l. 6) of name min. Both are prohibited.

MU2: Each atomic component contains at most one behavior model.

Atomic components may contain a single behavior model at most. If an atomic component contains no behavior model, the MontiArcAutomaton infrastructure expects a handcrafted GPL behavior implementation - either following MontiArc's naming conventions or as referenced by its component implementation property.

```

1 component BehaviorController {
2
3   behavior statechart CautiousBehavior { ✗ // Redundant
4     // ...
5   }
6
7   behavior statechart PressingBehavior { ✗ // Redundant
8     // ...
9   }
10 }
```

MAA

Listing 4.6: The atomic component BehaviorController contains two behavior models (ll. 3-9).

Defining more than one behavior model in a component is redundant as MontiArcAutomaton does currently not support switching between behavior models. The issued error resulting from defining two behavior models in a single atomic component is depicted in Listing 4.6. Here, the component type BehaviorController contains the two behavior models CautiousBehavior (ll. 3-5) and PressingBehavior (ll. 7-9).

MU3: Atomic components reference at most one GPL behavior implementation.

Similarly to defining multiple behavior models, referencing multiple GPL behavior implementations is prohibited. Listing 4.7 shows the resulting errors.

```

1 component MultiSensor<T> {
2   port
3     out T data;
4
5   implementation UltrasonicImpl; ✗ // Multiple implementations.
6
7   implementation ColorSensorImpl; ✗ // Multiple implementations.
8 }
```

MAA

Listing 4.7: MultiSensor references the two component implementations UltraSonicImpl (l. 5) and ColorSensorImpl (l. 7).

MU4: Atomic components either contain a behavior model or reference a GPL behavior implementation.

Atomic components that contain a component behavior model while declaring a GPL behavior implementation introduce an ambiguity as the source of their behavior is underspecified. Consequently, the MontiArcAutomaton ADL prohibits such models. The reference to a GPL behavior implementation may be implicit. In this case, the name of the behavior implementation artifact is derived from the component's name. This ensures compatibility with MontiArc [HRR12] models.

MU5: Atomic components reference at most one run-time environment.

Although it is possible that a handcrafted GPL behavior implementation of an atomic component conforms to more than one run-time environment, working with multiple run-time environments in the same application is prohibited. Therefore, atomic MontiArcAutomaton ADL components may reference only a single run-time environment. Referencing multiple run-time environment produces the error depicted in Listing 4.8, where component type MultiSystemDistanceSensor references the RTE java-timesync (l. 5) and the RTE python-timed (l. 6).

```

1 component MultiSystemDistanceSensor {
2   port
3     out Integer distance;
4
5   rte java-timesync; ✗ // Multiple run-time environments.
6   rte python-untimed; ✗ // Multiple run-time environments.
7 }
```

MAA

Listing 4.8: The component type MultiSystemDistanceSensor references two run-time environments (ll. 5-6).

For components with behavior models, the declaration of RTE conformance is prohibited. How the component is translated is subject to the processing code generator.

Convention Conditions

Convention conditions check the well-formedness of names. Similar to most GPLs that impose certain naming conventions on their constituents (for instance, Java classes should begin with an upper-case letter), MontiArc and MontiArcAutomaton impose such naming conventions to improve model comprehensibility as well.

MontiArcAutomaton reuses all convention conditions of MontiArc and adds new convention conditions regarding variable names and behavior names. The convention integrity conditions that MontiArcAutomaton inherits from MontiArc are presented in Section 3.4 of [HRR12]. Violation of convention conditions produces warnings.

MC1: Variable names begin with a lower-case letter.

Following the conventions for ports and configuration parameters, component variable names should start with a lower-case letter as well. Listing 4.9 illustrates the resulting warning for the atomic component `HistogramPrinter`, which contains a variable `History` of type `collections.List<T>`. As MontiArcAutomaton expects variables to start with a lower-case letter, it emits a warning.

```
1 component HistogramPrinter<T> {
2   port
3     in T data;
4
5   collections.List<T> History; ⚠ // Variable names
6                               // start lower-case.
7 }
```

MAA

Listing 4.9: The component type `HistogramPrinter` declares a variable of name `History` (l. 5) which produces a warning regarding its name.

MC2: Behavior model names begin with capital letters.

Component behavior models are considered singleton instances as well. Hence, the optional names of component behavior models should start with capital letters. The component behavior model of the atomic component `RobotController`, as illustrated in Listing 4.10, contains a behavior model of type `fsm` and name `controller` (ll. 2-4), which violates this context condition. Consequently, MontiArcAutomaton raises the depicted warning.

```

1 component RobotController {
2   behavior fsm controller {  // Behavior names should
3     // ...                                // begin with capital letters.
4   }
5   // ...
6 }
```

MAA

Listing 4.10: The behavior implementation of `RobotController` (l. 2) violates the context condition to start its name with an upper-case letter.

Referential Integrity Conditions

Referential integrity context conditions check the well-formedness of references to language elements. This includes checking whether subcomponent declarations provide enough type arguments and configuration arguments, whether referenced ports exist, and whether referenced elements are of expected types. Violation of referential integrity context conditions produces errors. The MontiArcAutomaton ADL inherits all referential integrity conditions from MontiArc are described in Section 3.3 of [HRR12]. Only its context condition on subcomponent declaration is relaxed regarding component configuration parameters with default values (cf. Section 4.1.1).

MR1: Arguments of configuration parameters with default values may be omitted during subcomponent declaration.

MontiArcAutomaton introduces default parameter values to component configuration parameters. Subcomponent declarations thus may omit arguments for parameters with a default value (Section 4.1.1). Therefore, the context condition R9 of MontiArc [HRR12], which requires that all configuration parameters of a component type have to be assigned during subcomponent declaration, is replaced. Instead, MontiArcAutomaton requires subcomponent declarations to give arguments for all configuration parameters without default values only. Nonetheless, a valid MontiArc model, which correctly defines possible arguments for all parameters of a subcomponent declaration (even those with default values), remains a valid MontiArcAutomaton ADL model. Regarding the component type `Clock` (Listing 4.2), with the mandatory configuration parameter `short` and the optional configuration parameter `long`, the composed component `ClockWork` depicted in Listing 4.11 is erroneous: The subcomponent declaration of `orange` (l. 4) provides too few arguments. The subcomponent declaration of `red` (l. 5) provides too many arguments.

MR2: No initial values for variables of pure generic types.

Checking the compatibility of initial assignments to component variables of generic types is impossible without component instantiation information. Consequently, Monti-

```

1 component ClockWork {
2   component Clock(1) green;
3   component Clock(1,2) yellow;
4   component Clock orange; ✗ // Too few arguments.
5   component Clock(1,2,3) red; ✗ // Too many arguments.
6   // ...
7 }
```

MAA

Listing 4.11: Component `ClockWork` declares subcomponents of type `Clock` with too few arguments (l. 4) and too many arguments (l. 5).

ArcAutomaton prohibits such assignments. Listing 4.12 illustrates the error resulting from assigning the value 255 to component variable `lastReading` of component type `RGBSensor`. Please note that this does not prohibit initial values for variables of complex types that employ generic types (such as `Set<T>`).

```

1 component RGBSensor<T> {
2   port
3     out T data;
4
5   T lastReading = 255; ✗ // Assigning to generic type.
6   // ...
7 }
```

MAA

Listing 4.12: The component type `RGBSensor` provides a generic type parameter `T` and contains a port `data` (l. 3) and variable `lastReading` of type `T` (l. 5). Assigning initial values to `lastReading` thus is prohibited.

MR3: No default values for configuration parameters of purely generic types.

Arguments for generic type parameters are assigned in subcomponent declarations. Component configuration parameters may refer to the component's generic types. However, checking the compatibility of a default value assigned to a configuration parameter of generic type is impossible without component instantiation information as well. Thus, the component type defining such an assignment cannot be checked at design time. Hence, MontiArcAutomaton prohibits such assignments as well.

The component type `MaxMotor` depicted in Listing 4.13 provides the configuration parameter `max` with default value 10 (l. 1). The parameter is of generic type. Hence, assigning a default value is prohibited.

```

1 component MaxMotor<T>[T max = 10] { ✗ // Assigning to
2   port                                // generic type.
3     in T command;
4   // ...
5 }
```

MAA

Listing 4.13: The component MaxMotor uses the generic type parameter T as data type of its configuration parameter max and assigns a default value of 10 to it.

MR4: All mandatory component configuration parameters precede the parameters with default values.

Component types may define default values for each configuration parameter (Section 4.1.1). However, if any configuration parameter defines a default value, all following parameters must define a default value as well. Otherwise, assigning arguments at component instantiation requires more complex argument mapping semantics that might become confusing or even non-deterministic.

The component type Validator illustrates this issue with three configuration arguments (ll. 1-3): the first parameter, min has a default value of 0, the second parameter avg has a default value of 1, and the third parameter, max, is mandatory. Instantiating the component as Validator(10) leaves open whether 10 should be mapped to min, avg, or max. Similarly, the mapping semantics of Validator(10, 20) are confusing. Consequently, MontiArcAutomaton produces the error depicted.

```

1 component Validator[int min = 0,
2                           int avg = 1,
3                           int max] { ✗ // Invalid parameter order.
4   // ...
5 }
```

MAA

Listing 4.14: The component Validator defines two configuration parameters. The first two parameters feature a default value but the third does not.

Type Correctness Conditions

Type correctness context conditions check the correct usage and combination of typed elements, including components, parameters, ports, and variables. For the type correctness conditions inherited from MontiArc see Section 3.1 of [HRR12]. The type correctness conditions of MontiArcAutomaton are illustrated below. Their violation generally produces errors.

MT1: Interface components prescribe no component behavior.

Interface components describe extension points of platform-independent software architectures that need to be replaced with compatible, platform-specific components prior to code generation. Replacing composed components or atomic components with a behavior is not desired: replacing composed subcomponents may change the software architecture dramatically and deviate from its original intentions. Replacing atomic components with behavior model replaces behavior intended to be common for the derivable architectures. To this effect interface components are prohibited to comprise behavior models or references to GPL behavior implementations.

```

1 interface component Inverter { x // Interface component
2   port                               // with behavior model.
3     in Integer input,
4     in Integer output;
5
6   behavior statechart {
7     // ...
8   }
9 }
```

MAA

Listing 4.15: The interface component `Inverter` contains a behavior model (ll. 6-8).

The interface component Listing 4.15 illustrates the error resulting containing a behavior model (ll. 6-8). Similarly, Listing 4.16 illustrates the error resulting from defining composed interface components. The component `Clocks` is declared to be of interface kind (l. 1) but contains two subcomponents (ll. 2-3), which is prohibited.

```

1 interface component Clocks { x // Composed interface
2   component Clock(1,2) clock0; // component.
3   component Clock(1) clock1;
4   // ..
5 }
```

MAA

Listing 4.16: The interface component `Clocks` is composed as it contains two subcomponents (ll. 2-3), which is prohibited.

MT2: Only platform-specific, atomic components declare a run-time environment

Composed components, atomic components with behavior models, and interface components are independent of the target GPL. Their translation is solely up to the employed code generators. Hence, they do not need to be tied to a run-time environment. Instead, this even restricts their applicability without necessity. As MontiArcAutomaton aims to increase reuse, this consequently is prohibited as depicted in Listing 4.17.

Here, the atomic component `AtomicController` contains a behavior model (ll. 3-5) but also declares a run-time environment (l. 6). Thus MontiArcAutomaton raises the depicted error.

```
1 component AtomicController {  
2   // ...  
3   behavior petrinet {  
4     // ...  
5   }  
6   rte java-timesync; ❌ // Components with behavior models  
7 }                                // may not declare run-time environments.
```

MAA

Listing 4.17: The component `AtomicController` is atomic and contains a behavior model (ll. 3-5), but declares a run-time environment (l. 6).

MT3: Only atomic components may declare component variables.

Component variables describe part of a component's state to support component behavior modeling. Allowing variables in composed components could be misused to describe the shared state of its subcomponents and hence is prohibited. Listing 4.18 illustrates the resulting error with component type `SensorArray`, which is composed and defines the variable `threshold` (l. 5). This consequently produces an error.

```
1 component SensorArray {  
2   component Ultrasonic;  
3   component ColorSensor;  
4  
5   int threshold; ❌ // Variable in composed component.  
6   // ...  
7 }
```

MAA

Listing 4.18: The composed component `SensorArray` erroneously declares a variable (l. 5).

MT4: Interface components only inherit from interface components.

Interface components describe platform-independent extension points of the architecture in terms of comprise ports, configuration parameters, and generic type parameters only. Platform-specific components may inherit from interface components to concretize these. Interface components that inherit from platform-specific types would inherit the parent's types platform-specific properties (such as RTE or GPL behavior reference) and thus would be platform-specific themselves. Therefore, MontiArcAutomaton prohibits such inheritance. Listing 4.19 illustrates this with two compo-

ment types NXTUltrasonic (ll. 1-7) and Sensor (ll. 9-12). The component type NXTUltrasonic is concrete in being atomic and referencing a GPL component implementation (l. 5) as well as a run-time environment (l. 6). The interface component Sensor extends NXTUltrasonic (ll. 9-10), which produces an error.

```

1 component NXTUltrasonic {
2   port
3     out Double distance;
4
5   implementation nxt.sensors.Ultrasonic;
6   rte java-timesync;
7 }
8
9 interface component Sensor
10  extends NXTUltraSonic {  // Interface type extends
11    // ...                                // platform-specific type.
12 }
```

MAA

Listing 4.19: Component NXTUltrasonic (ll. 1-7) references an implementation (l. 5) and declares a RTE (l. 6), i.e., it is platform-specific, but the inheriting component Sensor (ll. 9-12) is an interface component.

MT5: Inheriting platform-specific, atomic components override the component implementation reference of their super types.

Platform-specific atomic components must override the component implementation reference of their parents. It is possible to have two component types with the same GPL behavior implementation (which entails reading from the same incoming ports, respecting the same parameters, and writing to the same outgoing ports). However, this creates multiple component types for the exact same behavior and is hardly useful. Consequently, MontiArcAutomaton prohibits this by enforcing inheriting component types to override their parents' component implementation references. The component type RegulatedMotor (ll. 7-10) of Listing 4.20 illustrates this problem: it inherits from the component type ROSMotor (ll. 1-5) that references a GPL behavior implementation (l. 4), but does not override this property. Thus, the port maxSpeed (l. 9) introduced by RegulatedMotor will not be considered by the GPL behavior implementation of ROSMotor as the latter is unaware of this port. Consequently, either maxSpeed is superfluous or the component will not work as expected.

MT6: Initial assignments to variables conform to their type.

Initial assignments to variables of concrete types have to conform to the variable's type. This applies to assigning values as well as to references as depicted in Listing 4.21. Here,

```

1 component ROSMotor {
2   port
3     in Float speed;
4   implementation nxt.sensors.Ultrasonic;
5 }
6
7 component RegulatedMotor extends ROSMotor { ✖ // Implementation
8   port                                         // reference not
9     in Float maxSpeed;                      // overridden.
10 }
```

MAA

Listing 4.20: The component `RegulatedMotor` (ll. 7-10) extends from `ROSMotor` (ll. 1-5), but does not override its implementation reference.

the component type `Logger` contains two variables: `bufferSize` of type `Integer` (l. 2) and `logToFile` of type `Boolean` (l. 3). It initializes the variable `bufferSize` with the Boolean value `true`, which produces an error. Furthermore, it initializes the variable `logToFile` with a reference to the `String` configuration parameter `prefix`. Hence, it produces a second error.

```

1 component Logger[String prefix] {
2   Integer bufferSize = true; ✖ // Incompatible type.
3   Boolean logToFile = prefix; ✖ // Incompatible type.
4   // ...
5 }
```

MAA

Listing 4.21: The component type `Logger` defines two variables with initial values incompatible to their types (ll. 2-3).

MT7: Default values of parameters conform to their type.

Similar to variable assignments, assigning default values to configuration parameters has to respect the parameters' types. The component type `IMotor`, illustrated in Listing 4.22, provides a configuration parameter `max` (l. 1) of type `int` with default value `"10"` of type `String`. MontiArcAutomaton detects this and raises the depicted error.

```

1 component IMotor[int max="10"] { ✖ // Incompatible type.
2   port
3     in int command;
4   // ...
5 }
```

MAA

Listing 4.22: The component type `IMotor` defines the configuration parameter `max` of type `int` with a default value `"10"` of type `String`.

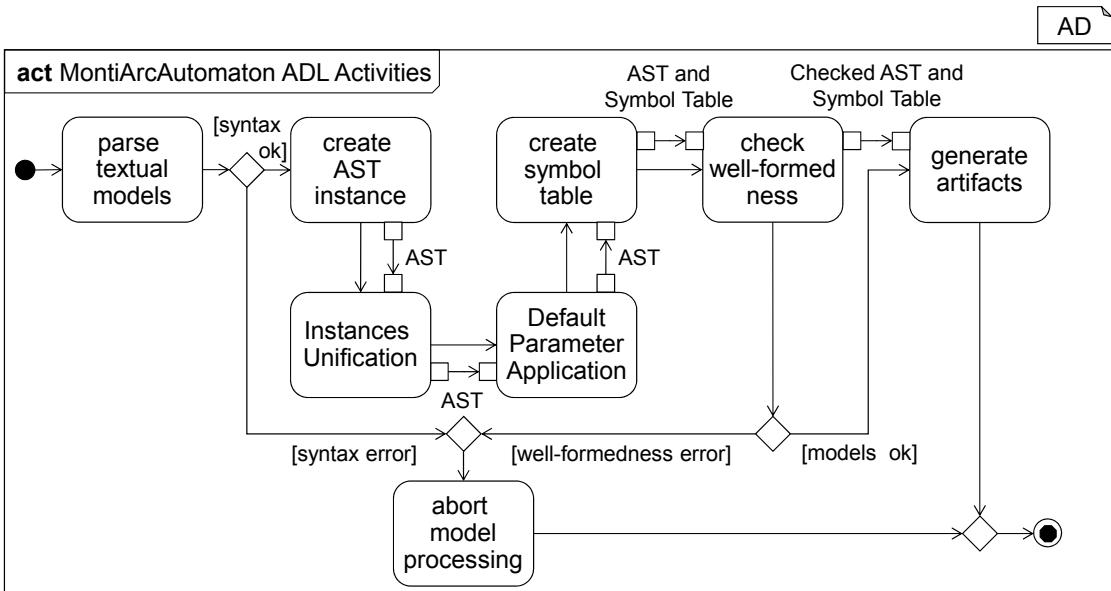


Figure 4.4: MontiArcAutomaton extends the typical MontiCore execution with the two new activities for both model transformations.

4.1.5 Transformations on the MontiArcAutomaton ADL AST

The AST resulting from parsing a MontiArcAutomaton ADL software architecture model is a representation of the processed model's content. As the AST is the central data structure for symbol table creation and code generation, two model transformations optimize it for subsequent usage (cf. Figure 4.4). The first translates subcomponent declarations with multiple subcomponents into multiple subcomponent declarations with a single subcomponent each. The latter applies the default parameter values to subcomponent declarations. The MontiArcAutomaton ADL applies all transformations prior to symbol table creation and context condition checking. Hence, all transformed software architectures must pass the context conditions prior to code generation and further analyses. This ensures that transformations yield a valid MontiArcAutomaton software architecture. To this effect, MontiArcAutomaton extends the default activities of MontiCore as depicted in Figure 2.3 with two new activities prior to symbol table creation. This also yields the benefit that applying the transformations to the AST suffices and their effects do not need to be recreated manually for the model's symbol table entries. Furthermore, performing transformations prior to code generation allows code generators to be unaware of any model transformations performed beforehand (cf. Req. *TRQ-7*).

Subcomponent Instances Unification

MontiArc distinguishes between subcomponent declarations and subcomponents. A subcomponent declaration consists of a component type, arguments for its configuration pa-

rameters, and a list of subcomponent names. This construction allows defining multiple subcomponents of the same component type and with the same component arguments conveniently. Listing 4.23 depicts a subcomponent declaration with two instances (l. 7). Here, the composed component DoubleAdder takes two numbers via incoming ports *a* and *b* and performs multiple additions using only subcomponents of type Adder (ll. 7-8) with the output being emitted via outgoing port *c*. The subcomponents of type Adder are instantiated with arguments for offsets.

```

1 component DoubleAdder {
2   port
3     in Integer a,
4     in Integer b,
5     out Integer c;
6
7   component Adder<Integer>(1) first, second;
8   component Adder<Integer>(2) third;
9
10  // first calculates a+b+1
11  connect a -> first.a;
12  connect b -> first.b;
13
14  // second calculates a+b+1
15  connect a -> second.a;
16  connect b -> second.b;
17
18  // third calculates ((a+b+1) + (a+b+1)) + 2
19  connect first.c -> third.a;
20  connect second.c -> third.b;
21
22  connect third.c -> c;
23 }
```

MAA

Listing 4.23: The composed component DoubleAdder declares three subcomponents in two subcomponent declarations (ll. 7-8).

However, subsequent processing has to consider all subcomponent instances of all subcomponent declarations. This has led to issues with the inherited MontiArc AST calculating the actual subcomponent instances lazily. Developers got confused by finding two unnamed subcomponents of type Adder in the AST but no subcomponent instances (due to lazy calculation). As the MontiArcAutomaton AST inherits from the MontiArc AST, this issue is propagated to MontiArcAutomaton as well. To facilitate development with MontiArcAutomaton ASTs, the subcomponent instances unification transformation iterates over the architecture's AST and replaces each subcomponent declaration of multiple instances with multiple subcomponent declarations of a single instance each. The configuration arguments of subcomponents are preserved and the result is a valid MontiArcAutomaton software architecture again.

Applying the transformation to the software architecture MultiAdder changes its AST in-place and results in the architecture depicted in Listing 4.24, which contains three subcomponent declarations (ll. 7-9) with a single subcomponent instead.

```

1 component MultiAdder {
2   port
3     in Integer a,
4     in Integer b,
5     out Integer c;
6
7   component Adder<Integer>(1) first;
8   component Adder<Integer>(1) second;
9   component Adder<Integer>(2) third;
10
11   // Connectors remain unchanged
12 }
```

MAA

Listing 4.24: The transformed MultiAdder component features three subcomponent declarations with a single subcomponent instance each.

Default Parameter Value Application

MontiArcAutomaton introduces default values to configuration parameters. Respecting these during subcomponent instantiation either requires aware code generators or appropriate model transformations prior to code generation. Since multiple code generators have been developed prior to the introduction of default parameters, retaining their compatibility was of the essence as required by Req. TRQ-7. Therefore, MontiArcAutomaton ADL models are transformed prior to code generation to apply default values to the configuration parameters of instantiated component types.

To this effect, this transformation traverses the AST of composed components (such as the component type DoubleClock depicted in Listing 4.3) and checks every subcomponent declaration for missing arguments. If such a subcomponent declaration is encountered, its type is loaded and the default values for omitted arguments are applied.

```

1 component DoubleClock {
2   component Clock(1,2) clock0;
3   component Clock(1,10) clock1;
4   // ..
5 }
```

MAA

Listing 4.25: The component type DoubleClock after applying the default parameter value to its second subcomponent declaration (l. 3).

After applying this transformation, the component type DoubleClock (Listing 4.3) is transformed in-place into the component type depicted in Listing 4.25. Here, the

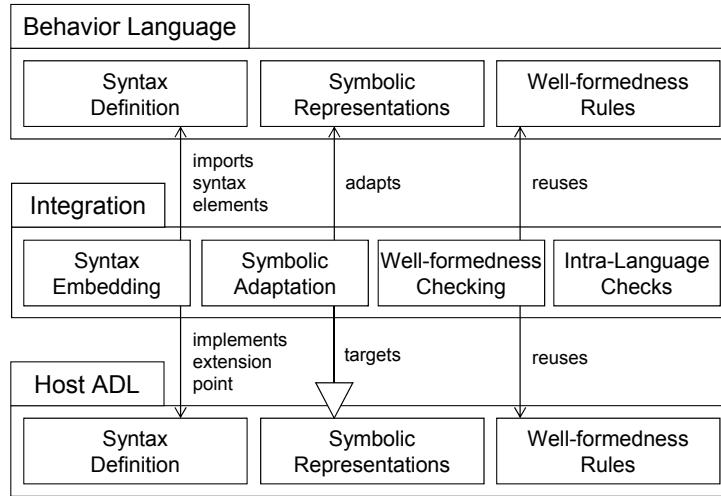


Figure 4.5: Language integration relies on syntax embedding, symbolic adaptation, and well-formedness rule reuse.

subcomponent declaration of instance `clock1` (l. 3) is completed with the default value 10 for its second configuration parameter.

4.2 Embedding Component Behavior Languages

Integrating component behavior languages into an ADL either requires to compose the languages *a priori*, or to provide composition mechanisms for independently engineered languages. *A priori* composed language aggregates are tightly coupled and hinder integration of new languages. Hence, they are hardly reusable in projects with different language requirements. *A posteriori* language integration either requires changes to the participating languages (invasive approaches) or employing languages designed for composition. The latter enables composition without language modifications (non-invasive approaches). For the integration of component behavior languages into the MontiArc-Automaton ADL, we present a concept for non-invasive, *a posteriori* language integration. This concept rests on the separate integration of syntax and static semantics and relies on the assumptions that (a) the behavior languages to be integrated describe input-output behavior, and (b) they can be integrated into a single, well-defined extension point of the ADL. Both assumptions rest on the nature of C&C architectures that encapsulate behavior in components. Integration of DSLs into other C&C constituents (such as ports or connectors) requires definition of an according extension point and clarification of the integration semantics. We currently do not consider this.

Our language integration concept employs syntax embedding, symbolic adaptation, and well-formedness checking as depicted in Figure 4.5. The integration combines the syntaxes of the component behavior language and of the host ADL at the single, well-defined extension point. This already enables creating and parse models of the resulting

language aggregate. However, to check the well-formedness of integrated models, their joint interpretation requires adapting the languages' symbols accordingly (e.g., adapt names of an embedded language's inputs to incoming ports of the MontiArcAutomaton ADL). With the symbolic adaptation in place, checking the well-formedness of integrated models reuses the well-formedness rules of both languages and may include new inter-language well-formedness rules. Thus, language engineers can easily develop and integrate modeling languages as required by the participating domain experts that enact the roles of application modelers.

We present a realization of this concept for MontiArcAutomaton that builds on the language integration mechanisms of MontiCore to enable pervasive model-driven engineering (Req. *MRQ-3*). This realization employs language embedding to integrate the syntax of behavior languages into the MontiArcAutomaton ADL and language aggregation to enable interpretation of embedded models. For the former, it exploits parser integration, for the latter it relies on language families. Both mechanisms are presented in Section 2.2.2.

Section 4.2.1 presents embedding of component behavior language syntax. Based on this, Section 4.2.2 explains symbolic integration, which includes adaptation and integrated well-formedness checking. Finally, Section 4.2.3 describes the infrastructure required for language integration and a modeling language to configure integration.

4.2.1 Syntactic Behavior Language Embedding

Language embedding is a syntactic integration mechanism, where the host language's grammar provides an extension point in form of a designated production. A production of the language to be embedded is registered for this extension point and whenever an integrated model is parsed, the infrastructure for processing the embedded language's production is invoked. To facilitate reuse, we do not integrate the behavior languages' and host language's grammars, but the parsers MontiCore generates (cf. Section 2.2). This facilitates exchange of embedded languages and does not impose modification of the languages' grammars.

The MontiArcAutomaton ADL provides an extension point for behavior languages inside components. Productions of behavior languages are mapped to this extension point. The parser MontiCore generates for the MontiArcAutomaton ADL then delegates parsing of the behavior languages' non-terminal productions to their responsible parsers. The mapping required to enable such embedding can easily be specified as both languages are available to the integrator.

Conceptually, our approach to behavior language integration amounts to describe (a) which element of the host language serves as extension point and (b) which element of the behavior languages should be mapped to this extension point. In MontiCore, such extension points are defined in terms of external productions. The productions of the behavior languages require no designation and each production of a behavior language may be embedded. While convenient, this may lead to hardly useful combinations (for instance, embedding only the production representing states of a finite state machine into components will hardly produce input-output behavior). Checking this is subject

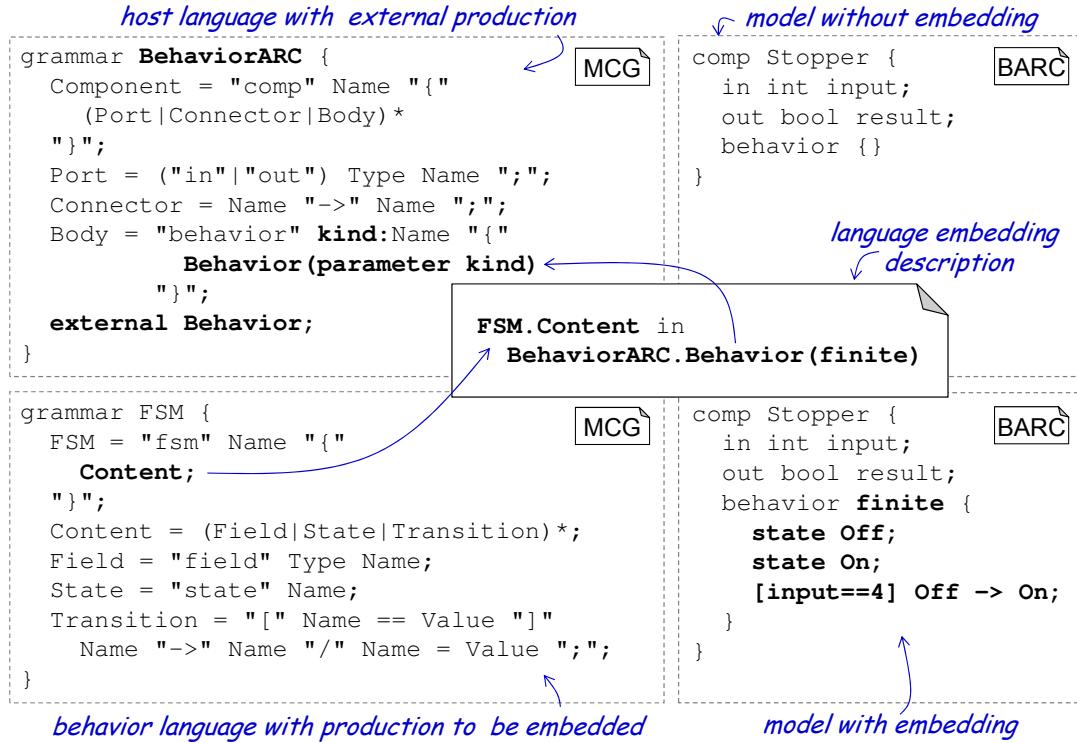


Figure 4.6: Language embedding requires to specify which productions of the behavior language are embedded under which conditions into the extension point of the host language.

to future work in the context of language interfaces [CVdBCR15] and their application to language composition [CCF⁺15b]. Figure 4.6 depicts the syntactic language embedding and resulting models with a variant of the ARC grammar (top left) as introduced in Listing 2.1. This variant, **BehaviorARC** uses a different Body production to enable embedding behavior languages into its Behavior production. The latter is a parametrized production that expects the single parameter kind, which is used to distinguish embedded languages. The top right of Figure 4.6 shows a **BehaviorARC** component model with two ports, but without embedded behavior.

At its bottom left, Figure 4.6, presents an excerpt of a grammar for finite state machines. The **FSM** grammar comprises the container production **FSM** and its **Content**. The latter contains arbitrary many fields, states, and transitions, where fields are either input data sources or output data sinks, states are names, and transitions employ guards of single equality checks over fields to describe the possibilities to switch between states. If the guard holds, a transition can fire and emit messages via a single outgoing port. As depicted at the center of Figure 4.6, the production **Content** of **FSM** is embedded into **Behavior** of **BehaviorARC**. The argument for this embedding is **finite** which is passed to be the kind of **Body** and helps MontiCore to distinguish which parsers to

select. It also becomes a pseudo-keyword in the concrete syntax of integrated models: Whenever MontiCore parses the name after `behavior`, it looks up which parsers are registered for this name and delegates parsing to the responsible parser. Hence, if the value parsed for `kind` does not match the keyword a parser is registered to, parsing fails. Consequently, the integrated model depicted at the bottom right of Figure 4.6 uses this pseudo-keyword and arbitrary `FSM.Content` elements for its `Behavior`.

With pure MontiCore, language embedding must be specified in language configuration files as depicted in Figure 2.5. These files specify, among embedding information, various other language-specific information to configure MontiCore's model processing infrastructure. This includes a reference to the generated data type to represent root elements of this language (which represent model files), configuration of a factory for root elements of this language (which includes specification of embedding), and declaration of a workflow to parse models of this language. Listing 4.26 shows an excerpt of the language configuration of a previous version of the MontiArcAutomaton ADL, which specifies a root class (l. 2), a root factory (ll. 4-9), and a parsing workflow (ll. 11-12). The root factory specification contains a declaration of a designated start production (l. 7). As some productions, for instance to process package declarations and `import` statements, are provided by MontiCore, the language engineer must specify where her own language begins. The start production declaration defines this with the name of `maa`. Also the root factory describes embedding of the `FSMContent` production into the `BehaviorModel` production of the MontiArcAutomaton ADL under condition of the keyword `finite`. Here, `maa` identifies the parser where the parser of `FSMContent` should be embedded. The token `ef` is the name for the embedded parser of `FSMContent` and allows embedding productions into this parser as well.

```

1 language MontiArcAutomatonADL {
2   root MontiArcAutomatonRoot<MCCompilationUnit>;
3
4   rootfactory MontiArcAutomatonRootFactory
5     for MontiArcAutomatonRoot<MCCompilationUnit> {
6
7       MCCompilationUnit maa <<start>>;
8       FSMContent ef in maa.BehaviorModel(finite);
9     }
10
11   parsingworkflow MontiArcAutomatonParsingWorkflow
12     for MontiArcAutomatonRoot<MCCompilationUnit>;
13 }
```

LNG

Listing 4.26: A MontiCore language configuration file for the integration of production `FSMContent` of the AUTOMATA language (Chapter 5) into components.

Language configurations require in-depth knowledge about MontiCore and its language processing concepts, such as the root elements and which parsing workflow to select.

With MontiArcAutomaton, all of these are fixed by design, hence only the embedded productions must be specified. For these, also the extension point is fixed. Hence, language embedding amounts to specifying productions of the behavior language with the discriminating keyword (e.g., `finite`) only. MontiArcAutomaton therefore provides means to liberate language engineers from MontiArcAutomaton ADL internals such as selecting correct root and parsing workflows. Section 4.2.3 presents these.

```

1 component Filter[int max] {
2   port
3     in int value,
4     in Boolean doFilter,
5     out int result;
6
7   behavior finite {
8     state permeable;
9     state impermeable;
10
11    [doFilter == false] permeable -> permeable / result = value;
12    [doFilter == true] permeable -> impermeable / result = max;
13    [doFilter == true] impermeable -> impermeable / result = max;
14    [doFilter == false] impermeable -> permeable / result = value;
15  }
16 }
```

MAA

Listing 4.27: The component `Filter` contains an embedded FSM model comprising two states and four transitions to describe its behavior.

For example, to embed models of the FSM language (Chapter 5) into MontiArcAutomaton ADL components, the language engineer integrating FSM must specify which of its productions should be used. In case, she selects to embed the `FSMContent` production into the `BehaviorModel` extension point of the MontiArcAutomaton ADL grammar, models such as `Filter` depicted in Listing 4.27 are possible.

This component model contains an embedded state machine (ll. 7-15) to filter integer numbers. If filtering is active (the component's input port `doFilter` receives `true`), values are capped to the components configuration parameter `max`. To this effect, the state machine defines two states (l. 8-9) and four transitions (ll. 11-14). These transitions read the value of `doFilter` and send corresponding values via the port `result`. Everything between the opening curly bracket after `finite` (l. 7) to the next closing curly bracket (l. 15) is embedded from the FSM grammar.

The embedded state machine expects to read data from an input of name `doFilter` and expects to write data to an output of name `result`. In stand-alone FSM models, these names are symbols referring to fields of the state machine itself. In the context of a component, these symbols should refer to ports of the component, which the FSM language is independent of. For proper integration, including inter-language well-formedness checking, this change of symbol meaning must be explicated, such that

MontiArcAutomaton can ensure that `doFilter` and `result` are of data types compatible to the valuations and assignments on the transitions. Thus, syntactic language embedding alone does not suffice, but requires *symbolic* integration as well.

4.2.2 Symbolic Language Integration

The names used in a model to reference parts of the same or other models are symbolic references with certain meaning. For instance, the left-hand side of the assignment `doFilter = false` of the first transition depicted in Listing 4.27 is only meaningful in an integrated state machine if `doFilter` references a component port or variable. Other interpretations, such as assigning values to a state or to another value are not intended. However, the FSM language is independent of the MontiArcAutomaton ADL and, hence, unaware of the concepts of component ports and variables. In stand-alone FSM models, `doFilter` references an output of the automaton. This interpretation must change for integrated FSM models.

Generally, when integrating a behavior modeling language into MontiArcAutomaton ADL components, the meaning of these symbolic references may change. MontiArcAutomaton and its ADL rely on MontiCore's language aggregation mechanisms (cf. Section 2.2.2) to reflect these changes in meaning. These mechanisms require that each participating language explicates their *symbols* for relevant models parts. MontiArcAutomaton utilizes adaptation between the symbols describing input and output models elements of behavior languages and its port and variable symbols. This enables their correct interpretation and to reuse the language's well-formedness checks.

Non-invasive symbolic integration requires no changes to participating languages and only little additional infrastructure. The additional infrastructure consists of adapters between the languages' relevant symbols and new well-formedness rules that arise from integration. Furthermore, the integration requires specifying, how to create and manage symbols of the behavior languages as well as to select which well-formedness rules of the behavior languages to apply to embedded models. For both, existing infrastructure in the respective behavior languages can be reused. Thus, symbolic integration of behavior languages into the MontiArcAutomaton ADL requires:

- **Adaptation:** Integration of behavior languages into the MontiArcAutomaton ADL requires to identify their symbols relevant to input and output. The ports and variables of the MontiArcAutomaton ADL must be adapted to these to allow their interpretation in the context of embedded behavior models.
- **Restriction:** Behavior language integration may change the requirements on well-formedness of models. To reflect this, new well-formedness rules might need to be added as well as existing ones removed.
- **Infrastructure Integration:** The MontiArcAutomaton ADL and behavior languages provide means to create, manage, and resolve symbols. These must be integrated into a language family, such that the joint interpretation of their symbols is possible. This includes the related infrastructure as well.

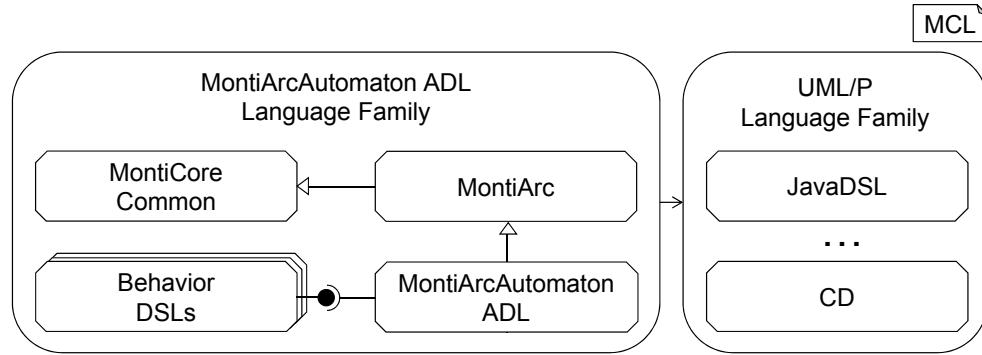


Figure 4.7: The MontiArcAutomaton ADL language family comprises the MontiArcAutomaton ADL, MontiArc, languages of the UML/P, and provides extension points to integrated component behavior languages.

As MontiCore prescribes only minimal language infrastructure (for instance, symbol tables and well-formedness rules are optional), the integration might require to provide such infrastructure for a behavior language.

In general, combined interpretation of models of different languages (whether embedded or in separate artifacts) requires aggregation of adapters, well-formedness rules, and infrastructure into a joint language family. The language family of the MontiArcAutomaton ADL (Figure 4.7) aggregates the MontiArc [HRR12] language, from which it inherits not only syntactically, but also symbolically (i.e., it inherits and extends MontiArc’s syntax as well as its symbols). Via MontiArc, it transitively extends also from the MontiCore Common language that provides common productions, such as types and literals. It also aggregates the UML/P language family [Rum11] to use Java/P and class diagrams for data type modeling. Furthermore, it provides extension points to aggregate the different infrastructure necessary to integrate component behavior languages.

The MontiCore infrastructure required for symbolic language integration comprises:

- Symbol table modules: In MontiCore, creation and management of symbols is performed by its symbol table infrastructure, which takes care of producing symbol table entries from corresponding AST nodes, persisting, qualifying, and resolving these. Integration reuses the symbol table infrastructure of behavior languages without modifications.
- Adapters: To reflect changes in the meaning of references, MontiCore relies on adaptation between symbol table entries of related concepts. These adapters are specific to integration and must be developed as explained in [HLMSN⁺15].
- Context conditions: Integrating models of behavior languages into components requires to apply the behavior languages’ well-formedness rules as well. Additionally, integration might entail new well-formedness rules. With MontiCore, well-formedness rules are implemented as context conditions and integration must

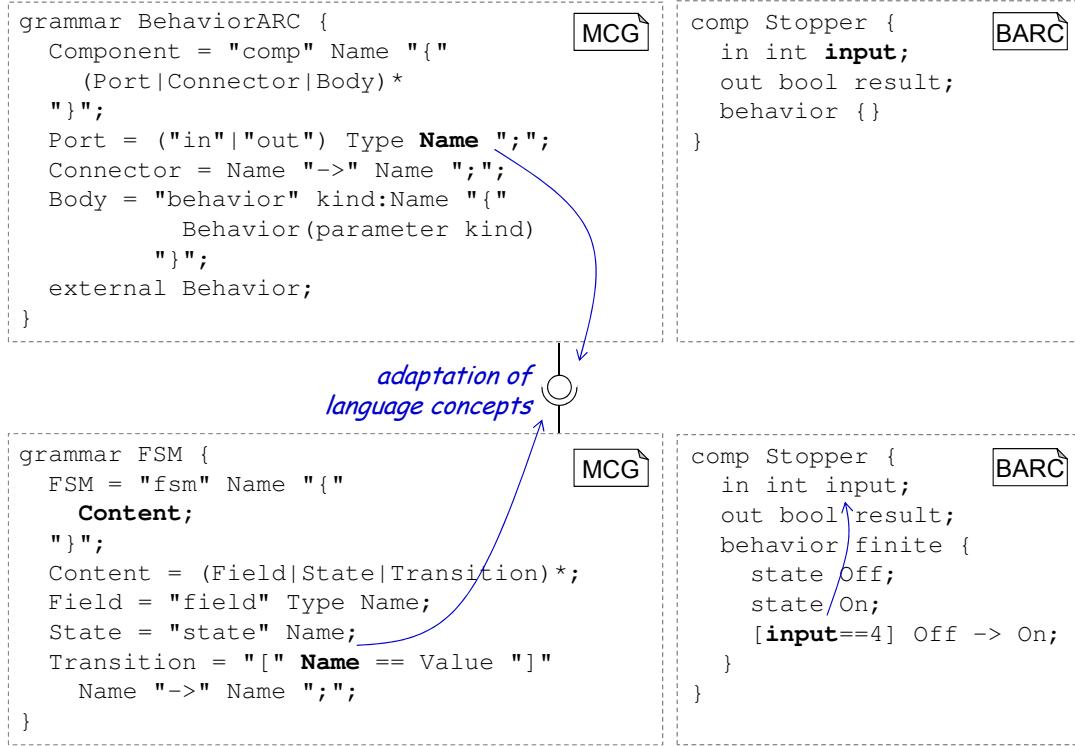


Figure 4.8: Adaptation between names of different languages enables interpreting references in embedded behavior language properly.

provide the context conditions to apply. The intra-language context conditions can be reused with minimal effort, the new context conditions must be implemented as presented in [Vö11].

- Workflows: MontiCore languages may employ different workflows to optimize their ASTs or symbols. If required, integration must consider a behavior language's workflows as well.

Non-invasive context condition reuse commands the existence of adapters: consider a well-formedness rule of `FSM` that checks whether the value assigned to a field matches the field's type (such as the field `int a`, depicted at bottom right in Figure 4.6). If the embedding of `FSM` models into MontiArcAutomaton ADL components prohibits fields in favor of ports, checking the well-formedness of assignments tries to resolve the symbol for the name `a` and will fail to produce a field symbol table entry. Instead, the MontiArcAutomaton ADL must provide the corresponding port symbol. Adapters bridge that gap by interpreting the names in transition guards as references to ports as depicted in Figure 4.8. The adapters ensure that certain names expected by the CFG of `FSM` are interpreted as references to `BehaviorARC` elements defining these names.

Figure 4.9 shows the realization of the adaptation of `BehaviorARC` ports to `FSM`

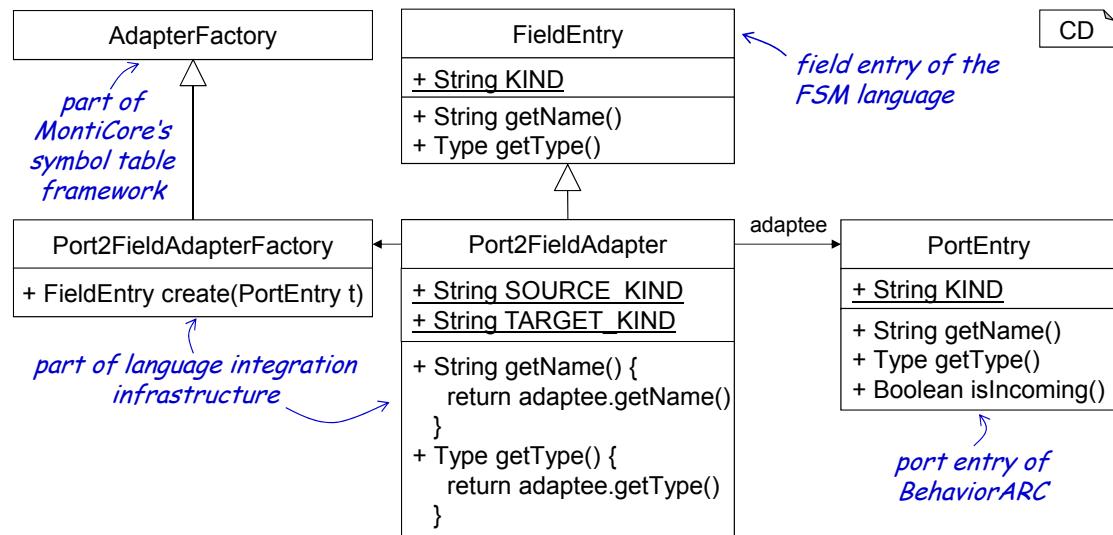


Figure 4.9: An adapter to use port entries of BehaviorARC as field entries for symbolic integration of FSM models.

fields via their symbol table entries. Whenever a name expected to reference a field is looked up by BehaviorARC, it returns a port symbol table entry disguised as a field entry instead. To this effect, the language engineer integrating FSM into BehaviorARC must develop and register the available adapters. Each adapter references the source symbol kind (cf. Section 2.2.1) of the entry type it adapts, the target symbol kind of the entry it disguises as, and provides a factory to create adapters of its type. Whenever a resolver looks up the name and kind of the field entry corresponding to the expected field name input, MontiCore will return an instance of `Port2FieldAdapter` instead. The adapter acts as a field entry and can be used, for instance, to check whether the `Type` associated with `input` matches the type of the value it is compared to.

Aside from reflecting changes in meaning, language engineers may exploit adaptation to make other concepts available to participating languages. With the MontiArc-Automaton ADL, components may yield configuration arguments that are passed at component instantiation and are constant from then on. Although FSM does not have such a concept, adapting configuration arguments to non-writable fields (which requires another inter-language well-formedness rule prohibiting assignments to fields that actually are component arguments) might be useful to allow usage of configuration arguments for behavior calculation.

The next section presents the infrastructure to integrate symbol tables, context conditions, adapters, and workflows into the MontiArcAutomaton ADL language family.

4.2.3 Language Integration Infrastructure

Integrating a behavior language into the MontiArcAutomaton ADL requires declaring which of its productions should be embedded, specifying a keyword for this production,

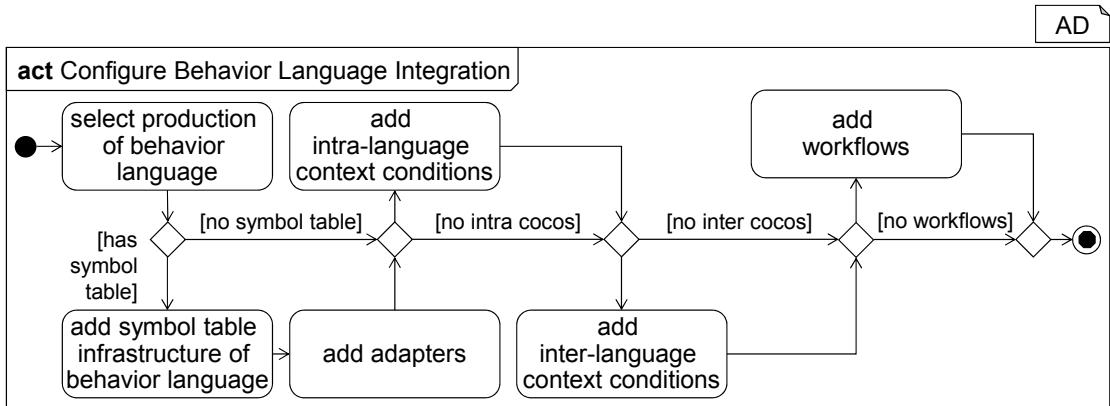


Figure 4.10: Integrating behavior languages into MontiArcAutomaton requires consideration of symbol table infrastructure, adapters, and context conditions.

and integrating of symbol table infrastructure, context conditions, and workflows. The symbol table infrastructure comprises entries, entry creators, qualifiers, resolvers, and deserializers. Specification of context conditions enables including intra-language conditions of the behavior language (as not all its well-formedness rules might be applicable in the context of a component) and adding new intra- and inter-language well-formedness rules. Not all of these modules are necessary. Figure 4.10 therefore describes the language engineer’s efforts for integration of a single behavior language. It begins with selecting the production of the behavior language to embed into components including its embedding keyword. This production defines which behavior language models parts can be processed inside components and governs which part of the language’s generated MontiCore parser to integrated (cf. Section 4.2.1). If the language to be embedded has symbol table infrastructure, it needs to be integrated and adapters between relevant symbols must be provided as explained in Section 4.2.2. Afterwards, if available, intra-language context conditions and inter-language context conditions, as well as workflows are integrated. To integrate multiple languages, this must be repeated for each language.

As MontiCore relies on DSLTools to process models, configuration with behavior languages must be provided in form of a language combination specific DSLTool. MontiArcAutomaton supports two ways to define such tools: either via handcrafting subclasses of certain MontiCore classes and implementing their methods correctly, or using a domain-specific embedded language (DSEL) [Fow10] that parametrizes a generic integration tool. While the DSEL’s expressiveness is restricted to support the most common use cases of behavior language integration, it enables configuring the MontiArcAutomaton ADL with new component behavior languages with little effort. If the behavior language to embed has special integration requirements, handcrafting the integration enables harnessing the full language integration power of MontiCore. Either way, language integration requires no changes to participating languages as required by Req. *MRQ-6*. The following sections explain how the MontiArcAutomaton ADL can be extended with behavior languages via handcrafting integration artifacts or using the DSEL.

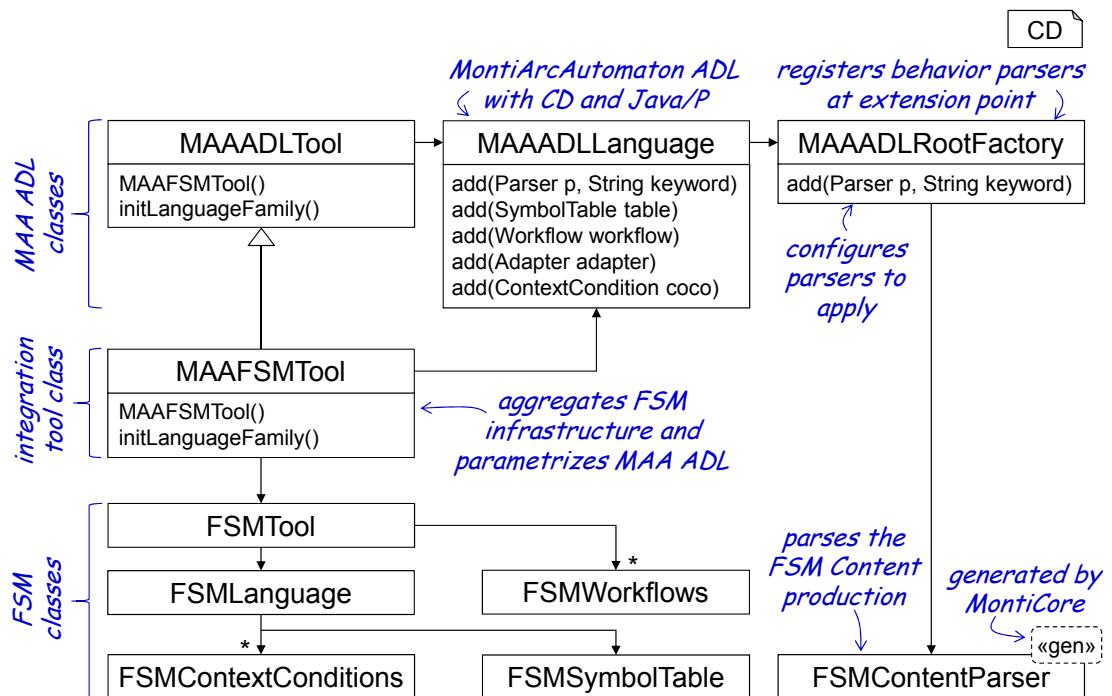


Figure 4.11: Integrating FSM behavior into the MontiArcAutomaton ADL requires provision of a single new class inheriting from `MAAADLTool` only.

Handcrafting the Integration Infrastructure

With MontiCore languages, language family configuration is part of their DSLTools. Each language's DSLTool contains an instance of MontiCore's `LanguageFamily` class, which contains symbol table entry creators, resolvers, qualifiers, deserializers as required by the individual language. It also contains the context conditions and workflows to apply. The `MontiArcAutomaton` ADL is defined in terms of the `MAAADLanguage` class used by the `MAAADLTool`. The language class `MAAADLanguage` extends from `LanguageFamily` and configures the `MontiArcAutomaton` ADL to interact with MontiArc, UML/P, and Java/P. Furthermore, it provides methods to add language infrastructure of behavior languages. It also controls which parsers to use via configuring an instance of `MAAADLRootFactory` (cf. Section 4.2.1).

Figure 4.11 illustrates handcrafted integration with the FSM language of Figure 4.6. The top three classes starting with MAAADL are part of the MontiArcAutomaton ADL and can be reused for integration of arbitrary MontiCore behavior languages. The class MAAFSMTool implements the integration of FSM infrastructure into the Monti-ArcAutomaton ADL. To this effect, it overrides the method `initLanguageFamily()` of MAAADLTool and adds relevant information retrieved from the FSMTTool to the MAAADLLanguage. The remaining bottom five classes starting with FSM are part of the FSM language and used by the accessed via the FSMTTool.

The stand-alone DSLTool `FSMTool` holds an instance of the `FSMLanguage` class, another descendant of `LanguageFamily`, and the workflows (such as model transformations) that are applied to FSM models. The `FSMLanguage` class contains symbol table infrastructure (including symbol table entries, entry creators, qualifiers, resolvers, and deserializers) and context conditions of the FSM language. All of these can be reused via integration into the `MAAADLLanguage` language. To this effect, the `MAAFSMTTool` extends the `MAAADLTool`, which contains the parametrizable `MAAADLLanguage`, and references the `FSMTool`. From the latter, it retrieves FSM workflows, context conditions, and symbol table to the MontiArcAutomaton ADL. It also contributes inter-language context conditions, adapters, and overrides the root factory to register the parser for the `FSM.Content` production the extension point of the MontiArcAutomaton ADL. Hence, when using the `MAAFSMTTool` to parse integrated models, the MontiArcAutomaton ADL parser invokes the `FSMContentParser` for each component behavior of the FSM language, combines their ASTs, creates the corresponding symbol table entries, checks its well-formedness, and performs workflows using the provided infrastructure.

Handcrafting integration infrastructure requires development of a single class with a few lines of code only. However, MontiCore gives languages engineers much freedom regarding the implementation of languages (for instance it does not prescribe how to create and manage symbol tables). Thus, integration requires in-depth expertise about the behavior language to be integrated and specifying universal implementations of the relevant methods is impossible.

A Groovy DSEL for Behavior Language Integration

To facilitate integration of behavior languages, MontiArcAutomaton employs a domain-specific embedded language based on the GPL Groovy [KKL⁺15]. Groovy is a programming language for the Java virtual machine, which allows omitting syntactic sugar (such as brackets around method arguments and dots for method concatenation). Thus, it lends itself to develop DSELS [Fow10, Gho10] with similar feature than Java but a less verbose syntax. Models of the Groovy behavior configuration language (GBC) specify which modules of a behavior language are contributed to integration. MontiArcAutomaton processes these and configures the modeling language infrastructure of MontiCore with collected relevant integration properties. As Groovy is compatible to Java and MontiCore language modules are implemented in Java, the GBC can interface the MontiCore modules with ease. This, for instance, allows interacting with the `FSMTool` class depicted in Figure 4.11 to retrieve its information instead of specifying each required module manually. It ultimately liberates developers performing language integration from requiring expertise about behavior language implementation details.

Listing 4.28 illustrates these benefits with a GBC model integrating the FSM language into the MontiArcAutomaton ADL. First, it defines a unique name (l. 1), which is used to identify components of the language in the resulting aggregate and as the pseudo-keyword under which the parser of the selected production will be embedded. Afterwards, it references the behavior language production to be embedded (l. 2). The latter is identified by a qualified name consisting of its dot-separated path, grammar

```
1 name "finite"
2 behavior "FSM.Content"
3 tool new fsm.FSMTool()
4 coco new maafsm.NoFieldsInEmbeddedModels()
5 adapter new maafsm.Port2FieldAdapter()
```

GBC

Listing 4.28: Groovy behavior configuration model for embedding the FSM language using instances of MontiCore language implementation classes.

name, and production name. It references the language's DSLTool class `FSMTool` (l. 3), from which it retrieves the `FSMLanguage` and workflows. From the language, it retrieves entries, entry creators, qualifiers, resolvers, and deserializers of the `FSM` symbol table. As integrated `FSM` models should act on ports and variables of the surrounding component, (a) using `Field` elements should be prohibited and (b) the interpretation of references on transitions must be changed to ports and variables. The former requires a new context condition and the latter an adapter as presented in Section 4.2.2. Hence, the GBC model references a context condition class (l. 4) as well as an adapter class (l. 5). The syntax of this language is defined by the fluent interface [Fow10] of the class `GBCBuilder` depicted in Figure 4.12. This class employs a variant of the builder pattern [GHJV95] to create instances of `BehaviorConfiguration` that describe integrated languages. The `GBCTool` extends the `MAAADLTool`, interprets its `BehaviorConfiguration` instances, and configures the `MontiArcAutomaton` ADL accordingly. With Groovy's feature to omit syntactic sugar, instances of `BehaviorConfiguration` can be defined as depicted in Listing 4.28. Please also note that the order of method invocations, and hence the order of model keywords, is arbitrary.

Usage of GBC models with the `GBCBuilder` imposes the requirements on language implementation with MontiCore: For one, the behavior language must provide a class inheriting from MontiCore's `DSLTool` class. As MontiCore processes models with instances of `DSLTools`, this is de facto standard and a requirement that is fulfilled almost naturally by using MontiCore [Vö11]. However, technically, it is possible to use MontiCore without a `DSLTool` and languages providing no tool are not supported for Groovy configuration. The `GBCBuilder` also expects all context conditions to inherit from MontiCore's `ContextCondition` class. Although this is standard for MontiCore languages as well [Vö11], it technically is possible to check well-formedness rules differently. If the well-formedness rules do not inherit from `ContextCondition`, `GBCBuilder` cannot support these. Finally, the object passed to method `adapter()` must implement MontiArcAutomaton's `IEntryAdapter` interface. Implementing this interface entails that each adapter provides an instance of MontiCore's `AdapterConfiguration`, which contains a qualifier, a resolver, and a deserializer for each adapter. The interface `IEntryAdapter` is not part of MontiCore. However, the required adapters are specific to the integrated languages and hence, must be created for each language combination anyway. Thus, enforcing implementing of `IEntryAdapter` is no severe requirement.

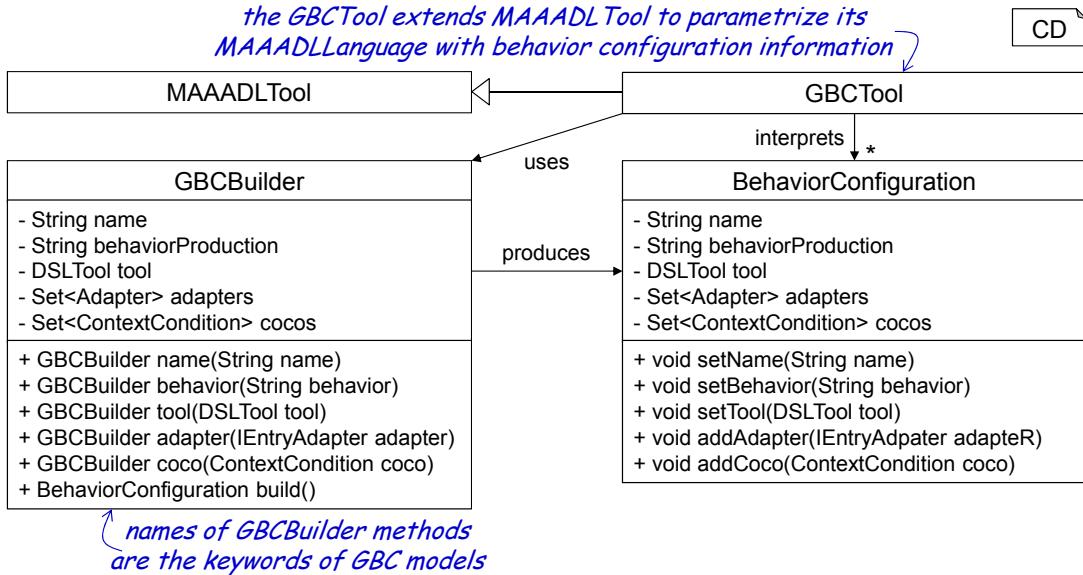


Figure 4.12: Quintessential classes of MontiArcAutomaton's Groovy behavior configuration infrastructure.

Using the GBC language to integrate behavior languages amounts to providing the required GBC models (one per language) as well as the required types and instances of the language implementation infrastructure explained above. With this, the overall project layout for integration of FSM into MontiArcAutomaton could be as depicted in Figure 4.13. Elements of the FSM project and of the MontiArcAutomaton project are reused without modification. For convenience, the integration artifacts are placed into a third project. It also is possible to place these into the application project, which might be useful in contexts where the specific language combination is not intended to be reused with different applications. Invoking the GBCTool with the configuration enables processing integrated models, parses the model into a combined AST, and produces additional infrastructure not depicted here (such as the symbol table entries of component and behavior elements).

4.2.4 Language Integration Semantics

Models of integrated languages must be able to describe input-output behavior reading from discrete data sources (incoming ports and variables) and writing to discrete data sinks (outgoing ports and variables). They perform their calculations (typically in read-compute-write cycles) within a single FOCUS time slice of the surrounding architecture. Hence, they partition time slices into sequences of operations. We consider these untimed in the FOCUS sense, but causally related. The semantics of integrated behavior thus follows the concept of superdense time [MP93], which, following [Lee10], distinguishes between a discrete “time continuum” (the global FOCUS time) and “untimed causally-

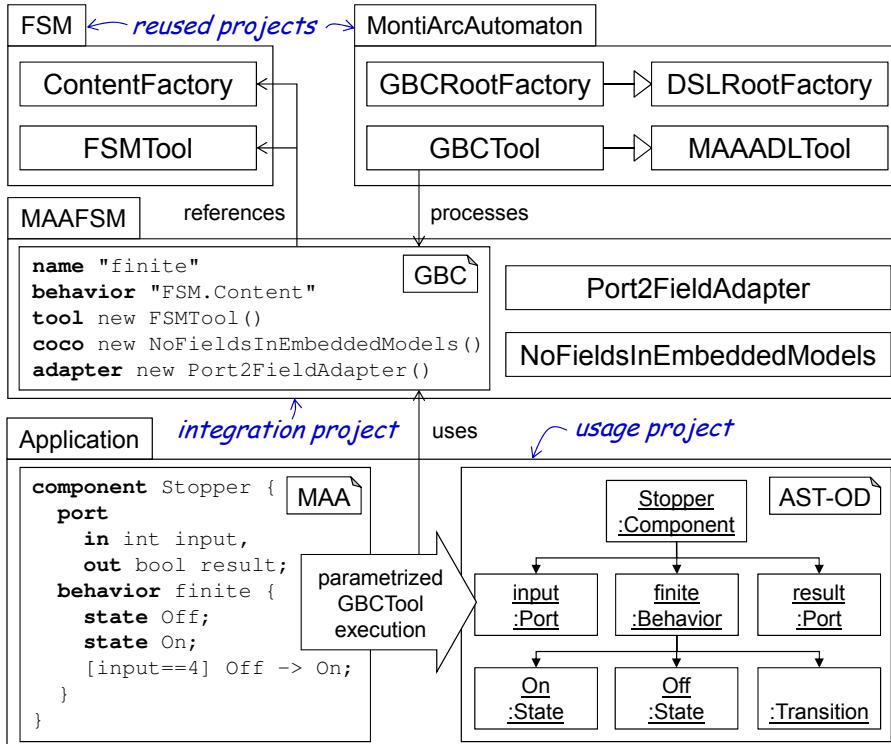


Figure 4.13: Exemplarily behavior language configuration and artifacts in the context of their containing projects.

related actions” (a behavior model’s actions within the time slice of a component).

We also assume that the operations performed in the superdense time of a component are computed fast enough to not interrupt the architecture’s functionality. Also, in every time slice, the MontiArcAutomaton ADL provides values on each incoming port of a component. Hence, behavior languages may evaluate complex expressions over the complete interface of a component, instead of reacting on single messages only.

4.3 Discussion

MontiArcAutomaton models describe structure and behavior of logically distributed component & connector software architectures. The MontiArcAutomaton ADL itself is compact and easy to comprehend (cf. Chapter 9). To achieve this, it focuses on the core elements of C&C software architectures, which are components, connectors, and configurations (subcomponent topology) [MT00]. While this eases the learning curve [MDT07], the MontiArcAutomaton ADL thus lacks direct support of interesting features of other ADLs, such as

- ways to change the subcomponent topology at run-time [VVKM00, ACN02, SOK05, JBCG05, RBH⁺07, FG12],

- supporting multiple primitive behaviors per component [FG12] and switching between these at component instantiation or during run-time,
- a distinction between different types of ports [SSL11, FG12, BKH⁺13],
- properties of ports (aside from their types) [MRT99, ACN02, JBCG05],
- more complex connectors [GMW00, KGO⁺01, MCWF02, JBCG05, BCL⁺06, FG07, UNT10], or
- dynamic subcomponent replication [MDEK95, NPR13].

Integrating any of these comes at the price of introducing additional notational noise [Wil01] and accidental complexities [FR07]. As MontiArcAutomaton and its ADL rely on the language workbench MontiCore [KRV08b, KRV10], adding new features by means of language integration (cf. Section 2.2.2) is possible in a structured way and allows reusing great parts of the existing model processing infrastructure [Vö11, LNPR⁺13, HLMSN⁺15]. Hence, whenever tailoring the MontiArcAutomaton ADL to certain requirements is necessary, the implementation can rely upon the powerful language definition and integration features of MontiCore.

While MontiArcAutomaton supports integration of arbitrary modeling language for component behavior, embedding languages without dynamic semantics (such as class diagrams) will not produce input-output behavior without further interpretation. Future research on the globalization of modeling languages [CDB⁺14, CCF⁺15a] might help to make the requirements for *meaningful* behavior language embedding explicit and evaluable. In the same vein, embedding arbitrary productions of behavior languages will rarely produce meaningful models. Making language interfaces explicit and marking “behavior productions” as such could support automated integration efforts.

MontiArcAutomaton currently integrates syntax and static semantics only. Explicating dynamic semantics of MontiCore languages is subject to research and hence integration of dynamic semantics is not considered in this thesis.

Behavior language integration in MontiArcAutomaton is easier than general language composition as it only requires embedding into a single, well-defined extension point and joint interpretation of a few symbols (mostly ports and variables) to reuse most of the embedded languages infrastructure. The single extension point of MontiArcAutomaton must be implemented by a single production per behavior language.

4.4 Related Modeling Languages

Multiple architecture modeling languages and frameworks for C&C systems have been brought forth [MT00, MLM⁺13, RMT14a]. These languages and frameworks have emerged from different domains, such as automotive [BHS99, Hö07], avionics [FG12], or robotics [SSL11, DKS⁺12, BKH⁺13] and focus different aspects and challenges of architecture engineering from academic and industrial perspectives. Most of these are “first-generation ADLs” [MDT07] that are “solely focused” on technology instead of

business-related or domain-specific aspects. The flexible integration of domain-specific modeling languages with ADLs is rare and usually overly complicated. Where an architecture modeling infrastructure is driven by the demands of specific domain, extensibility usually is focused to a lesser extent and domain-specific issues are challenged instead.

Although extensibility is considered “a key property of modeling notations” [MDT07] most C&C software ADLs support a fixed set of language elements and component behavior languages. A recent survey on industrial use of ADLs observed that tailoring an ADL “towards whole company needs” [MLM⁺13] is partly feasible with “pre-existing extension mechanisms” yet these mechanisms are insufficient, the corresponding tools are “too generic”, and the ADLs are of “insufficient expressiveness”. This limits the language’s expressible principal design decisions and ultimately requires to describe important design decisions in GPL artifacts bundled with the architecture or renders the language makes it unusable.

The AADL [FG12] modeling language for hardware and software components of embedded systems features language elements to model hardware components, software components, and related properties, where MontiArcAutomaton focuses on modeling logical software architectures. AADL distinguishes component types from their implementations and the former defines the type’s interface. Component implementations inherit the implemented type’s interface and describe the type’s internals, such as sub-components or calls to subprograms. Components implementations of type thread may define unconditional sequences of subprogram calls. Subprograms comprise component-like interfaces and represent a “callable unit of sequentially executable code” [FG12]. AADL can also be extended with behavior languages through sub-languages conforming to the behavior annex [BFBFR07], which lacks integrated semantics with the surrounding architecture [YHMP09] and is overly complicated.

AutoFOCUS [HF11] is a C&C ADL and modeling infrastructure for the engineering of embedded distributed systems. It is based on the semantics of FOCUS [BS01, RR11] as well. It supports time-synchronous messaging with weakly causal and strongly causal component behavior. Behavior is modeled as state transition diagrams. However, AutoFOCUS does not distinguish between component types and their instantiations. This hampers component reuse.

The xADL [DVdHT01, DVdHT02] focuses on architecture extensibility as well and is grounded on a metamodel for XML-based ADLs. It shares many features with the MontiArcAutomaton ADL (such as atomic and composed components, instantiation, component behavior models), and provides language elements for product lines and variability the MontiArcAutomaton ADL does not support. Extension in xADL focuses on language integration on the metamodel level and does neither support non-invasive language integration, nor integration of model processing infrastructure [NDZR04]. Also, to the best of our knowledge, xADL does not support code generator composition. Instead, its “architecture instantiation schemas” [DVdHT01] tie software architecture models to specific implementations.

MathWorks Simulink [Tya12] features a block diagram language for the description of software as components and connectors. Stateflow [Sta] extends blocks with state

transition diagrams. The Stateflow semantics are not completely defined and have been formalized in various ways [HR04a, MC12]. In contrast to MontiArcAutomaton, StateFlow does not support behavior language integration.

SysML [Wei06, FMS11] is a collection of graphical modeling languages for the development of complex software systems. It is based on a subset of extended UML [OMG10] and comprises languages to describe requirements, structure, and behavior of systems. Structure is defined in block definition diagrams, internal block diagrams, and package diagrams. The internal block diagram language contains components (called “parts”), connectors, and ports. Hence it is similar to MontiArc models. Behavior is modeled with activity diagrams, sequence diagrams, state machine diagrams, and use case diagrams. In contrast to SysML, the semantics of the MontiArcAutomaton ADL yields well-defined semantics based on the FOCUS framework. The SysML semantics is grounded in the employed code generator.

Few other ADLs, for instance ArchJava [ACN02] or Plastik [JBCG05], are implemented as domain-specific embedded languages [Hud96, vDKV00]. This usually allows reusing arbitrary concepts of the host language, including loops and conditional expressions, but limits their application to platforms supporting the host GPL.

Research in robotics software engineering produced a number of approaches to modular robot components [VNE⁺01, Bru01, BKM⁺05, BBC⁺07, QGC⁺09, NFBL10]. Simultaneously, research towards model-driven robotics software engineering has received attention [Mur02, WICE03] as well: by now, there are specific modeling languages for imperative and event-driven robot programming [MAHR10, BDHN10, LW11, ASH⁺12, BBH13], kinematics and geometric relations [SCS07, FBC11, LSGB12], definition of assembly tasks [THR⁺13, Van13], perception tasks [Blu13, Hoc13], and software architectures [SSL11, Tro11, DKS⁺12, NW12, BKH⁺13].

Established robot architecture programming frameworks such as CARMEN [MRT03], Fawkes [NFBL10], OpenRDK [CCIN08], ROS [QGC⁺09], or YARP [FMN08] require definition of software architectures in terms of GPLs. Therefore, they lack abstraction, comprehensibility, and reuse of frameworks employing ADLs.

Many popular robotics architecture modeling frameworks [RMT14a], such as SmartSoft [SSL11], SafeRobots [RMT14b], RobotML [DKS⁺12], or BRICS [BKH⁺13] enable seamless development of robotics software architectures. These frameworks provide solutions to domain-specific issues, such as advanced communication patterns, deployment, or planning that are not tackled by MontiArcAutomaton. Although most of these frameworks employ state of the art language workbenches, they neither support integration of application-specific behavior languages, nor code generator composition. The authors of [SSL11] explicate this as “freedom from choice” to support application developers in creating solutions instead of dealing with infrastructure mechanisms. If the employed language workbenches however support language aggregation, inheritance, and embedding (or similar mechanisms [EGR12]) the presented language integration activities might be applied to these frameworks as well.

Most robotics ADLs employ notions of components and connectors to describe the structure of the system under development and require development of component be-

havior with GPLs [Tro11, SSL11, BKH⁺13]. The RobotML [DKS⁺12] modeling language for design and deployment of robot applications uses finite state machines to describe component behavior, which is similar to the MontiArcAutomaton version presented in [RRW13b]. In contrast to MontiArcAutomaton, the RobotML does not provide facilities to extend it with different DSLs for data types or component behavior. The AMARSi language family [NW12] enables modeling of “Motor Skill Architectures” and therefore models software components with behavior as well. Component behavior is modeled in terms of differential equations. AMARSi also does not consider modeling language extension. The BRICS [BKH⁺13, VKB14] metamodel of component & connector software architectures focuses on the separation of concerns between development responsibilities. Therefore, the metamodel explicates features like scheduling, monitoring, and component configuration. BRICS does not consider behavior modeling yet and thus relies on GPL component behavior implementations as well. DiasSpec [CKS11] is a modeling language for development of Sense/Compute/Control [TMD09] software architectures that supports modeling of structural aspects as well. Similar to previous approaches, component behavior has to be provided in terms of GPL artifacts. The OpenRTM [ASK⁺05, ASK08] ADL provides a sophisticated component model, where components may contain state machines that use C++, Java, Python, or C# expressions to enable and fire transitions. This notion of state machines is tightly coupled to the component model and corresponding infrastructure parts, and OpenRTM does not support using different component behavior languages. The data types of OpenRTM are specified as CORBA (Common Object Request Broker Architecture) [Sie00] types which enables platform-independence of data types. Orocos [Bru01, KSB10] is another modeling language for robot software architectures which supports component behavior implementation in the GPLs C++, Python, Lua. The latter enables describing component behavior with Lua DSELs, which is employed to some degree by the FAWKES [NFB10] framework as well.

Overall, to the best of our knowledge none of the existing approaches towards software architecture modeling support the minimally invasive language integration features as provided by MontiCore and hence do not support easy integration of application-specific component behavior modeling languages. Most ADLs also are tied to in-extensible frameworks supporting code generation for only a few target languages – sometimes due to the employed type system being one of a GPL.

Chapter 5

A Behavior Language with I/O^ω Automata

MontiArcAutomaton comprises the AUTOMATA modeling language based on the I/O^ω automata paradigm [Rum96] to describe the state-based input-output behavior of components platform-independently. Using AUTOMATA models to describe component behavior decouples components from GPLs and enables their reuse with different platforms. A previous version of this language was presented, where the automata language elements were part of the MontiArcAutomaton grammar [RRW14a]. The language presented in the following is a stand-alone language for I/O^ω automata, which is integrated into MontiArcAutomaton using the language integration mechanism presented in Section 4.2. Furthermore, AUTOMATA embeds the Java/P modeling language to describe expressions, which is a MontiCore language that resembles Java 1.5 [Sch12]. These enhancements have been initially developed in a Master's thesis [Sch14] and have been refined since then.

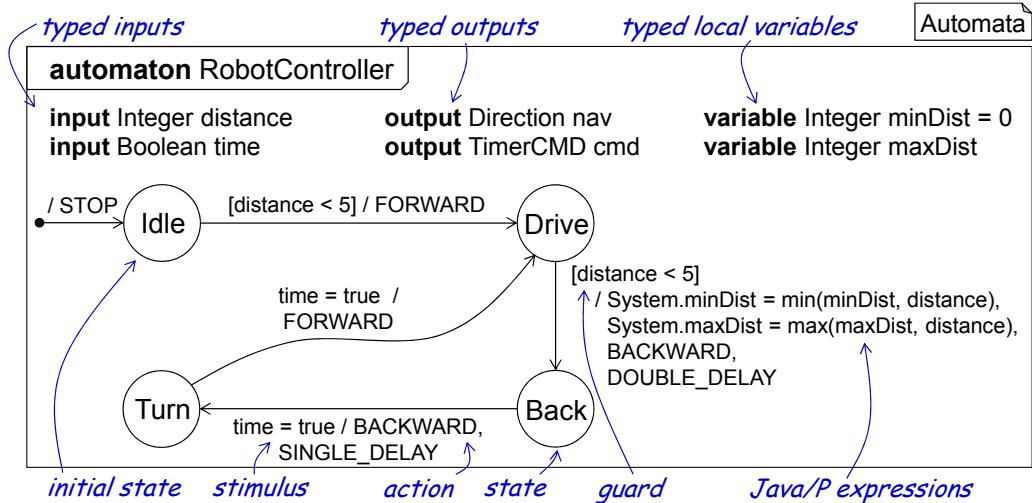


Figure 5.1: The stand-alone automaton `RobotController` demonstrates most features of the AUTOMATA language.

Figure 5.1 illustrates the features of the AUTOMATA language with the automaton `RobotController`. This automaton could be used to describe the input-output behavior of the atomic component `StateBasedController` as illustrated in Figure 4.1.

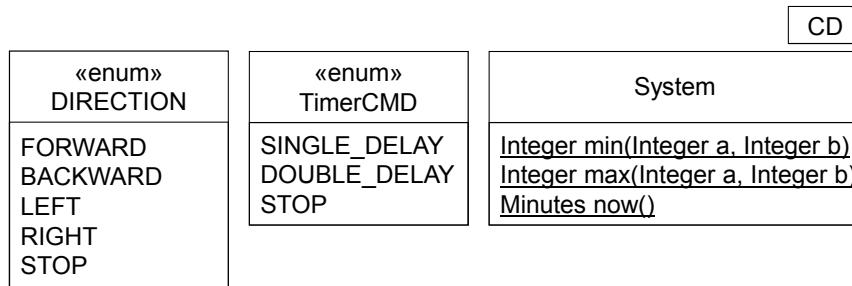


Figure 5.2: The data types the automaton RobotController operates on.

The automaton operates in the context of the class diagram types `Direction`, `TimerCMD`, and `System` (depicted in Figure 5.2). It also defines inputs `distance` and `time`, outputs `nav` and `cmd`, and variables `minDist` and `maxDist` of the corresponding CD data types. The automaton consists of the four states `Idle`, `Drive`, `Back`, and `Turn` that describe the different states the exploring robot can assume (as motivated in Section 2.4). Initially, the robot stops. Once it is signaled to start by simulating an obstacle¹, it starts to drive forward until it encounters an obstacle (i.e., the distance measured is less than 5). Then, it updates the variables `minDist` and `maxDist` using corresponding methods provided by the class diagram type `System`. The variables are updated in order of occurrence in the textual model. Afterwards, it sends `BACKWARD` to the output `nav`, and `DOUBLE_DELAY` to output `cmd`. Again, the assignments are performed in the order of occurrence. Embedded into the component `StateBasedController` of Figure 4.1, this would invoke the timer to count down an interval of twice its normal interval length and steer the robot backwards. After it receives `true` from the input `time`, (i.e., the timer signaled that the set interval has passed) it first sends `BACKWARD` to the output `nav`, and then `SINGLE_DELAY` to output `cmd`. In the context of component type `StateBasedController`, this would rotate the robot for a normal interval length. Finally, once it again receives `true` on input `time` (i.e., the set single interval has passed) it sends `FORWARD` to output `nav`.

The automaton `RobotController` illustrates most features of AUTOMATA: It distinguishes between typed inputs, typed outputs, and typed, local, variables. All of these may reference class diagram types. Variables may be initialized with appropriate values. Transitions may feature a guard (denoted by square brackets “[. . .]”), stimuli that describe expected values of inputs, and actions that assign values to outputs. Each of these is optional and may use Java/P expressions. The transition from `Idle` to `Drive`, for instance, features a guard that checks the value of input `distance` and an action that assigns the value `FORWARD`. The output that the value `FORWARD` is assigned to is determined automatically based on the type `Direction` of `FORWARD` and the available outputs and variables. In case there is only a single output or variable of type `Direction`, the assignment’s name can be omitted. The same holds for inputs. For the assignments to variables `minDist` and `maxDist` on the transition from `Drive` to

¹In lieu of integrating additional activation hardware.

Back, this is not possible, as the value returned by method `min()` is of type `Integer` and can be assigned to both variables.

Listing 5.1 displays the textual model of the `RobotController` automaton. Similar to MontiArc and MontiArcAutomaton, AUTOMATA models are structured in packages (l. 1) and support to import types from class diagram models (ll. 3-5). Automaton definitions start with the keyword `automaton` (l. 7) followed by the automaton's name. The automaton's context is defined in terms of inputs, outputs, and variables (ll. 8-13). Each consists of a type and a name, where the type is an (un-)qualified reference to an imported class diagram data type. Variables may also have a value or are left uninitialized initially. For the latter, AUTOMATA assumes default values similar to popular GPLs. The automaton defines its content in terms of states and transitions in arbitrary order. States are defined by the keyword `states` followed by a set of names (l. 15). Initial states are, following the notation of [Rum96], declared explicitly by the keyword `initial`, followed by a set of names (l. 17). Initial state definitions may declare initial behavior – such as initially assigning the value `STOP` to output `nav` (l. 17). AUTOMATA supports multiple (initial) state declarations for structuring purposes.

Transitions require no special keyword, but are defined by their unique syntax instead: a transition consists of a source state name, a target state name, a guard, multiple stimuli, and multiple actions (ll. 19-29). Stimuli and actions may refer to imported data types. For instance, the action `minDist = System.min(...)` refers to the imported data type `System` as depicted in Figure 5.2. Everything but the source state name is optional and specifying a transition by a single name describes an unconditional loop from the state to itself. Stimuli and actions may be contained in optional curly brackets for structuring purposes (ll. 22-27).

In the following, Section 5.1 introduces the language elements of AUTOMATA by example. Afterwards, Section 5.2 presents the symbol table structure and entries of AUTOMATA. Section 5.3 describes AUTOMATA's context conditions which rely on the symbol table entries. Section 5.4 presents a post-processing model transformation before Section 5.5 describes the integration of AUTOMATA into MontiArcAutomaton.

5.1 Language Elements

AUTOMATA provides language elements to describe non-hierarchical, finite automata with underspecification that operate in the context of inputs, outputs, and variables. Inputs represent the accessible environment, outputs represent parts of the environment that can be effected, and variables are part of the automaton's state. In this context, automata describe state-based input-output behavior and, hence, can read values from inputs and variables and assign values to outputs and variables. The intuition of AUTOMATA semantics is that they are performed in cycles, where they read all current values on their inputs and variables, perform up to one transition, and write the next values to their outputs and variables. They are not integrated into components or connected to other automata. Hence, neither timing, nor communication feedback cycles [HRR12] are issues with this language. These become important issues when em-

```
1 package robot;
2
3 import Direction;
4 import TimerCMD;
5 import System;
6
7 automaton RobotController {
8     input Integer dist;
9     input Boolean time;
10    output Direction nav;
11    output TimerCMD cmd;
12    variable Integer minDist = 0;
13    variable Integer maxDist;
14
15    states Idle, Drive, Back, Turn;
16
17    initial Idle / STOP;
18
19    Idle -> Drive [distance < 5] / FORWARD;
20
21    Drive -> Back [distance < 5]
22        / { minDist = System.min(minDist, dist),
23              maxDist = System.max(maxDist, dist),
24              BACKWARD,
25              DOUBLE_DELAY };
26
27    Back -> Turn true / { BACKWARD, SINGLE_DELAY };
28
29    Turn -> Drive true / FORWARD;
30 }
```

Automata

Listing 5.1: The textual model of the automaton `RobotController` depicted in Figure 5.1.

bedding AUTOMATA models into components and, thus, are discussed in Section 5.5.1. This section introduces the AUTOMATA language elements by example, which can be used with models of different semantics. The complete AUTOMATA grammar is available in Section A.2 of the appendix in two versions: one simplified for human comprehension and the actual MontiCore grammar.

5.1.1 Automaton Declaration

An automaton declaration begins with the keyword `automaton` followed by the automaton's unqualified name and curly brackets. The latter contain the automaton's context consisting of inputs, outputs, and variables, and its content comprising states and transitions. Listing 5.2 shows the declaration of the automaton `MotorController`.

```

1 automaton MotorController {
2   // ...
3 }
```

Automata

Listing 5.2: The declaration of the automaton MotorController begins with the keyword `automaton` followed by its name and curly brackets.

5.1.2 Inputs, Outputs, and Local Variables

An automaton may define the inputs, outputs, and variables to describe the context it operates in. Each consists of a keyword, a type, and a name. Inputs start with the keyword `input`, outputs with the keyword `output`, and variables with the keyword `variable`. Listing 5.3 depicts the automaton `TimedMotorController`, which defines three inputs (ll. 2-4), one output (l. 5), and two variables (ll. 6-7).

```

1 automaton TimedMotorController {
2   input    Minutes ts;
3   input    Float cur;
4   input    Float max;
5   output  Float speed,
6   variable Boolean stopLights;
7   variable Minutes t0 = System.now();
8 }
```

Automata

Listing 5.3: The automaton `TimedMotorController` defines three inputs, one output, and two variables (ll. 2-7).

Where initial values for variables are omitted, the values default to specific values similar to Java: numerical values default to 0, Boolean values default to `false`, and complex objects default to `null`.

5.1.3 Values

Values checked by guards and inputs or assigned to outputs are expressions of the Java/P language. This includes values and literals for common types, such as `true` and `false` for Boolean types, various number formats, and strings, and complex expressions over CD types, such as concatenated method calls with multiple parameters. Listing 5.4 illustrates this with initial values for the automaton's variables `stopLights` and `t0` (ll. 6-7). The first assignment assigns the literal value `false` to `stopLights` and the second assignment retrieves and assigns the current time via method `now()` of data type `System` (as depicted in Figure 5.2).

Additionally, AUTOMATA introduces the (pseudo) value `NoData`, denoted by “`--`”, to describe the case of explicitly assigning *no message* in architectures using timed stream semantics [RR11, RRW14a].

```
1 automaton InitializedMotorController {
2   input      Minutes ts;
3   input      Float cur;
4   input      Float max;
5   output    Float speed,
6   variable Boolean stopLights = false;
7   variable Minutes t0 = System.now();
8 }
```

Automata

Listing 5.4: The automaton `InitializedMotorController` initializes its variables (ll. 6-7).

5.1.4 State Declarations, Initial States, and Initial Outputs

A state declaration introduces one or multiple states and begins with the keyword `states`. An automaton must contain at least one state declaration with at least one state. Defining multiple state declarations is allowed. They are interpreted as a single state declaration. The automaton `StatebasedMotorController` of Listing 5.5 features two state declarations (ll. 9-10) defining multiple states each. Each state declaration begins with the keyword `states`, followed by a list of state names. Each state may be preceded by an optional list of stereotypes (cf. state `Error` in l. 9). Such stereotypes facilitate language extension to a certain degree but lack the documentation of language elements codified in the grammar. In MontiArcAutomaton, stereotypes are provided as means to extend the language in-place for developers of model processors (such as code generators). Consequently, evaluation of stereotypes is left to model processing tools.

```
1 automaton StatebasedMotorController {
2   input      Minutes ts;
3   input      Float cur;
4   input      Float max;
5   output    Float speed,
6   variable Boolean stopLights;
7   variable Minutes t0 = System.now();
8
9   states Idle, Stopping, <>log>> Error;
10  states Accelerating, Decelerating;
11
12  initial Idle, Stopping / { speed = 0, true };
13 }
```

Automata

Listing 5.5: The automaton `StatebasedMotorController` introduces five states to operate on (ll. 9-10).

Following the notion of initial states in [Rum96], they are declared separately. Initial state declarations begin with the keyword `initial` followed by a list of state names and

a single initial output per initial state declaration. Listing 5.5 declares that the states `Idle` and `Stopping` are initial and that starting with one of these states initially assigns 0 to `speed` and the value `true` to `stopLights` (l. 12). Defining multiple initial states is a form of underspecification that must be realized by the responsible code generators accordingly. Again, AUTOMATA derives that `true` should be assigned to `stopLights` via type matching and the curly brackets are optional.

5.1.5 Transitions

A transition declares a source state from which it originates, a target state the automaton reaches if the transition's guard holds and its stimuli are satisfied, and actions in terms of assignments. Generally, transitions have the form

```
Source -> Target [Guard] {Stimulus} / {Action};
```

where `Source` and `Target` are names of the automaton's states. The guard is a Boolean Java/P expression over the automaton's context. `Stimulus` is a set of valuations of the form `name = expression`, where `name` must be the name of an input or of a variable and multiple valuations are separated by commas. `Action` is a set of assignments of the form `name = expression`, where `name` must be the name of an output or of a variable and multiple assignments are separated by commas as well. The right-hand side of valuations and assignments (`expression`) are Java/P expressions and may reference data types. As target, guard, inputs, and outputs are optional, `Source`; is a valid transition as well. This transition describes an unconditional loop from `Source` to itself that neither reads values, nor assigns values. If the square brackets of a guard are present, the guard may not be empty. The same holds for the curly brackets of inputs and outputs.

Listing 5.6 illustrates part of a system to prohibit truck drivers from skipping breaks by controlling a truck's speed based on the current maximum speed and the time since the last break. To this effect, the automaton `TruckMotorController` defines multiple transitions to describe when the system should accelerate, decelerate, and stop (ll. 14-33). The first transition (ll. 14-15) defines that the automaton should switch from state `Idle` to `Accelerating` when the current speed (provided via input `cur`) is less than the current maximum speed (via input `max`) and the truck is driving for less than 240 minutes. In that case, it should increase speed by 10 and not flash the stop lights. To check this, it defines a guard with a Boolean Java/P expression over inputs `cur`, `max`, and variable `t0`. If the guard expression holds, the two assignments `speed = cur + 10` and `stopLights = false` are applied and the automaton switches to state `Accelerating`. The transition from state `Decelerating` to state `Waiting` (l. 26) omits a guard but is enabled if the valuation `cur = 0` holds, i.e., if the truck is standing. If this holds, it switches to `Waiting` without assigning values to outputs or variables. After waiting 60 minutes, the transition from `Waiting` to `Accelerating` (ll. 28-31) defines that the truck starts driving again, it deactivates its stop lights, and the count down to the next break starts again. The last transition (l. 33) denotes a loop from state `Accelerating` to itself that increases speed if slower than the current speed limit.

```
1 automaton TruckMotorController {
2   input      Minutes ts;
3   input      Float cur;
4   input      Float max;
5   output     Float speed,
6   variable   Boolean stopLights;
7   variable   Minutes t0 = System.now();
8
9   states    Idle, Stopping, <<log>> Error;
10  states    Accelerating, Decelerating;
11
12 initial  Idle, Stopping / { speed = 0, true };
13
14  Idle -> Accelerating [cur < max && t0 - ts <= 240]
15      / speed = cur + 10, false;
16
17  Accelerating -> Decelerating [cur >= max]
18      / speed = cur - 10, true;
19
20  Accelerating -> Decelerating [t0 - ts > 240]
21      / speed = cur - 10, true;
22
23  Decelerating -> Decelerating [t0 - ts > 240]
24      / speed = cur - 10, true;
25
26  Decelerating -> Waiting cur = 0;
27
28  Waiting -> Accelerating [t0 - ts >= 60]
29      / { speed = cur + 10,
30            false,
31            t0 = System.now() };
32
33  Accelerating [cur < max] / speed = cur + 10, false;
34
35  // ...
36 }
```

Automata

Listing 5.6: The automaton `TruckMotorController` contains multiple states and transitions to control a truck within speed limits.

5.1.6 Alternative Stimuli

Alternatives enable stimuli to react on different input values similar to a logical disjunction. While this behavior can be emulated with multiple transitions, alternatives reduce the number of required transitions and hence improve the model's comprehensibility. Listing 5.7 illustrates an alternative in the stimulus of the automaton's first transition (l. 8). This transition is enabled, if the current value at mode is either 1 or 2. Specifying

the valuations name (i.e., mode) is optional if it can be identified unambiguously via its type. The values in alternatives may be sequences of expressions, nested alternatives are however prohibited.

```

1 automaton BinaryMotorController {
2   input Integer mode;
3   output Boolean brake;
4
5   states Idle, Accelerating;
6   initial Idle, Stopping / true;
7
8   Idle -> Accelerating mode = alt{1,2} / false;
9   // ...
10 }
```

Automata

Listing 5.7: The automaton `BinaryMotorController` uses an alternative (l. 9) to enable a transition for alternative values.

5.2 Symbol Table

The AUTOMATA symbol table serves two purposes: it facilitates checking the well-formedness of models as its entries capture the essence of models without the AST infrastructure. This allows using automata in conjunction with other models without combining their ASTs (and thus hindering evolution of the participating languages). To this effect, it consists of multiple entry types that represent parts of automata or data types in terms of the AUTOMATA language. Providing its own type system decouples it from existing type systems, yet enables easy integration via adaptation [LNPR⁺13, HLMSN⁺15]. Similar to visibilities in object-oriented programming languages, where selected items of a class are hidden from the remaining system (for instance using the keyword `private`), the AUTOMATA symbol table hides initial state outputs, guards, stimuli, and actions.

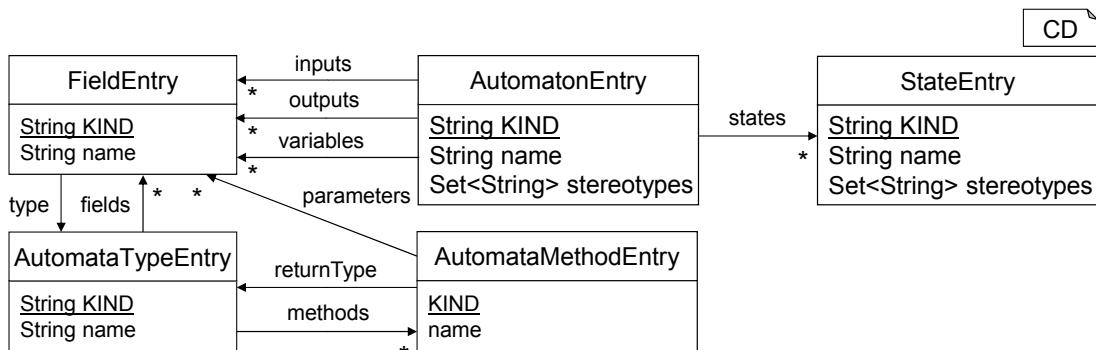


Figure 5.3: An excerpt of the AUTOMATA symbol table entries.

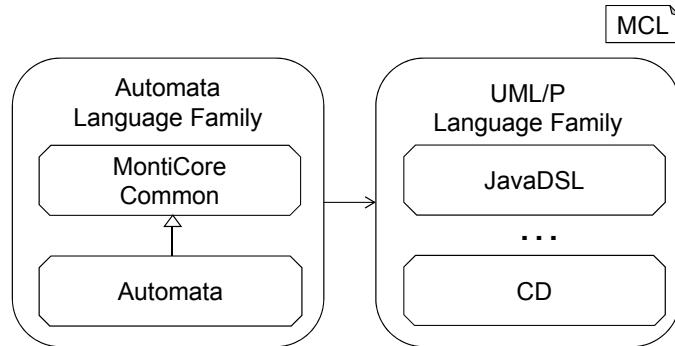


Figure 5.4: The AUTOMATA language family.

The main symbol of AUTOMATA models is the `AutomatonEntry` depicted in Figure 5.3 that contains the most important elements of an automaton. It features a set of states in form of `StateEntry` instances, which have a name, and indicate whether the represented state is initial and whether it has an initial output. The `AutomatonEntry` also contains the automaton's context in terms of `FieldEntry` instances that are either inputs, outputs, or variables. Each `FieldEntry` has a type represented by an `AutomataTypeEntry` that has a name, methods, and fields. The latter are `FieldEntry` instances again.

AUTOMATA employs an object-oriented type system. Adaptation enables integration with the MontiArcAutomaton ADL as well as employing the Java/P type checking mechanisms. Therefore, analyses, such as context condition checks, can reason over such types without being closely tied to the Java/P type system. The Java/P mechanisms are unaware of AUTOMATA type entries and hence, adaptation is required to exploit the Java/P well-formedness checks. The AUTOMATA language family is depicted in Figure 5.4. It combines the AUTOMATA language with Java/P and class diagrams of UML/P. AUTOMATA extends the MontiCore COMMON language, which provides various fundamental productions (such as types and literals).

The AUTOMATA language also allows using class diagram types and provides corresponding adapters from and to class diagram type entries as well. This approach uses symbol table entries only, thus changes to a language's syntax are transparent to language integration. Furthermore, integration of other data types languages, such as C++ or Python, for expressions and types, requires proper adapters only. The adaptation from AUTOMATA inputs, outputs, and variables to MontiArcAutomaton ports and variables follows the same pattern and employs adapters between AUTOMATA field entry and ports and between AUTOMATA type entries and MontiArcAutomaton type entries. That adaptation is part of the behavior language integration explained in Section 4.2. Prior to that, the intra-language context conditions check well-formedness of stand-alone AUTOMATA models based on their AST and symbol table entries. The next section presents these context conditions.

5.3 Context Conditions

Well-formedness of AUTOMATA models is checked with context conditions that rely on the checked model's AST and symbol table entries. These context conditions are based on the context conditions developed for a previous version of this language [RRW14a] and have been refined in a Master's thesis [Sch14]. Similar to the MontiArcAutomaton ADL context conditions, they are grouped into four categories as well [RRW14a].

5.3.1 Uniqueness Conditions

The uniqueness conditions of AUTOMATA guarantee that the names of language elements are unique and restrict use of modeling elements where appropriate. This, for instance, includes prohibiting multiple states, inputs, outputs, and variables of the same name. Violation of uniqueness conditions produces errors unless noted otherwise.

AU1: The name of each state is unique.

Defining multiple states of the same name is a source of errors, especially when used in conjunction with different stereotypes. To avoid such confusion, AUTOMATA prohibits models with multiple states of the same name. Listing 5.8 shows an automaton with two state declarations (ll. 4-5) that each define a state named `Explaining`. Furthermore, the third state declaration (l. 6) defines two states of name `Idle`.

```

1 automaton HRI {
2   input String userInput;
3   output String systemOutput;
4   states Explaining; ✖ // Duplicate state 'Explaining'.
5   states Explaining; ✖ // Duplicate state 'Explaining'.
6   states Idle, Idle ✖ // Duplicate state 'Idle'.
7   initial Idle;
8 }
```

Automata

Listing 5.8: The automaton `HRI` defines multiple states of the same name (ll. 4-6).

AU2: Each state is declared initial at most once.

Each state should be declared initial at most once. As declaring a state initial multiple times does not obstruct the meaning of it being initial, this context condition produces a warning only. Listing 5.9 illustrates the automaton `Logging` that contains three initial state declarations (ll. 4-6). The first declaration designates the state `File` to be initial twice and hence produces a warning accordingly. The second and third initial state declarations (ll. 5-6) each declare the state `Cloud` to be initial and therefore produce warnings as well.

```
1 automaton Logging {  
2   input String message;  
3   states Screen, File, Cloud;  
4   initial File, File;  // 'File' multiply initial.  
5   initial Cloud;  // 'Cloud' multiply initial.  
6   initial Cloud;  // 'Cloud' multiply initial.  
7 }
```

Automata

Listing 5.9: The automaton Logging declares the states File (l. 4) and Cloud as initial multiple times (ll. 5-6).

AU3: The names of all inputs, outputs, and variables are unique.

Defining multiple inputs, outputs, or variables of the same name leads to underspecification when using these in stimuli or actions. Hence, it is prohibited to have inputs, outputs, and variables of the same name. Listing 5.10 depicts a violation of this context condition: The automaton Buffer uses the name data for an input (l. 2), for an output (l. 3), and for a variable (l. 4).

```
1 automaton Buffer {  
2   input Integer data;  // 'data' multiply defined.  
3   output Boolean data;  // 'data' multiply defined.  
4   variable String data  // 'data' multiply defined.  
5   states Buffering;  
6   initial Buffering;  
7 }
```

Automata

Listing 5.10: The automaton Buffer defines an input (l. 2), an output (l. 3), and a variable (l. 4) of the same name.

5.3.2 Convention Conditions

The convention conditions of AUTOMATA check the well-formedness of names and automata structure. Convention conditions produce warnings unless stated otherwise.

AC1: The automaton has at least one input.

Automata without inputs are rarely useful and omitting inputs might hint at issues. Hence, AUTOMATA produces warnings for models without inputs as illustrated in Listing 5.11. The depicted automaton ColorSensor defines no inputs and consequently, AUTOMATA produces a warning (l. 1).

```

1 automaton ColorSensor { ⚠ // No inputs.
2   output Color color;
3   states Offline, Measuring;
4   initial Offline;
5 }
```

Automata

Listing 5.11: The automaton ColorSensor provides no inputs.

AC2: The automaton has at least one output.

AUTOMATA models without outputs cannot produce observable output behavior and hence are also rarely useful. Therefore, AUTOMATA prohibits such models as well. Listing 5.12 displays an automaton without inputs and the corresponding warning (l. 1).

```

1 automaton BinaryMotor { ⚠ // No outputs.
2   input Boolean run;
3   states Offline, Online;
4   initial Offline;
5 }
```

Automata

Listing 5.12: The automaton BinaryMotor declares no outputs.

AC3: The automaton has at least one state.

Trivially, automata without states cannot produce any input-output behavior. Thus, AUTOMATA produces an error for such models. Listing 5.13 shows an automaton without states and the corresponding error. The automaton StatelessBuffer defines two inputs (ll. 2-3) and an output (l. 4), but no states.

```

1 automaton StatelessBuffer { ✗ // No states.
2   input Boolean store;
3   input Data data;
4   output Data result;
5 }
```

Automata

Listing 5.13: The automaton StatelessBuffer defines no states.

AC4: The automaton has at least one initial state

AUTOMATA models must define at least one initial state. Otherwise, the automaton will never activate, and hence, produce no output behavior. This context condition also produces an error. The automaton MapBuilder given in Listing 5.14 defines multiple

inputs and outputs (ll. 2-5) and a single state `Storing` (l. 6). It omits declaration of any initial states and hence, AUTOMATA produces the depicted error (l. 1).

```
1 automaton MapBuilder { x // No initial states.
2   input Pose pose;
3   input Obstacle obstacle;
4   input Float probability;
5   output Map updatedMap;
6   states Storing;
7 }
```

Automata

Listing 5.14: The automaton `MapBuilder` declares no initial states.

AC5: The automaton's valuations and assignments use only allowed Java/P modeling elements.

The Java/P modeling language allows recreating all language elements of Java 1.5. Hence, it features control structures, conditional expressions, and further elements unsuitable for stimuli and actions. This context condition therefore restricts the right-hand sides of valuations and assignments to the following well-defined expressions, which are:

- Array access: `data[0]`
- Class cast: `(Float) data[0]`
- Field access: `data.length`
- Infix expression: `data[0] + 1`
- Object instantiation: `new Integer(3)`
- Method invocation: `data.clone() [0]`
- Parenthesized expression: `(data[0] + 1)`
- Prefix expression: `++data[0]`
- Names: `x`
- and all type-compatible values.

Where the examples hold for a `Float` input, output, or local variable `x` of the automaton and an array `data` of type `Integer`. AUTOMATA reuses all Java/P context conditions related to these expressions.

Listing 5.15 illustrates the error resulting from using a prohibited expression with the automaton `PersonFollower`. The automaton contains a transition from its state `Idle` to its state `Following` that is enabled if the value of `startFollowing` is true. It uses an embedded conditional expression of the Java/P as assignment to its output speed (ll. 9-10).

```

1 automaton PersonFollower {
2   boolean startFollowing;
3   input Float dist;
4   output Integer speed;
5   states Idle, Following, Lost;
6   initial Idle;
7
8   Idle -> Following true
9     / speed = if (dist < 2) { speed; } else { speed + 2 } ✗ // Prohibited
10    // expression.
11  // ...
12 }
```

Automata

Listing 5.15: The automaton `PersonFollower` uses a prohibited Java/P conditional expression for an assignment (ll. 9-10).

AC6: The names of automata start with capital letters.

Each AUTOMATA model defines a specific automaton type of the modeled structure. Hence, similar to components in MontiArcAutomaton or classes in Java, AUTOMATA expects these to start with a capital letter. If the name starts with a lower-case letter, AUTOMATA issues a warning as depicted in Listing 5.16.

```

1 automaton robotArm { ⚠ // Automata start upper-case
2   // ...
3 }
```

Automata

Listing 5.16: The automaton's name begins with a lower-case letter (l. 1).

AC7: The names of inputs, outputs, and variables start with a lower-case letter.

Inputs, outputs, and variables resemble ports of the MontiArcAutomaton ADL or fields in GPLs. Consequently, AUTOMATA expects these to start with a lower-case letter. Inputs, outputs, and variables starting with a capital letter produce warnings accordingly. Listing 5.17 illustrates the warnings arising from inputs (l. 2), outputs (l. 3), and variables (l. 4) with names that begin with capital letters.

AC8: State names begin with a capital letter.

For better comprehensibility and distinction between inputs, outputs, variables, and states, AUTOMATA expects state names to start with a capital letter. State names starting with lower-case letters produce a warning as displayed in Listing 5.18. The depicted

```
1 automaton Scheduler {
2   input Proc[] Procs;  // Inputs start lower-case.
3   output Proc Next;  // Outputs start lower-case.
4   variable Proc[] Buf;  // Variables start lower-case.
5   states Scheduling, Error;
6   initial Scheduling;
7 }
```

Listing 5.17: Automaton Scheduler contains an input, an output, and a variable of names starting with capital letters (ll. 2-4).

automaton ToastService defines two states of which the second state, toasting (l. 4), begins with a lower-case letter. For this, AUTOMATA produces a warning.

```
1 automaton ToastService {
2   input Integer numToasts;
3   output Error message;
4   states Idle, toasting;  // States start upper-case.
5   initial Idle;
6 }
```

Listing 5.18: Automaton ToastService defines the state toasting that starts with a lower-case letter (l. 4).

5.3.3 Referential Integrity Conditions

The referential integrity context conditions of AUTOMATA ensure the well-formedness of references to modeling elements. This includes checking references for missing inputs, outputs, variables, and states as well as considering their direction (e.g., for references to inputs and outputs) correctly. Referential integrity context conditions produce errors unless stated otherwise.

AR1: Names used in guards, valuations, and assignments exist in the automaton.

Names used in guards, valuations, and assignments refer to inputs, outputs, or variables. Consequently, referencing nonexistent inputs, outputs, or variables obviously is a mistake that entails incomplete and erroneous automata. Consequently, AUTOMATA rejects such models and raises errors. Listing 5.19 displays the automaton LaneFollower that contains a transition (ll. 7-8) referencing a nonexistent input or variable (whether an input or a variable is missing, cannot be inferred from the name start as it might reference to both). As AUTOMATA employs Java/P expressions on transitions, this context condition also checks names used in such expressions.

```

1 automaton LaneFollower {
2   input Color lane;
3   output Integer speed;
4   states Idle, Follow, Lost;
5   initial Idle;
6
7   Idle -> Follow [lane == RED && start] ✘ // Unknown input or
8       / speed = 3;                                // variable 'start'.
9   // ...
10 }
```

Automata

Listing 5.19: The automaton LaneFollower contains a transition from Idle to Following that contains a guard referencing the missing input or variable start (ll. 7-8).

AR2: Inputs, outputs, and variables are used correctly.

AUTOMATA models may define transitions featuring guards, stimuli, and actions. Guards and stimuli may read values from inputs and variables only, while actions may assign values to outputs and variables only. This prohibits that an automaton reads its own output as well as that it writes to its inputs. Both may lead to self-cycles and is rarely useful.

```

1 automaton CoffeeService {
2   input Integer numCoffees;
3   input Integer strength;
4   output Error msg;
5   output Pose next;
6   states Idle, Preparing, Deploying;
7   initial Idle;
8
9   Idle -> Preparing [msg == ""] ✘ // Reading from output.
10      / next = new Pose(3,4,5);
11
12  Preparing -> Idle numCoffees = 0 ✘ // Writing to input.
13      / strength = 10;
14  // ...
15 }
```

Automata

Listing 5.20: The automaton CoffeeService contains two transitions: one (ll. 9-10) reads from the output msg, the other (ll. 12-13) assigns a value to the input strength.

The automaton CoffeeService depicted in Listing 5.20 displays two transitions. The first transition (ll. 9-10) is enabled if the value of the output msg is the empty

string. This comparison is prohibited and produces the illustrated error. The second transition (ll. 12-13) assigns the value 10 to the input strength, which is also forbidden. In consequence, AUTOMATA produces a second error.

AR3: Used states exist.

Similar to inputs, outputs, and variables, the states used as initial states or referenced by a transition must exist for the automaton to be well-formed. The context condition AR3 ensures this by producing errors for initial state declarations and transitions using undefined states. Listing 5.21 shows the resulting errors from declaring the nonexistent state `Idle` to be initial (l. 5) and from using this state in a transition (l. 7).

```
1 automaton WindowController {  
2   input Float temp;  
3   output WindowState state;  
4   states Working;  
5   initial Idle; ✖ // Inexistent state 'Idle'.  
6  
7   Working -> Idle [temp < -100]; ✖ // Inexistent state 'Idle'.  
8   // ...  
9 }
```

Automata

Listing 5.21: The automaton `WindowController` uses the nonexistent state `Idle` in an initial state declaration (l. 5) and in a transition (l. 7).

AR5: Types of valuations and assignments without names are unambiguous.

AUTOMATA allows omitting the names of valuations in stimuli and the names of assignments in actions if the type of the right-hand side matches unambiguously to the type exactly one corresponding input, output, or variable. In case this is ambiguous, AUTOMATA cannot derive the intended names and raises errors. Listing 5.22 shows two erroneous transitions with ambiguous assignments: the stimulus of the first transition (l. 11) is enabled if the value `true` holds. However, `true` is of type Boolean and hence may be read from the input `start` (l. 2) or from the variable `paused` (l. 5). Consequently, AUTOMATA produces an error (l. 11). The action of the second transition (l. 12) assigns the values 1 and 2. This is ambiguous as well as both may be assigned either to the outputs `x`, `y` (ll. 3-4), or to the variable `step` (l. 6). Therefore, AUTOMATA produces another error (l. 12).

5.3.4 Type Correctness Conditions

The type correctness context conditions of AUTOMATA ensure correct usage of typed elements. This includes inputs, outputs, variables, valuations, and assignments. Violation of type correctness conditions raises errors.

```

1 automaton AssemblyController {
2   input Boolean start;
3   output Integer y;
4   output Integer x;
5   variable Boolean paused;
6   variable Integer step;
7
8   states Init, Work;
9   initial Init;
10
11  Init -> Work true / x = 3, y = 4; ✗ // Ambiguous stimulus.
12  Work -> Init paused = true / 1, 2; ✗ // Ambiguous action.
13 }
```

Automata

Listing 5.22: The automaton AssemblyController contains a transition with an ambiguous stimulus (l. 11) and another transition with an ambiguous action (l. 12).

AT1: Guard expressions evaluate to a Boolean truth value.

The guards of transitions represent logical expressions that restrict when transitions are enabled. As such, they ultimately must evaluate to a truth value. As AUTOMATA operates on binary truth values, its transition guards must evaluate to a Boolean value. The IrrigationController depicted in Listing 5.23 contains a transition from state Working to itself (l. 7). This transition features with a guard that does not evaluate to a Boolean truth value and is erroneous as marked by AUTOMATA. The evaluation is performed using the Java/P type checking framework.

```

1 automaton IrrigationController {
2   input Float aridity;
3   output Boolean water;
4   states Working;
5   initial Working;
6
7   Working [aridity + 1] / true; ✗ // Non-Boolean guard.
8 }
```

Automata

Listing 5.23: The automaton IrrigationController contains a transition with non-Boolean guard (l. 7).

AT2: Types of valuations and assignments must match the type of the assigned input, output, or variable.

The types of valuations of stimuli must conform to the type of the referenced input or variable. Similarly, the types of assignments of actions must conform to the type of the

referenced output or variable. To ensure this, AUTOMATA evaluates the right-hand side of valuations and assignments and compares the resulting type to their left-hand side using the type checking framework of the Java/P (see Section 2.2.1).

```
1 automaton ElevatorCabinController {
2   input Integer floor;
3   input Boolean f11Pressed;
4   input Boolean f12Pressed;
5   output Direction dir;
6   variable Integer prevFloor;
7
8   states Wait, Drive;
9   initial Wait / prevFloor = "N/A"; ✖ // Incompatible assignment.
10
11  Wait -> Drive floor = "1", ✖ // Incompatible valuation.
12    f11Pressed = true
13    / dir = UP,
14    prevFloor = 1;
15
16  Wait -> Drive floor = 2,
17    f11Pressed = true
18    / dir = true, ✖ // Incompatible assignment.
19    prevFloor = 2;
20  // ...
21 }
```

Automata

Listing 5.24: Automaton ElevatorCabinController contains two incompatible assignments (ll. 9,18) and one incompatible valuation (ll. 12-14).

The automaton ElevatorCabinController of Listing 5.24 features an initial output and two transitions to describe the behavior of an elevator cabin running between two floors. The initial output of state Waiting (l. 9) assigns the String value "N/A" to the Integer variable prevFloor. The first transition (ll. 11-14), features a stimulus with an invalid valuation as it compares the expression "1" of type String with the input floor of type Integer. The second transition (ll. 16-19) features an action with an incompatible assignment to the output dir of type Direction, which it assigns the Boolean value true to.

AT3: The special literal value NoData is not used for variables.

The special literal NoData (denoted “--”) represents the absence of messages between two time slices for timed streams (cf. Section 2.4). Its use with variables is prohibited. Listing 5.25 shows the automaton AlarmController that controls a time-based alarm system that supports manual overriding (input over) and stores the latest override in the variable latest. The automaton features an initial output (l. 8) and two transitions (ll. 10-13) that violate this context condition: The initial output and the first transition

(ll. 10-11) assign NoData to latest. The second transition (l. 13) reads NoData from latest.

```

1 automaton AlarmController {
2   input Time t;
3   input Boolean over;
4   output Boolean activateAlarm;
5   variable Time latest;
6
7   states Off, On;
8   initial Off / latest = --; ✖ // Assigning -- to variable.
9
10  Off -> On [t > 1800 && !over] ✖ // Assigning -- to variable.
11    / false, latest = --;
12
13  Off -> On latest = -- / true; ✖ // Reading -- from variable.
14  // ...
15 }
```

Automata

Listing 5.25: The automaton AlarmController assigns the special literal NoData to variable latest (ll. 8, 11) and compares NoData to it (l. 13).

5.4 A Transformation on the Automata AST

AUTOMATA features a single model transformation that is applied prior to its context conditions. This transformation identifies names of stimuli and actions where these haven been omitted due to the unambiguous matching of the assigned expression's types (cf. Section 5.1.6). It determines the type of the assigned expressions via adaptation to Java/P entries and employing the Java/P type evaluation mechanism. If a name of an input, output, or variable - depending on the expression's source - of matching type is found and unambiguous, it is added to the stimulus or action accordingly. In case no matching name is found, the transformation aborts model processing.

5.5 Integrating Automata into MontiArcAutomaton

AUTOMATA models can be integrated into MontiArcAutomaton ADL components using the integration mechanisms presented in Section 4.2. Using these requires integration of an AUTOMATA non-terminal production into the MontiArcAutomaton ADL BehaviorModel extension point as well as symbolic adaption and integration of specific inter-language well-formedness rules. This enables definition of integrated models as depicted in Listing 5.26. Here, the AUTOMATA model parts embedded after behavior (ll. 16-34) describe various transitions that reference inputs and outputs to read and write data. As the embedded automaton operates in the context of a component, the

names of these inputs and outputs must be interpreted as ports and variables of the surrounding component.

```
1 package architecture;
2
3 import bumperbotmodels.types.TimerCmd;
4 import bumperbotmodels.types.TimerSignal;
5 import bumperbotmodels.types.MotorCmd;
6
7 component BumpControl {
8   port
9     in Integer distance,
10    in TimerSignal signal,
11    out TimerCmd timer,
12    out MotorCmd right,
13    out MotorCmd left;
14
15 behavior automaton {
16   state idle, driving, backing, turning;
17
18   initial idle / {right=MotorCmd.STOP, left=MotorCmd.STOP};
19
20   idle -> driving [ocl:distance < 1] /
21           {right = MotorCmd.FWD,
22            left   = MotorCmd.FWD};
23
24   driving -> backing [ocl:distance < 5] /
25           {right = MotorCmd.BWD,
26            left   = MotorCmd.BWD,
27            TimerCmd.DOUBLE};
28
29   backing -> turning {TimerSignal.ALERT} /
30           {right = MotorCmd.FWD,
31            TimerCmd.SINGLE};
32
33   turning -> driving {TimerSignal.ALERT} /
34           {left = MotorCmd.FWD};
35 }
36 }
```

MAA

Listing 5.26: The component BumpControl contains an embedded AUTOMATA model with four states and four transitions (ll. 15-35).

Integration changes consideration of the dynamic semantics of AUTOMATA performing in MontiArcAutomaton ADL components regarding FOCUS timing and modification the static semantics via integration-specific well-formedness rules. This section first discusses the semantics of AUTOMATA that are integrated into MontiArcAutomaton ADL components and describes the required integration artifacts afterwards.

5.5.1 Semantics of Integrated Automata Models

Stand-alone AUTOMATA models are executed in cycles. In each cycle, an automaton reads all values from its inputs and variables, evaluates its guards, fires up to one transition, changes state, and writes new values to its outputs and variables. As AUTOMATA are intended to describe behavior or a software part, reading values and writing values are conceived sequentially. Thus, each iteration of an automaton's cycle actually consists of an ordered sequence of computation steps.

Two different dynamic semantics for AUTOMATA models are introduced in [RRW14a]: the automata can be interpreted time-synchronously or event-driven. The former reacts once messages on all inputs are available; the latter reacts once the first input is available. For integration, we assume the time-synchronous interpretation and interpret the individual steps of an automaton's read-compute-write cycle as "untimed causally-related actions" [Lee10] happening in the superdense time of a single FOCUS time slice as introduced in Section 4.2.4. Embedded AUTOMATA models also operate in another context than stand-alone models. The former operate in a component. Thus they receive inputs via incoming ports and component variables and write outputs to outgoing ports and component variables. Enabling this requires access to ports and component variables. Embedded automata also may access the configuration parameters and type parameters of the embedding component. These changes in static semantics are described by adaptation and well-formedness rules as introduced Section 4.2.2. Their realization for automata is presented next.

5.5.2 Integration Infrastructure

Embedded AUTOMATA models reference ports and variables to read and write values. Thus declaration of automaton inputs and outputs must be prohibited and the references to these on transitions must be interpreted accordingly. We solve the former by embedding only the production `AutomatonContent` of the AUTOMATA grammar (cf. Listing A.1) into MontiArcAutomaton ADL components. As inputs, outputs, and automaton variables are part of `AutomatonContext`, which is underivable from `AutomatonContent`, prohibiting these does not require context conditions. Changing the meaning of names on transitions requires adapters from ports and component variables to inputs, outputs, and automaton variables. These adapters resemble the adapters between FSM fields and ports as depicted in Figure 4.9. With these two adapters in place, all context conditions regarding inputs, outputs, and automaton variables are automatically applied to ports and component variables. Besides specifying which production to embed and providing adapters, integration of AUTOMATA into the MontiArcAutomaton ADL also requires to specify how to create, qualify, resolve, and deserialize AUTOMATA symbol table entries. As explained in Section 4.2.2, the corresponding modules of AUTOMATA can be reused without modification. Furthermore, the workflow of the AUTOMATA AST transformation (Section 5.4) must be registered with the MontiArcAutomaton ADL family as well.

```
1 name "automaton"
2 behavior "Automata.AutomatonContent"
3 tool new AutomataTool()
4 adapter new AutomataType2TypeReferenceAdapter();
5 adapter new TypeReference2AutomataTypeAdapter();
6 adapter new Port2AutomataFieldAdapter();
7 adapter new Variable2AutomataFieldAdapter();
8 adapter new AutomataField2JavaFieldAdapter();
9 adapter new JavaType2AutomataTypeAdapter();
```

GBC

Listing 5.27: Groovy behavior configuration model for embedding part of the AUTOMATA language.

Listing 5.27 illustrates the Groovy behavior configuration model to embed AUTOMATA models into MontiArcAutomaton. After declaration of parser discriminator and model keyword `automata` (l. 1), it specifies to embed the `AutomataContent` production (l. 2) and uses the `AutomataTool` to provide information about symbol table infrastructure, context conditions, and workflows (l. 3). Afterwards, the model specifies six adapters. The first two adapters (ll. 4-5) adapt between data type symbols of both languages. The second two adapters (ll. 6-7) adapt between automaton fields and ports and variables of components. The last two adapters adapt between AUTOMATA symbols and Java/P symbols, which is required to reason about expressions on transitions.

5.6 Discussion

The AUTOMATA behavior language enables describing state-based input-output behavior of components platform-independently. The language implements a variant of I/O $^\omega$ automata [Rum96] and conforms to the FOCUS [BS01, BR07, RR11] messaging semantics. Modeling with AUTOMATA requires comprehension of few modeling elements only, but describes component behavior in a structured way that is better amenable to analyses than GPL implementation. This comes at the cost of expressiveness: in contrast to UML Statecharts [OMG10], AUTOMATA cannot represent hierarchies. While this is not necessary as hierarchical composition is possible on component level, there also are various component kinds, for which better behavior modeling formalisms than state-based automaton behavior may exist. Describing the behavior of components interfacing software libraries, APIs, or frameworks to interact with hardware, operating system functionalities, or to perform complex calculations currently seems be more efficient with GPL implementations. For components with behavior specific to certain contexts, other modeling languages are better suitable. As MontiArcAutomaton enables developers to describe the behavior of components either with GPLs or the most-appropriate modeling languages, selection of the best formalisms is up to the application modelers.

Chapter 6

Reusable Architectures through Bindings and Libraries

*Architecture should speak of its time and place,
but yearn for timelessness.*

Frank Gehry

Enabling reuse is crucial to engineering. Without the reuse enabled by standardization of parts and components, many disciplines, such as civil engineering, electrical engineering, or mechanical engineering would have flourished less. However, according to a recent survey on architecture language usage [MLM⁺13], “reuse is not commonly performed during architecting activities” and the most common form of structured reuse is reusing component definitions or realizations (29% of the respondents).

With the MontiArcAutomaton ADL, we aim for two dimensions of reuse. First, it enables reusing components in multiple architectures. Second, it also facilitates reusing complete architectures via interface components. For the latter, we developed concepts of component replacement that are realized as a modeling language and a M2M transformation. While both dimensions facilitate component maintenance and evolution efforts, the former is a common feature of C&C ADLs. This chapter therefore introduces the modeling language and transformation to develop and reuse platform-independent software architectures models as illustrated in Section 3.3.2.

The MontiArcAutomaton modeling language family introduced in Chapter 4 enables a pervasive model-driven engineering of C&C software architectures with exchangeable behavior languages and platform-independent as well as platform-specific components. Platform-independent components either are composed or contain behavior models and, hence, can be translated to arbitrary target platforms. However, a typical application requires components that interact with GPL artifacts to wrap operating system functionalities, libraries, or APIs as well. Interfacing such artifacts is often not supported by abstract and concise modeling languages [MHS05]. Thus, component application programmers and implementation library providers have to develop component behavior implementations in form of GPL artifacts. This gives rise to notational noise [Wil01] and accidental complexities [FR07]. Furthermore, these behavior implementations tie the components using the contributed GPL behavior implementations, and with it the complete software architecture, to the employed GPLs. While this is less important if

the logical software architecture is intended to be used for a single target platform, it hampers reuse with different target platforms (Req. *MRQ-8*) severely. Reusing an architecture containing platform-specific components tied to a specific GPL with other target platforms not supporting that GPL could be enabled by leaving out these components, hence creating “holes” in the architecture, filled by platform-specific components prior to code generation. Such holes represent extension points of the platform-independent software architecture, but introduce structural incompleteness to the software architecture. Such incompleteness renders validating an architecture’s integrity prior to code generation impossible, therefore MontiArcAutomaton aims to avoid this (Req. *MRQ-9*). Instead, a mechanism to represent such extensions points has to integrate into the C&C nature of MontiArcAutomaton. This either requires introducing new language elements to the MontiArcAutomaton ADL or to provide a solution apart from the ADL. In both cases, the implementations of the extension points need to be specified prior to code generation as the generated code relies on using the platform-specific implementations to compute component behavior. This information cannot be part of the software architecture model without tying it to specific GPLs or platforms, thus it needs to be stored in separate artifacts.

Our concept to architecture reuse relies on modeling the platform-independent architecture with interface components where modeling behavior is unfeasible. Application configuration models reference such abstract architectures and describe how the interface components are replaced by inheriting components for a certain platform. MontiArcAutomaton processes these application configuration models prior to well-formedness checking and performs the replacement model transformations. As the platform-independent architecture before applying replacement transformations is a valid MontiArcAutomaton ADL model, it can be reused without modifications as required by Req. *MRQ-8*.

This approach enables application modelers to engineer logical, platform-independent architectures containing interface components and to transform these to platform-specific architectures by binding the interface components to platform-specific, parametrized component types. Therefore, the architecture modeler develops the architecture to contain interface components from *interface libraries* developed by interface library providers. Afterwards, the implementation library provider develops proper *implementation libraries*. These realize interface libraries with platform-specific components extending their interface components. With *application configuration* models, the application modeler configures how the interface subcomponents should be replaced. In the end, MontiArcAutomaton processes the application configuration models, the referenced platform-independent software architecture, and library components to transform the platform-independent architecture into different platform-specific architectures according to the bindings. From these models, MontiArcAutomaton can generate executable systems. Figure 6.1 depicts the prime binding constituents and their relations. The platform-independent architecture uses platform-independent and interface library components only. The transformed platform-specific architecture uses platform-specific implementation library components only. The application configuration describes how

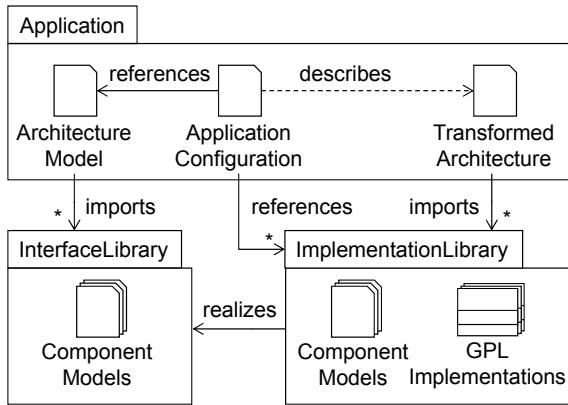


Figure 6.1: The most important constituents of bindings and their relations.

to derive the transformed, platform-specific architecture from the platform-independent architecture in terms of bindings.

This chapter introduces bindings, interface components, platform-specific components, and outlines the binding transformation in Section 6.1. Afterwards, Section 6.2 introduces interface libraries, implementation libraries, and three example libraries. Section 6.3 describes how MontiArcAutomaton applies bindings. Finally, Section 6.4 discusses observations and related approaches.

6.1 Modeling Platform-Independent Architectures

Behavior of MontiArc components is either specified by a corresponding GPL behavior artifact in the target platform GPL or emerges from their subcomponents' interaction. MontiArcAutomaton offers a third way to specify component behavior with embedded component behavior models. Furthermore, it introduces means to explicitly reference behavior implementation artifacts in primitive component types without behavior model (cf. Section 4.1). In the following, the umbrella term *behavior description* covers all mechanisms of MontiArcAutomaton to denote component behavior of atomic components, i.e., component behavior models as well as explicit and implicit references to implementation artifacts (see Section 2.4).

Modeling platform-independent architectures without structural underspecification requires means to describe the existence of components with platform-specific behavior without tying the architecture to a specific platform. MontiArcAutomaton therefore introduces the notion of *interface components* (cf. Section 4.1.1), which act as architecture extension points. Such components follow the general concept of interfaces in object-oriented software engineering, i.e., they can be used at architecture design time to describe properties expected from realizations, but they cannot be used to create executable systems. Instead, they need to be realized by platform-specific components that extend the interface component to be replaced. We define interface components as:

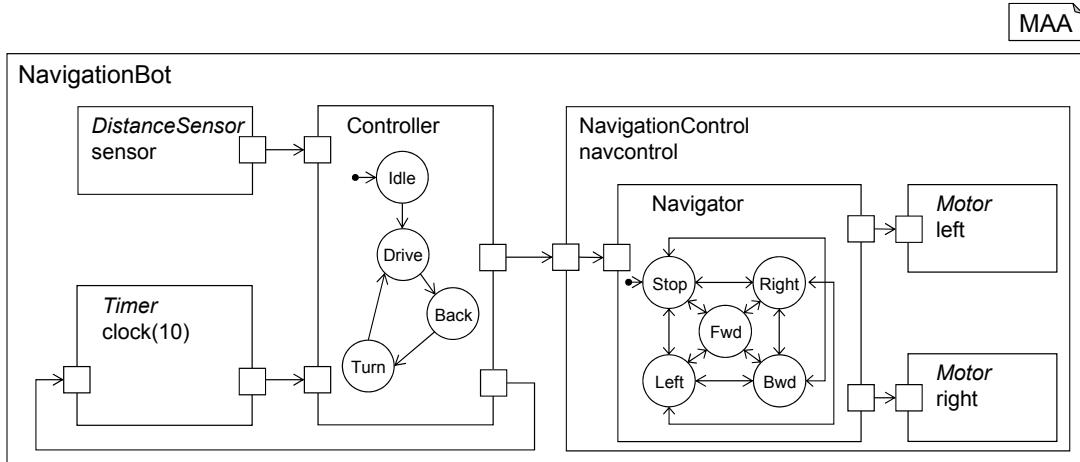


Figure 6.2: The NavigationBot is a variant of ExplorerBot (Figure 3.2) with an additional hierarchy level. Port names and types are omitted for clarity.

Definition 2 (Interface Component). *An interface component is a component without subcomponents, behavior model, or component behavior implementation reference.*

Thus, the interface components of MontiArcAutomaton resemble component interfaces [KGO⁺01, BCL⁺06, FG12] or component hulls [SSL11]. As they do not prescribe any form of component behavior, they can be used for modeling and analysis purposes, but not for code generation. Ultimately, an interface component neither has ties to a GPL, nor to a target platform, but defines the model elements MontiArcAutomaton requires to validate the containing architecture. This allows software architectures that contain interface components mixed with platform-specific components. This is useful if the architecture is intended to work with a single GPL but multiple different components for similar purposes (for instance, to easily exchange one sensor component for another). In contrast, we define platform-specific components as:

Definition 3 (Platform-Specific Component). *Platform-specific components are atomic components without behavior models, which declare conformance to a run-time environment.*

The NavigationBot depicted in Figure 6.2 is a variant of the software architecture ExplorerBot (cf. Figure 3.2) that relies on the interface components distance sensor and two motors to explore an unknown area. Contrary to ExplorerBot, the robot's Controller is connected to a composed component NavigationControl instead of being directly connected to two Motor instances. The Controller sends high-level movement commands to the navcontrol instance that uses an instance of component type Navigator to translate these into commands for two motors. The interface component clock provides an incoming port of CD type TimerCMD (cf. Figure 2.8), an outgoing port of CD type Boolean, and a configuration parameter of CD type Integer (here instantiated with value 10).

```

1 interface component Timer[Integer offset] {
2   port
3     in TimerCmd cmd,
4     out Boolean signal;
5 }
```

MAA

Listing 6.1: The interface component `Timer` with configuration parameters and ports of platform-independent CD types.

The textual representation of interface components (cf. Listing 6.1) begins with the keyword `interface` followed by a component definition and a body that may contain ports only. Obviously, interface components may not contain component behavior descriptions or references to run-time environments. The MontiArcAutomaton ADL context conditions (cf. Section 4.1.4) take care to reject such malformed models. Interface components are subject to inheritance and both, interface components and platform-specific components, may inherit from interface components. It is, however, not allowed to define interface components that inherit from platform-specific components as inheritance could propagate platform-specific properties of the super type. Such models are rejected by MontiArcAutomaton ADL context conditions as well.

With MontiArcAutomaton transforming software architectures to define only a single subcomponent per subcomponent declaration (cf. Section 4.1.5), a binding is a mapping from an interface subcomponent to a platform-specific subcomponent. It consists of a source, identifying a subcomponent in the architecture's subcomponent hierarchy to be replaced, and of a target, describing how it should be replaced. The latter consists of a platform-specific components and corresponding configuration arguments. Hence, together with the instance name of the subcomponent to be replaced, a binding's right-hand side describes a new subcomponent.

Definition 4 (Binding). *A binding for a MontiArcAutomaton software architecture \mathcal{A} is a tuple $(s, t(a_0, \dots, a_n))$, denoted as $s \rightarrow t(a_0, \dots, a_n)$, where:*

- $s = (s_0 \dots s_m)$ is a sequence of names that identifies an interface component T_s with configuration parameters p_0, \dots, p_k , $k \leq n$ in the subcomponent hierarchy of \mathcal{A} ,
- t is the name of a platform-specific component T with configuration parameters $p_0, \dots, p_k, p_{k+1} \dots, p_n$ for $k \leq n$, such that T extends T_s , and
- a_0, \dots, a_n is a list of configuration expressions, such that for each expression a_i its type matches the type of p_i .

with $i, n, m, k \in \mathbb{N}$.

The *subcomponent path* $s = (s_0 \dots s_m)$ unambiguously identifies a subcomponent s by following the sequence of subcomponent names through the subcomponent hierarchy of

A. The context conditions MontiArcAutomaton inherits from MontiArc (cf. [HRR12]) prohibit multiple subcomponents of the same name in a composed component, i.e., on each level of the subcomponent hierarchy. In consequence each prefix of s is unambiguous and, hence, s is. Given the component type NavigationBot depicted in Figure 6.2, valid subcomponent paths are (sensor), (clock), (controller), (sensor), (navcontrol), (nav-control,navigator), (navcontrol,left), (navcontrol,right). Furthermore, requiring that the platform-specific component T inherits from the interface component T_s allows adding, platform-specific configuration parameters to the platform-specific software architecture without adding it to the platform-independent architecture and hence fulfills Req. *MRQ-8.1*. As T can be composed, its behavior can be composed from arbitrary subcomponents as required by Req. *MRQ-8.3*. We write $s_0.s_1 \dots s_m$ to denote the subcomponent path (s_0, \dots, s_m) and $s \rightarrow T(a_0, \dots, a_n)$ to describe the binding (s, t, a_0, \dots, a_n) . Furthermore, we omit empty lists of configuration arguments where appropriate, such that $s \rightarrow t$ describes the same binding as $s \rightarrow t()$.

We employ bindings to define transformations of platform-independent software architectures into platform-specific software architectures. To transform the platform-independent software architecture of NavigationBot (see Figure 6.2) into a platform-specific software architecture, an implementation library provider has to develop platform-specific components that realize the interface components DistanceSensor, Timer, and Motor relative to the intended target platform. The application modeler then binds the subcomponents of these types properly. Given component types NXTUltrasonic, JavaTimer, and NXTMotor, such that NXTUltrasonic extends DistanceSensor, JavaTimer extends Timer, and NXTMotor extends Motor, the software engineer could specify bindings $\text{sensor} \rightarrow \text{NXTUltrasonic(S1)}$, $\text{left} \rightarrow \text{NXTMotor(A)}$, and $\text{right} \rightarrow \text{NXTMotor(B)}$ binding the NavigationBot architecture to a platform supporting the NXT components. The platform-specific software architecture resulting from applying these bindings to NavigationBot is depicted in Figure 6.3.

The bindings have also augmented the subcomponents `sensor`, `left`, and `right` with platform-specific arguments (`S1`, `A`, and `B`). These arguments identify hardware ports the behavior implementations of the respective component types interface with. As these are platform-specific, adding this information to the software architecture model would result in tying it to this very specific platform.

For more complex software architectures, bindings may lead to consistency issues if the subcomponent s of a component type T , which is instantiated multiple times as t_0 , t_1 , is bound to different component types, e.g., $t_0.s \rightarrow X$, $t_1.s \rightarrow Y$ with $X \neq Y$. In this case, application of these bindings would result in

- a subcomponent t_0 of type T with a subcomponent s of type X , and
- a subcomponent t_1 of a type T with a subcomponent s of type Y .

As MontiArc requires that the subcomponent t_0 of component type T has the same component type for all instances of T , the type T would be inconsistent. Furthermore, code generators that produce artifacts based on component types (Req. *TRQ-8*) will fail to produce proper artifacts.

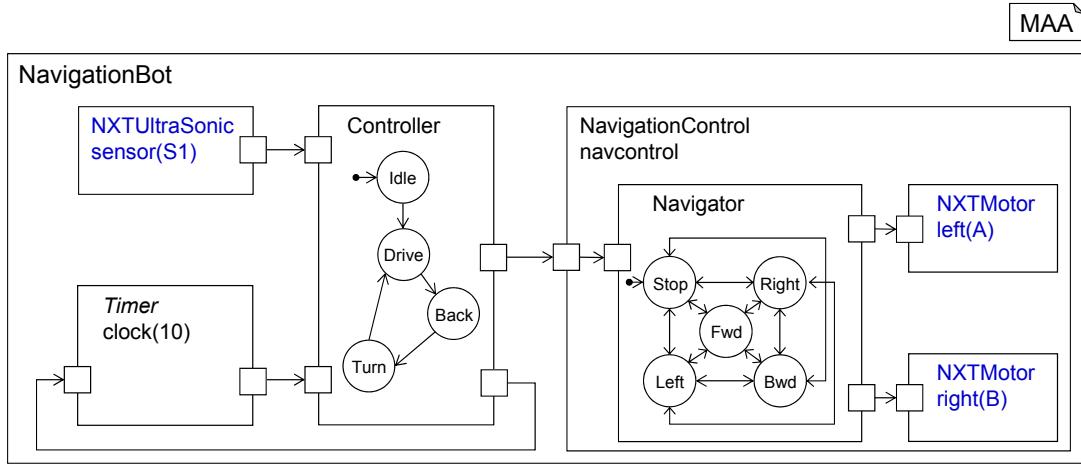


Figure 6.3: A platform-specific variant of the software architecture NavigationBot, where the interface subcomponents `sensor`, `navcontrol.left` and `navcontrol.right` have been bound to platform-specific components.

Figure 6.4 illustrates this with the component type `Motors` that contains two subcomponents `left` and `right` of component type `ValidatedMotor`. The latter receive messages for the motor and validate these before they are passed to the actual motor. To this effect, it consists of two subcomponents: `val` of component type `Validator` and `motor` of component type `Motor`. Given bindings `left.val → NXTVal`, `right.val → NXTVal`, `left.motor → NXTMotor(A)`, and `right.motor → ROSMotor(A)`, the resulting component `MotorsT` would be inconsistent in the type of its subcomponent `motor`. This inconsistency arises from a *clash* between the two bindings `left.motor → NXTMotor` and `right.motor → ROSMotor`, which bind the subcomponent `motor` instantiated by the component type `ValidatedMotorT` differently. In general, there is a clash between two bindings for the same software architecture if they bind the same subcomponent differently.

Definition 5 (Binding Clash). *There is a clash between two bindings $a_0 \dots a_n \rightarrow T_a(\dots)$ and $b_0 \dots b_m \rightarrow T_b(\dots)$ if they bind a subcomponent of a common parent component type to different component types, i.e., subcomponents a_{n-1} and b_{m-1} have the same type, a_n and b_m have the same name but $T_a \neq T_b$.*

Resolving clashes prior to applying bindings is crucial to the resulting software architecture's validity and the procedure presented in Section 6.3 takes care of that.

Overall, reusing a logical, platform-independent software on multiple target platforms requires the artifacts displayed in Figure 6.5. The MontiArcAutomaton language family provides a workflow to transform software architectures based on their application configuration's bindings. The application modeler develops the platform-independent architecture model and one application configuration model per target platform. The software architecture model uses interface components imported from platform-independent

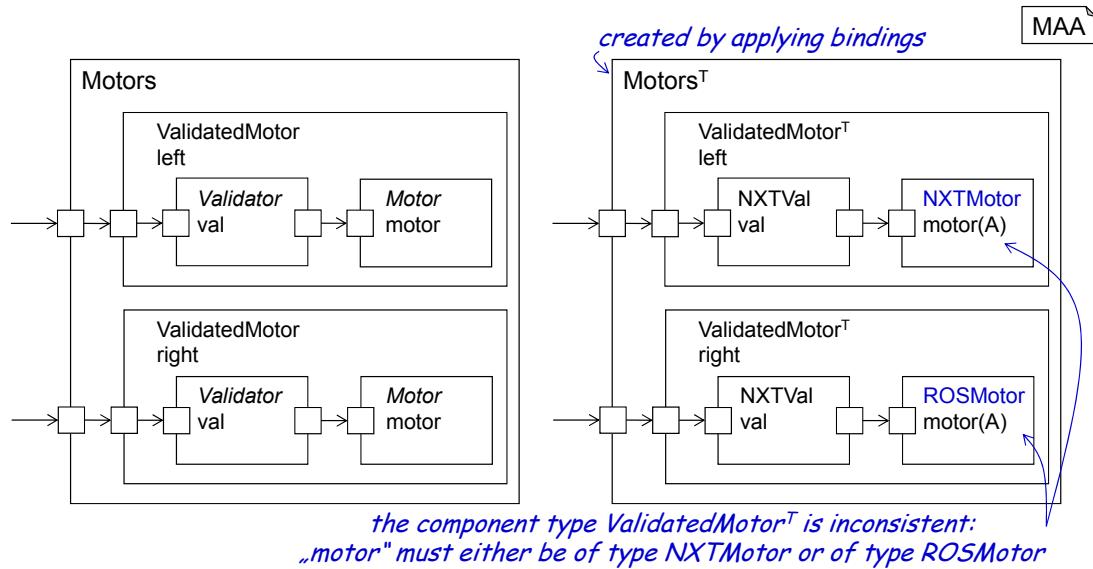


Figure 6.4: Example for a conflict between two bindings for the same software architecture: Subcomponent motor of component type ValidatedMotor must be of unambiguous kind.

interface libraries to describe the architecture's extension points. Each application configuration references the software architecture and contains bindings that describe how interface components of the software architecture should be replaced with components from the respective implementation libraries. MontiArcAutomaton invokes the binding transformation for each application configuration to transform the architecture into a platform-specific variant according to the bindings. This transformed architecture does not rely on interface components anymore and, thus, can be used for further analyses as well as for code generation. The application configuration modeling language to describe bindings is presented in Chapter 8. In the following, Section 6.2 presents interface and implementation libraries, before Section 6.3 describes the binding model transformation.

6.2 Interface Libraries and Implementation Libraries

MontiArcAutomaton employs a distinction between interface libraries and implementation libraries to enforce reuse of platform-independent software architectures with multiple platforms. Therefore, interface libraries may contain platform-independent models only and implementation libraries realize a specific interface library with components that extend the interface libraries' components for a specific target platform. This enables modeling software architectures without ties to specific platforms or GPLs and clarifies which components implementation library providers need to develop to realize a given platform-independent architecture on a specific platform. Hence, we define both kinds of libraries as follows:

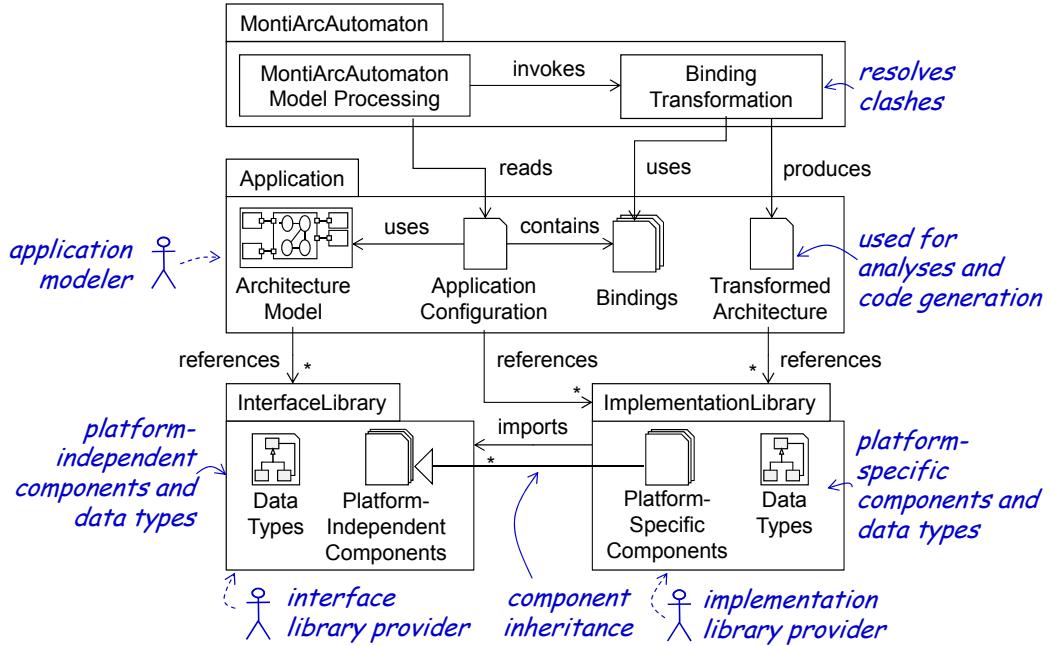


Figure 6.5: The MontiArcAutomaton ADL transforms a single platform-independent architecture model into one platform-specific architecture model per application configuration model.

Definition 6 (Interface Library). *An interface library is a named set of interface components with referenced data type models.*

Definition 7 (Implementation Library). *An implementation library is a named set of component models with referenced data type models and GPL behavior implementations. All components inherit from interface components of a single interface library.*

As platform-independent architectures may contain composed components, components with a behavior model, and interface components only, interface libraries are restricted to the same elements. This guarantees platform-independent interface libraries and that the importing software architectures remain platform-independent as well. The interface components of interface libraries need to be replaced by platform-specific components of implementation libraries prior to further processing. To ensure interface compatibility, implementation libraries must provide an inheriting component type for each interface component of the architecture. This ensures that it must provide at least the ports and configuration parameters of the interface component. Additionally, implementation libraries must contain the data type models required by their component models, corresponding platform-specific GPL behavior implementations and their required GPL data type artifacts. Furthermore, they may contain platform-independent, non-interface components as well.

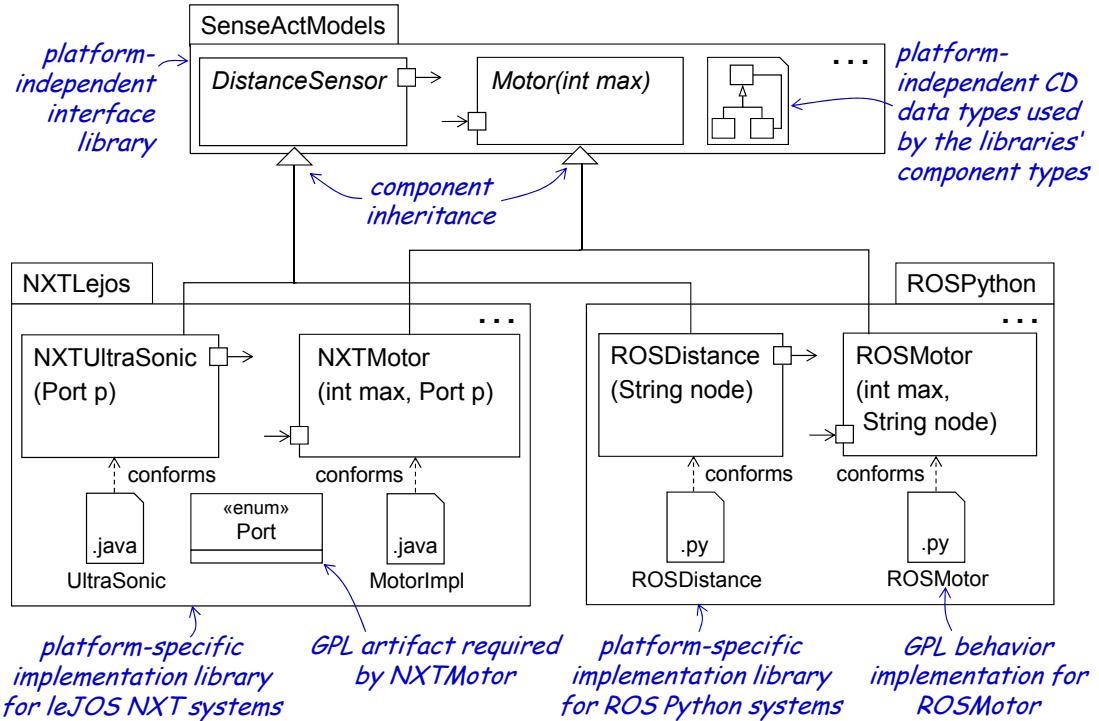


Figure 6.6: An excerpt of the interface library `SenseActModels` and the corresponding implementation libraries `NXTLejos` for NXT robots and `ROSPython` for robots employing the robot operating system (ROS [QGC⁺09]) in its Python implementation.

Figure 6.6 describes the relations between interface components and platform-specific components in the context of the containing libraries: The `SenseActModels` interface library comprises interface components to describe the interfaces of actuators and sensors as well as platform-independent class diagram data types to describe the types of the components' parameters, ports, and variables. The `NXTLejos` implementation library contains their platform-specific realizations in form of the component types `NXTUltraSonic` and `NXTMotor`. These extend the `DistanceSensor` and `Motor` interface components, respectively. Both platform-specific realization types introduce new parameters that require the bound subcomponents to provide proper arguments for. The library also contains the handcrafted Java behavior implementations for both component types and the platform-specific data types required by the components and their Java behavior implementations. The `ROSPython` implementation library provides platform-specific component realizations for platforms supporting the robot operating system (ROS [QGC⁺09]) and their handcrafted Python behavior implementations. To this effect, it contains the two component types `ROSDistance` and `ROSMotor`, which also inherit from the platform-independent components, but introduce different configuration parameters than the NXT-compatible component types.

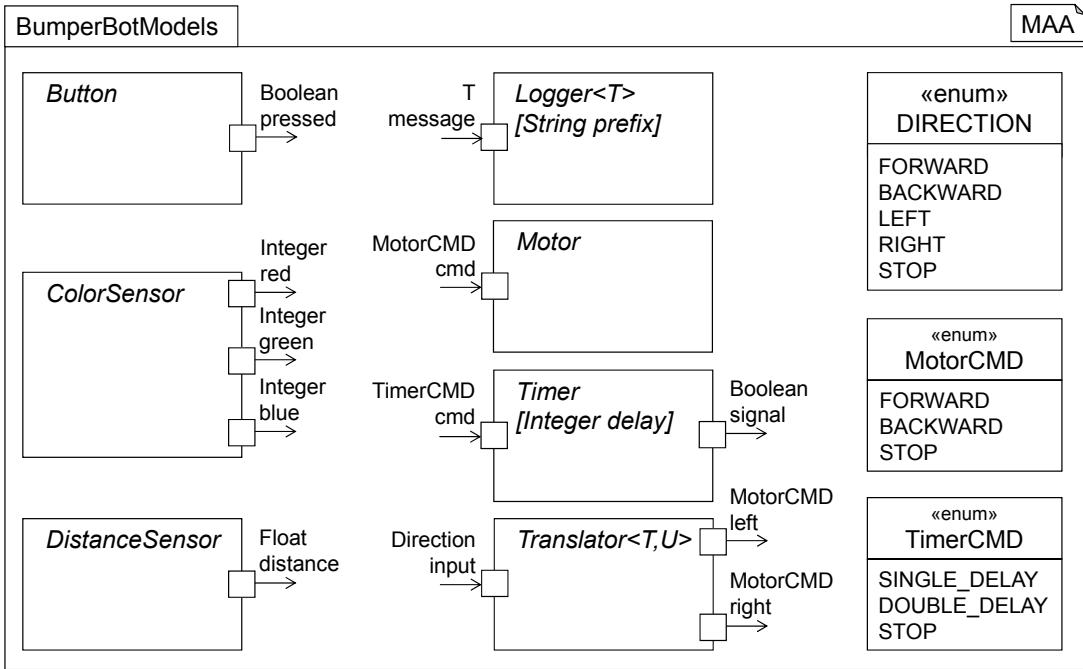


Figure 6.7: The component types and data types of interface library BumperBotModels. Although interface libraries may contain composed components as well as components with behavior models, this library requires interface components only.

The components of implementation libraries are referenced by the bindings of application configuration models, which imported the respective implementation library. Hence, bindings must provide proper arguments for the new, platform-specific, parameters. Hence, platform-specific parameters are part of the binding, but not of the platform-independent software architecture. Section 6.3 describes how to apply bindings.

The remainder of this section describes three example libraries: the interface library BumperBotModels, an implementation library for the LeJOS Java platform, and an implementation library for the ROS Python platform. For the evaluations presented in Section 9.1, further libraries were developed.

6.2.1 BumperBot Interface Library

The BumperBotModels interface library provides component types and UML/P class diagram data types to describe sensors and actuators of the exploration robots displayed throughout this thesis (as for instance in Figure 6.2), but independent of the GPL actually used to control these robots. The interface library BumperBotModels comprises the three sensor component types Button, ColorSensor, DistanceSensor, the actuator component type Motor, and the three component types Logger, Timer and Translator that wrap operating system functionalities. Figure 6.7 illustrates these

component types with their ports, configuration parameters, generic parameters, and the related data types `Direction`, `MotorCMD`, and `TimerCMD`.

Component Type `Button`

The component type `Button` wraps all forms of binary sensors that emit whether they are pressed currently. Whether these are complex touch screen devices or simple buttons is irrelevant to the component model that emits whether the measured data corresponds to being touched via port `pressed` of type `Boolean`.

Component Type `ColorSensor`

The component `ColorSensor` serves as an extension point for all forms of color sensors providing RGB data of a single point. To this effect, it omits incoming ports and emits messages via the three integer ports `red`, `green`, and `blue`. The data type `Integer` is a class diagram type and can be translated into GPLs as required.

Component Type `DistanceSensor`

The component type `DistanceSensor` provides the distance to the single next obstacle in the direction the sensor is faced. This distance is emitted via the port `distance` of class diagram type `Float`.

Component Type `Logger`

Components of type `Logger` persists incoming messages received via the incoming port `message`. For greater flexibility, it yields the generic type parameter `T` to define the type of its incoming port `message`. The component type also features the configuration parameter `prefix` that is supposed to be prefixed before each incoming message and has a default value of `" "`. Where the `Logger` logs the received messages to is subject to the component implementation and invisible to the component model.

Component Type `Motor`

Components of type `Motor` serve to control motors with a single degree of freedom. Such motors can rotate in two directions or stop. Thus the data type `MotorCMD` of incoming port `cmd` is restricted accordingly.

Component Type `Timer`

The component type `Timer` implements countdown functionalities. To this end, it features an incoming port `cmd` to platform-independent class diagram type `TimerCMD` and an outgoing port `signal` of platform-independent class diagram type `Boolean`. Furthermore, it features the configuration parameter `delay` of type `Integer` that describes how long the standard length of a countdown is. Depending on the received command,

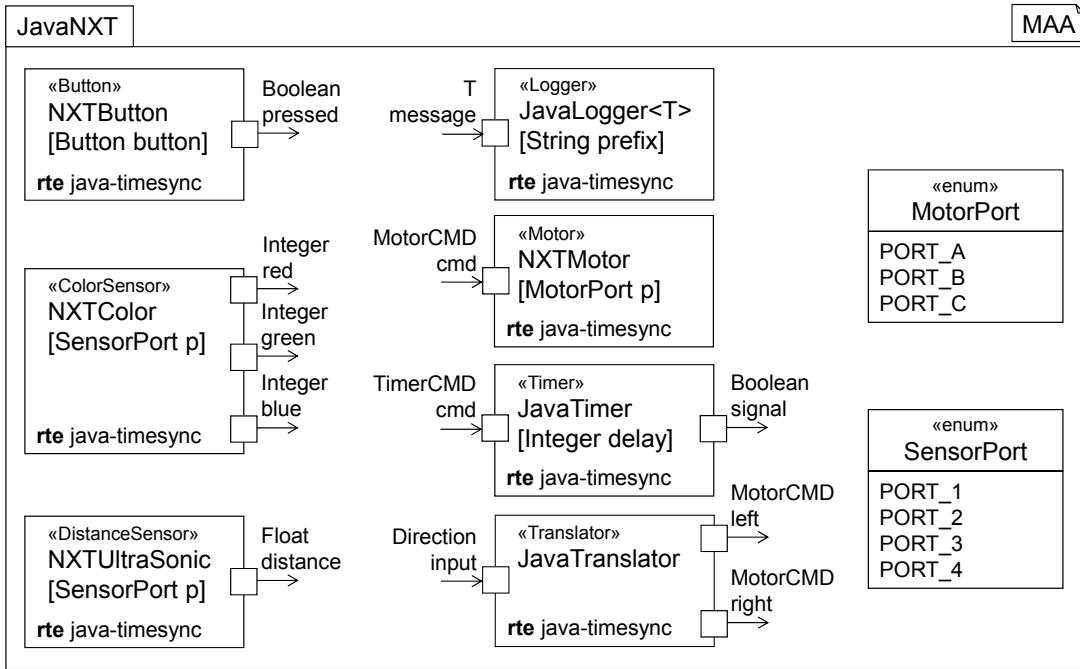


Figure 6.8: The component types and data types of implementation library JavaNXT.
The corresponding GPL behavior implementation artifacts are omitted for clarity.

the timer either starts a countdown of standard length, of double length, or stops the active countdown. This restriction is arbitrary and was introduced to reduce interface complexity [RRW13b].

Component Type Translator

The component type Translator translates a single input of type Direction received port via the port input into two outputs of type MotorCMD. This component type is used, for instance, with the MontiArcBumperBot displayed in Figure 2.7 on page 24.

6.2.2 JavaNXT Implementation Library

The JavaNXT implementation library realizes the component types of the interface library BumperBotModels for platforms that support the Java LeJOS [LeJ] operating system. To this effect, it contains one component type for each interface component of the interface library BumperBotModels as well as platform-specific data types. Figure 6.8 displays the component types and data types JavaNXT comprises. Please note, that due to comprehensibility, the interface components of BumperBotModels the respective component types of JavaNXT inherit from are omitted and illustrated via stereotypes instead.

Furthermore, JavaNXT contains a GPL behavior implementation for each contained component type. As each component type of JavaNXT is compatible to the run-time environment `java-timesync`, each component types' GPL behavior implementation artifact implements a certain interface of that run-time environment. This enables interaction of generated component artifacts. The following paragraphs explain the individual components' types of the JavaNXT implementation library and point to the LeJOS Java classes they interface.¹

Component Type NXTButton

Components of type `NXTButton` inherit from the component type `Button` of the implementation library `BumperBotModels`. Therefore, they provide the same interface, consisting of outgoing port `pressed` of data type `Boolean` and introduce the configuration parameter `button` of platform-specific data type `lejos.nxt.Button` to define the wrapped hardware button. The behavior implementation of `NXTButton` interfaces the LeJOS Java class `lejos.nxt.Button` to detect the configured button's state.

Component Type NXTColor

The component type `NXTColor` provides color measurements. To this effect, it wraps the LeJOS API class `lejos.nxt.ColorSensor`. Therefore, it extends the component type `ColorSensor` of the `BumperBotModels` library and introduces the configuration parameter `p` of type `SensorPort` that describes which physical port the behavior implementation of `NXTColor` is connected to. It emits color measurements via the ports `red`, `green`, and `blue` inherited from `ColorSensor`.

Component Type NXTUltraSonic

Components of type `NXTUltraSonic` inherit from `DistanceSensor` and interface ultrasonic-based distance sensors via the LeJOS class `lejos.nxt.UltrasonicSensor`. After measuring the distance to a single obstacle, it emits this distance via the port `distance` of type `Float`. Therefore, `NXTUltraSonic` introduces the configuration parameter `p` of type `SensorPort`.

Component Type JavaLogger

The `JavaLogger` component type extends the component type `Logger` of the interface library `BumperBotModels` to provide logging capabilities for Java-based systems such as LeJOS. It retains the inherited generic type parameter `T` and the configuration parameter `prefix`. Received messages are printed to the system's standard output.

¹The LeJOS API is available from: www.lejos.org/p_technologies/nxt/nxj/api/.

Component Type **NXTMotor**

The component type **NXTMotor** inherits from the interface component **Motor** of interface library **BumperBotModels** and introduces a new configuration parameter **p** of platform-specific type **MotorPort**. As **NXTMotor** inherits the complete interface of **Motor**, it also may receive messages via the port **cmd** of platform-independent data type **MotorCMD**. Its behavior implementation interfaces the Java class `lejos.nxt.Motor` connected to the NXT brick's port passed to **NXTMotor** via component instantiation.

Component Type **JavaTimer**

The component type **JavaTimer** extends the component type **Timer** of interface library **BumperBotModels** to realize countdown functionality with help of LeJOS class `lejos.util.Stopwatch`. To this effect, it sets the `Stopwatch` to a duration of either a single delay (as received via its configuration parameter) or a double delay. Each tick, it checks whether the delay received as `TimerCMD` via incoming port `cmd` has elapsed. In this case, it emits `true` via its outgoing port `signal`, otherwise `false`.

Component Type **JavaTranslator**

Component type **JavaTranslator** translates each incoming message of data type **Direction** into two outgoing messages of data type **MotorCMD**. To this effect, it extends **Translator** of the **BumperBotModels** interface library and provides a proper Java implementation for translation.

6.2.3 Python ROS Implementation Library

The implementation library **PythonROS** provides component types extending the component types of interface library **BumperBotModels** to realize their functionality on platforms using the robot operating system **ROS** [QGC⁺09] in its Python implementation. ROS describes software architectures as networks of interacting nodes that communicate via topics (typed, ordered message buffers that exist at system run-time only). Consequently, each component type of **PythonROS** creates a node for a specific task and, hence, yields a configuration parameter `node` of type `String` to define the created node's name. Similar to **JavaNXT**, the components interacting with hardware interface Lego NXT hardware via ROS. Therefore, these components use the `nxt_ros` package.² These components also yield additional configuration parameters to interface the connected hardware (either `frameID` or `motorID`). Figure 6.9 displays the component types of **PythonROS**. All are compatible to the run-time environment `python-timesync`. Details on ROS are available published [QGC⁺09] and online³.

²Available via http://wiki.ros.org/nxt_ros

³ROS website: <http://wiki.ros.org/>

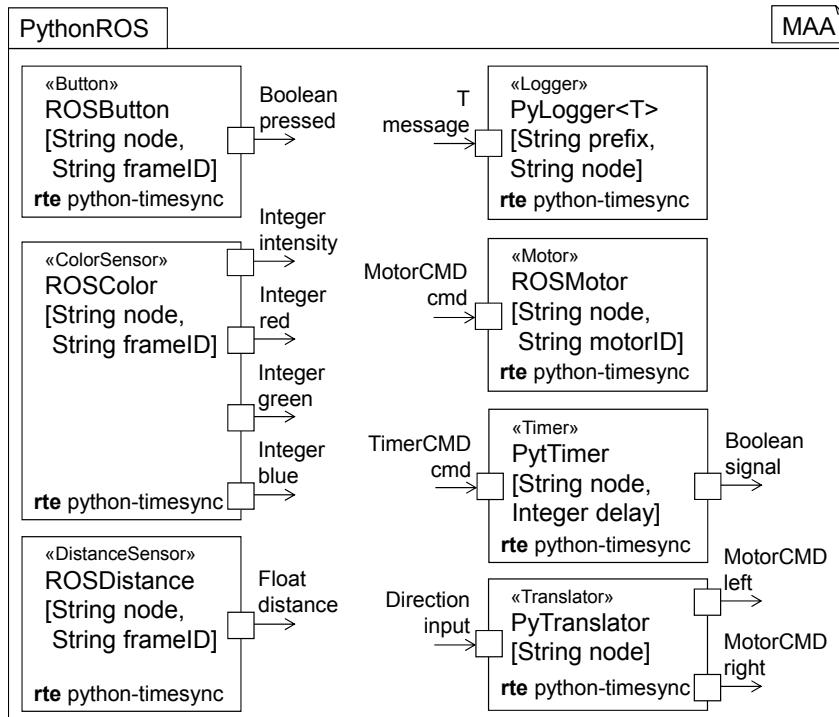


Figure 6.9: The component types and data types of implementation library PythonROS. The corresponding GPL behavior implementation artifacts and ROS message types are omitted for clarity.

Component Type `ROSButton`

The component type `ROSButton` wraps a touch sensor. Therefore, it extends the interface component `Button` provided by the interface library `BumperBotModels` and reads ROS messages of type `nxt_msgs/Contact` from the topic `/touch_sensor`. These messages are translated into a Boolean value send via the inherited outgoing port `pressed`. To configure the ROS node created by its component behavior implementation properly `ROSButton` introduces the configuration parameter `node`. Furthermore, it features the configuration parameter `frameID` to interface the correct touch sensor.

Component Type `ROSColor`

`ROSColor` realizes the functionalities of component `ColorSensor` of interface library `BumperBotModels`. To this effect, it extends `ColorSensor` and introduces two new configuration arguments as well as the new outgoing port `intensity`. The configuration arguments `node` and `frameID` configure the node created by the component's behavior implementation. This node reads ROS messages of type `nxt_msgs/Color` from topic `/color_sensor` and translates these into RGB values of type `Integer`. The new outgoing port `intensity` emits the measured intensity of the current RGB

color reading. As bindings prohibit rewiring, this port cannot be connected through bindings but can be used by platform-specific architectures only.

Component Type ROSDistance

Components of type ROSDistance extend `DistanceSensor` of the interface library `BumperBotModels` and emit distance measurements via the inherited outgoing port `distance` of platform-independent type `Float`. The component type reads sensor data messages of type `nxt_msgs/Range` from the ROS topic `/ultrasonic_sensor` and translates these accordingly. To this effect, it introduces the two configuration parameters `node` and `frameID` of type `String` to configure the ROS node their behavior implementation creates properly.

Component Type PyLogger

The component type PyLogger extends the component type `Logger` of interface library `BumperBotModels` to realize logging functionalities in Python-based ROS systems. Therefore, it introduces the configuration parameters `node` and `frameID`. For greater flexibility, it retains the generic type parameter `T` inherited from `Logger` as well as its configuration parameter `prefix`. It prefixes each received message with the value of `prefix` and prints it to the system's standard output.

Component Type ROSMotor

The component type ROSMotor extends the `Motor` of the `BumperBotModels` interface library. It receives messages of platform-independent data type `MotorCMD` and translates these into ROS messages of message type `nxt_msgs/JointCommand` passed to the topic `/joint_command`. The motor driver of `nxt_ros` reads messages from that topic and controls the motor identified via configuration parameter `frameID` through a node of name `name` accordingly.

Component Type PyTimer

PyTimer components realize countdown functionality for Python-based ROS systems. Hence, they introduce the configuration parameter `node` to configure their ROS node. The configuration parameter `delay` describes the duration of a single countdown. On receiving a `TimerCMD` message, the component either starts a countdown of single length, of double length, or stops an active countdown.

Component Type PyTranslator

Components of type PyTranslator realize the functionality of the component type `Translator` of interface library `BumperBotModels` for Python-based ROS systems. As such it also requires to create a node of the name passed via configuration parameter `node`. The component type does inherit from `Translator`, but not extend its interface.

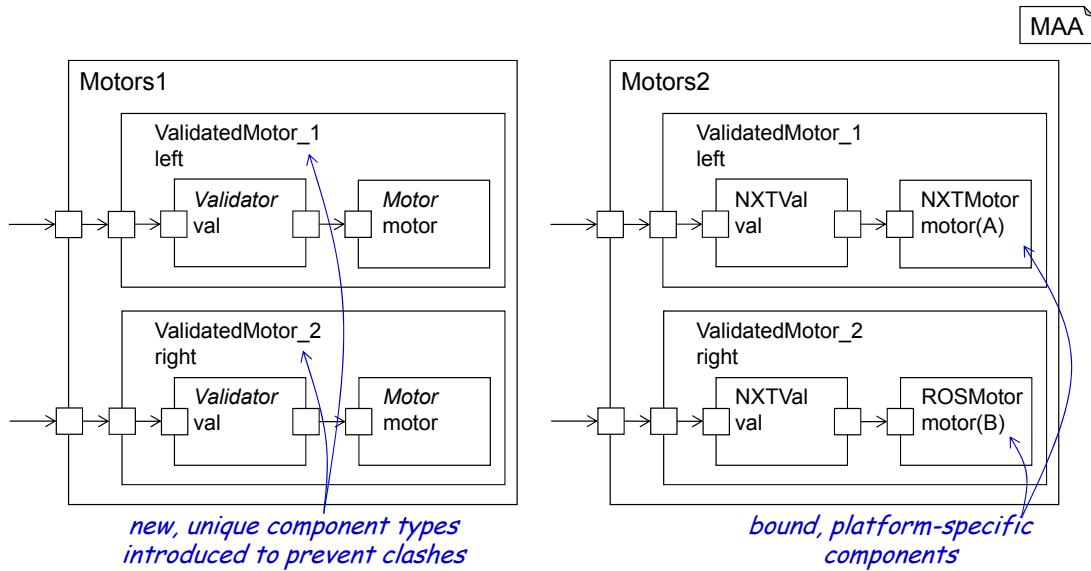


Figure 6.10: The clash between two bindings for subcomponent `motor` of type `ValidatedMotor` is resolved by introducing new component types.

6.3 Deriving Platform-Specific Architectures

The application of bindings to a platform-independent software architecture may raise clashes and, hence, produce invalid platform-specific software architectures. Nonetheless, MontiArcAutomaton allows binding interface subcomponents of the same component type differently, if these bindings refer to different instances of that type. Such – intended – clashes (cf. Figure 6.4) must be resolved by MontiArcAutomaton prior to code generation. The binding model transformation presented in this section takes care of resolving clashes while applying the bindings. Therefore, it replaces the types of all subcomponents with copies of new, unique names. For the component type `Motors` depicted in Figure 6.4, this results in the component type `Motors1` (as illustrated in Figure 6.10) prior to applying the bindings `left.motor` → `NXTMotor` and `right.motor` → `ROSMotor` and in `Motors2` afterwards. The resulting component `Motors2` is a valid MontiArcAutomaton component as it does not feature a component type with subcomponents of the same name but different types anymore. To achieve this, the binding transformation conducts a breadth-first search through the subcomponent hierarchy of the architecture's component type. During this search, it replaces the component types of interface subcomponents and applies the arguments according to the bindings. It also replaces the types of other subcomponents with copies of their original types with new and unique names to prevent clashes. The corresponding procedure is depicted in Listing 6.2.

The procedure `bind` expects an architecture root and a set of bindings. The latter must be free of conflicts and otherwise invalid bindings. Conflicts occur if two bindings

```

1 bind(ComponentType root, Bindings b)                                PC
2   Stack stack = new Stack()
3   newRoot = uniqueCopy(root)
4   stack.put("", newRoot)
5   while not s.isEmpty()
6     (pre, comp) = stack.pop()
7     for each subcomponent sc = (name, type(args)) of comp
8       if (pre == "")
9         q = sc.name
10      else
11        q = pre + "." + sc.name
12      if b(q).exists()
13        sc.type = b(q).type
14        sc.args = apply(args, b(q).args)
15      else
16        sc.type = uniqueCopy(sc.type)
17        stack.put(q, sc.type)
18   newRoot = reduce(newRoot)
19   return newRoot

```

Listing 6.2: The bind procedure replaces the types of subcomponents with either platform-specific bound types or new, unambiguous types.

reference the same subcomponent and bind it differently. Checking a set of bindings for such is trivial due to the MontiArcAutomaton well-formedness rules. Invalid bindings refer to nonexistent or subcomponents of non-interface types and bind these subcomponents to component types not extending the subcomponent's component type, or fail to parametrize the component type properly. To calculate bindings, bind utilizes a stack of tuples of subcomponent path names and component types. Initially, bind puts the empty qualified name and a copy of the architecture's root component on the stack (ll. 2-4). The name of the copy's type is ensured to be unique by the function `uniqueCopy()` (l. 3), which produces a copy of the passed component type including all constituents but changes its name accordingly. Afterwards, bind iterates over the stack's tuples and inspects every subcomponent `sc` of the component type currently visited (ll. 5-17). Then, it constructs the subcomponent path `q` of the currently inspected subcomponent `sc` to check with the current prefix and the subcomponent's name (ll. 8-11). Finally, bind checks whether `q` is bound (l. 12), i.e., `b` contains a binding for `q`, and determines the new type of `sc` accordingly (ll. 12-17). If a binding for `sc` exists, the component type of `sc` is changed to the bound type and the bindings arguments are applied (ll. 13-14). Otherwise, the type of `sc` is changed to a unique copy of itself and it is put onto the stack for further inspection (ll. 16-17). All these changes take place in `newRoot`, which is processed by procedure `reduce()` (l. 18) prior to returning it for further processing (l. 19). The procedure `reduce()` iterates over the subcomponent hierarchy of `newRoot` and reverts type renaming for subtrees where no bindings were applied. The procedure `bind` terminates for leaves of the subcomponent hierarchy (i.e., atomic components)

and therefore terminates for architectures with finite number of subcomponents on each level and finite subcomponent hierarchy depth.

The procedure `bind` returns a valid MontiArcAutomaton software architecture that describes the platform-specific architecture completely (as required by Req. *MRQ-8.4.*) Hence, the architecture can be processed by existing tooling (such as code generators, Req. *MRQ-8.5*) without need for modifications. It prevents clashes but creates new component types (ll. 3,16) for each subcomponent. Consequently, the number of new component types is bound by the number of subcomponents in the subcomponent hierarchy. Whether this number influences the resulting number of artifacts in a generated system depends on the code generators employed and their translation from component types to artifacts.

6.4 Discussion and Related Approaches

Bindings require a white-box view on subcomponent hierarchies as the left-hand sides of bindings pierce through component hulls to reference subcomponents normally hidden behind the containing components' interfaces. As MontiArcAutomaton extends MontiArc, the composition of components is always visible to the application modelers without means to hide it. If the latter was supported, post-modeling analyses could access the architectures' root elements only.

For architectures employing untimed or timed messaging semantics, bindings also do not change the architecture's structural semantics, which is invariant to applying bindings. Hence, the application configuration depends on the architecture but not vice versa. The architecture can be developed and evolved independently of possible bindings. Changes to the architecture model impact the bindings only in case components are (re)moved - which can render the bindings invalid. Adding or decomposing components does not impact the bindings. Hence, enabling the architecture modeler's white-box view on the complete architecture enables important functionality and provides a maximal stability between both models.

Application configuration models may define a single binding per subcomponent. For large software architectures, where multiple subcomponents of the same component type should be replaced by the same platform-specific component this is might be inconvenient. Defining bindings between component types would facilitate such specification. The ramifications of such bindings can be calculated easily by translating these into sets of bindings of each type's subcomponents. Furthermore, bindings replace subcomponents unconditionally. Conditional bindings might be useful if applied in the context of the target systems platform or domain model. Future work could investigate which forms of conditions are appropriate and integrate these.

Currently, bindings can introduce component types with additional ports not part of the inherited interface component's interface. However, bindings cannot connect these ports. Such rewiring might be useful, but entails greater changes to the platform-independent architecture than intended with our notion of bindings. Also, our notion requires the interfaces of interface components to be broad enough to support arbitrary

inheriting platform-specific components. With our approach, the interfaces are broad enough by design, as the platform-independent software architecture defines what is required. We also bind interface components only. Although the procedure `bind` supports binding of arbitrary subcomponents, such bindings are not intended. Especially binding composed components may change the software architecture beyond recognition and deviate it far from its original intentions. The procedure `bind` iterates the passed architecture root two times: once to transform the software architecture without clashes and a second time to reduce the number of new component types. In a future implementation this could be reduced to a single iteration that performs binding impact analysis before applying bindings.

Platform-specific, atomic components are restricted to declare conformance to a single run-time environment. While we did not encounter scenarios where conformance to multiple run-time environments was crucial, it would facilitate reuse of platform-specific components and implementation libraries with different contexts. Implementation of this is easy and requires changes to the `rte` modeling element (Section 4.1.1) and related context conditions only.

Replacing subcomponents with bindings requires interface-compatibility. MontiArc-Automaton enforces this via requiring an inheritance relation between the replaced subcomponent's type and the replacing type. As this does not suffice in the light of generic component type parameters, it also requires that the actual port types - after resolving the generics - are compatible as well. This notion of compatibility allows bindings to introduce new, unconnected, ports into the software architecture. As bindings do not provide means to introduce new connectors, these ports cannot be connected. While it is possible to restrict bindings to platform-specific components that provide exactly the ports of the replaced subcomponent, this restricts reuse and relaxing this did not pose problems in practice. Ports left unconnected by bindings are treated to receive no data and hence, binding component types relying on these ports might lead to unexpected effects. As the inherited MontiArc context condition CV6 (presented in [HRR12], p. 36) produces warnings for unconnected ports, the application modeler will be informed about this potential issue.

Another means to mark architecture extension points besides interface components could be achieved declaring component types as configuration parameters of other component types. Hence, passing instantiated components and using these to replace subcomponents could realize bindings as well. However, as component instantiation is part of the architecture, specifying platform-specific components as arguments for such parameters in it would tie the architecture to the respective platform. Furthermore, this also would require introducing new language elements to describe how component instances passed as configuration arguments would replace subcomponents and, thus, introduce additional notational noise [Wil01]. Describing replacement in form of bindings outside of the software architecture liberates component modelers from dealing with such unless they actually want to employ bindings.

Currently, MontiArcAutomaton supports translating the complete architecture to a target platform only. It is yet impossible to translate different parts of an architecture

to different platforms. This can be useful, for instance, to deploy part of an architecture to a cloud-based Java infrastructure while deploying components for sensors and actuators to a mobile platform requiring C++. Its realization entails many issues regarding augmentation with communication components, cross-platform marshalling, and timing, which are not within the scope of this thesis.

A solution for architectural extension points is described in [RRW14b] where *abstract components* are introduced to C&C software architectures to describe architecture extension points. Such components are abstract in providing only a component interface (cf. [SSL11, FG12]) without behavior and refer to platform-independent types only. Thus, these components are independent of specific APIs, libraries, or communication protocols and their structural information can be transformed into code for arbitrary target platforms. Nonetheless, from a structural view, these components are complete and are available to model checking. Current MontiArcAutomaton uses interface components instead, which are more restricted than abstract components. Application configuration models (as introduced in Chapter 7) refer to the software architecture and describe how its abstract components are replaced prior to code generation. Consequently, the platform-independent architecture is agnostic of its reuse with different target platforms. The solution presented in [RRW14b] relies on exchanging the actual behavior implementation artifacts of subcomponents prior by replacement-aware code generators. Subcomponents of the software architecture that are of abstract component types are mapped to platform-specific, component behavior implementations. Respecting replacement information requires binding-aware code generators to exchange the component behavior implementation during generation. Such replacements enable software architecture developers to delay commitment to a specific platform and ultimately increase reuse of architecture models and component models. While useful, that approach lacks expressiveness: Components, aside from their interfaces, are black-boxes to the environment and only they are responsible for configuration of their behavior descriptions (stored in aforementioned behavior implementation artifacts). Thus, component behavior descriptions can receive their configuration information solely from the component they belong to. As components are configured by the software architecture, passing additional, for instance platform-specific, configuration arguments (such as information on the physical connection of hardware) requires adding this information to the software architecture model – which ties it to specific platform properties. We enhanced the previous approach with a more explicit description mechanism to exchange subcomponent types instead of their behavior implementations. Thus, the new approach impacts the architecture model only and requires no consideration by code generators. Consequently, current MontiArcAutomaton uses different notions of bindings, which emphasize on the distinction between interface components and platform-specific components, and libraries. This yields the following improvements over [RRW14b]:

- Applying bindings to architectures with interface components produces type-safe architectures instead of changing component implementation details.
- Binding to platform-specific components may introduce new platform-specific pa-

rameters on model level.

- Existing code generators can be reused.
- Model libraries are replaced by interface libraries.
- Code libraries and their library models are redundant.

Current approaches to MDE with C&C ADLs do either not consider multi-platform reuse [GBWK09, BGP⁺10, CKS11, SSL11, FG12], or demand modeling platform details explicitly [ADvSP05, DKS⁺12, HGS⁺13]. Ignoring multi-platform reuse requires copying the architecture models and changing the platform-specific components manually. As the copied architecture models need to be maintained and evolved in parallel, this introduces additional efforts. Modeling platform properties explicitly [HGS⁺13] introduces complex notions to describe the target platform and the binding of components to it, which raises efforts regarding definition, maintenance, and evolution of platform models. Additionally, deployment may consider optimized code generation for specific target platform elements, realization of connectors between physically distributed components, or mechanical and electrical properties of the target platforms. Bindings alter a logical software architecture on component level to enable subsequent code generation of target GPL artifacts. How these are deployed on target platform elements is not within the scope of this approach.

The notion of “abstract platforms” [ADvSP05] in MDA is closely related as well. Abstract platforms are means to separate the “technology independent” and “technology related” aspects of target platforms. An abstract platform “represents the platform support that is assumed by the application developer at some point” [ADvSP05], which corresponds to the set of interface components used by a MontiArcAutomaton software architecture. Similar to our approach, the platform-independent model depends solely on the abstract platform and the platform-specific model depends on the concrete platform. In the same vein, the authors introduce a different notion of bindings. These bindings describe mappings from abstract platforms to concrete platforms and rely on “model libraries” [ADvSP05], which resemble our interface libraries, as well. In contrast to our approach, they do not restrict the content of interface libraries, which does not ensure that these are actually platform-independent.

Chapter 7

Compositional Code Generation

On two occasions I have been asked [by members of Parliament]: 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage

Pervasive, multi-platform, model-driven engineering with MontiArcAutomaton requires translation of components with embedded behavior models of flexibly integrated behavior modeling languages into GPL implementations for different target platforms. Thus, the code generation infrastructure must be able to integrate new generation capabilities for integrated languages and to translate the same architecture model into different GPLs (cf. Req. TRQ-1).

Currently, this either imposes development of a new monolithic code generator for the participating languages and target GPL from scratch, which usually entails informal reuse and duplicates knowledge, maintenance, and evolution efforts, or white-box extension of existing code generators. The former are hardly reusable with new language aggregates or different target GPLs (Req. TRQ-3). The latter requires the source artifacts of the generator to be extended, introduces hardly maintainable dependencies between the generators, and enforces generator developers to comprehend all participating generators. Also, both integration approaches must be repeated for each new component behavior language and hence either yield a multitude of similar code generators or a monolithic, and hardly maintainable, all-purpose code generator. Instead, MontiArcAutomaton introduces a compositional code generation framework that enables black-box integration of participating generators with minimal effort.

In the context of C&C architectures with embedded component behavior modeling languages, such as the MontiArcAutomaton ADL, code generation has to consider three prime translation concerns: (1) Translation of structural model elements, such as component interfaces and connectors; (2) Translation of behavior models; (3) Translation of data types. This chapter presents a concept to combine code generators for these translation concerns based on:

- *Generator kinds* that yield different properties to characterize the information provided and required to combine and execute generators (cf. [RRRW14]).

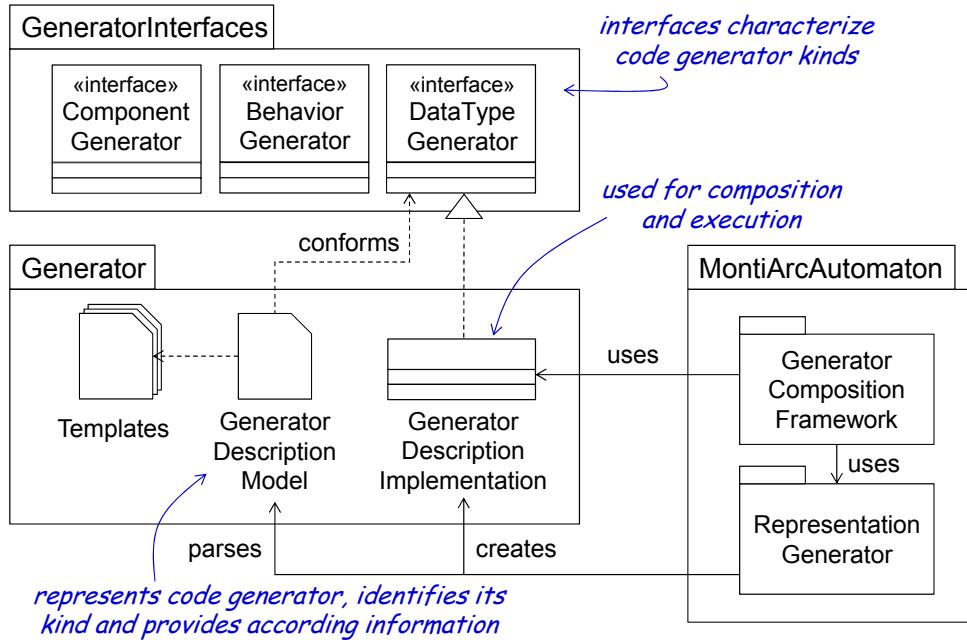


Figure 7.1: Each code generator provides a generator description model from which MontiArcAutomaton a description implementation is generated. The latter implements the interface referenced in the description model and is used by MontiArcAutomaton for composition.

- Explicit generator interfaces based on generator kinds.
- Behavior language integration as presented in Section 4.2.

This enables code generation for software architecture models using exchangeable component behavior languages, and separation of language and code generator concerns between language engineers, behavior code generator developers, and component structure generator developers (Figure 3.4). MontiArcAutomaton therefore enables black-box composition of code generators such that each participating code generator can be exchanged without changes to other participating generators. The compositional code generation framework of MontiArcAutomaton coordinates different generators, where each code generator is responsible for specific languages (Req. TRQ-3.2). Thus, composition requires no modification of the participating generators (Req. TRQ-6).

Figure 7.1 illustrates the code generator composition constituents and their relations. Each code generator provides a single code generator description model that represents it and references a single code generator interface. From this description, MontiArcAutomaton's representation generator produces an implementation of the referenced interface that is used by MontiArcAutomaton's generator composition framework to instantiate and configure the represented generator properly.

In the following, Section 7.1 introduces code generator kinds and their interfaces, before Section 7.2 describes the modeling language to describe these interfaces. Code generator composition exploits these implementations to configure, compose, and execute code generators automatically. Section 7.3 describes the composition process. Afterwards, Section 7.4 describes two sets of compositional code generators for the generation of MontiArcAutomaton ADL architectures with embedded I/O $^\omega$ automata into Java artifacts and into Python artifacts respectively. Finally, Section 7.5 discusses the presented approach and related approaches.

7.1 Code Generator Kinds

The MontiArcAutomaton ADL language family comprises the MontiArcAutomaton ADL, which provides extension points (see Section 4.2) for embedding component behavior into components. The components exchange messages via typed ports, whose type is modeled using a data type modeling language, such as class diagrams. Therefore, code generation for such integrated language families requires to cover at least the three prime translation concerns regarding C&C structures, component behaviors, and data types (cf. the interfaces depicted in Figure 7.1). On language level, component behavior languages and data type languages are exchangeable. On code generation level this is not reflected. We employ the three code generator kinds *component generator*, *behavior generator*, and *data type generator* to enable the modularity required from MontiArcAutomaton (Req. TRQ-3), which requires that code generators for all of the above aspects are exchangeable and that it should be possible to reuse, for instance, the same component structure generator with different behavior generators and data type generators - as long as these produce compatible GPL artifacts. Additionally, we require (Req. TRQ-3) that it is possible to employ different code generator implementations for the same target language, (e.g., in case a new target platform requires additional constraints not met by existing generators). Providing such compositionality requires a more precise definition of code generators than presented in Def. 1. Thus, for the scope of MontiArcAutomaton, a compositional code generator is defined as follows:

Definition 8 (Compositional Artifact Code Generator). *A compositional artifact code generator is a software that accepts models conforming to modeling language and produces valid GPL artifacts. It conforms a single code generator kind.*

This definition leaves open whether a compositional artifact code generator employs M2M or M2T transformations (cf. Section 2.1), how the input models are defined and passed to the generator, and which GPL the target artifacts conform to. It albeit requires that the GPL artifacts produced by a code generator are valid, relative to their language (e.g., a single Java method declaration is no valid Java artifact). This is not ensured by MontiArcAutomaton generators itself, but a requirement imposed on the component developers. Furthermore, it requires that each code generator conforms to a single code generator kind. For each code generation concern that requires participation in code generator composition, a unique generator kind has to be defined. Within the

scope of this thesis, component structure, behavior, and data types define the entirety of code generation concerns and thus, the entirety of code generator kinds as well. Each code generator kind explicates all information that corresponding generators provide and require for configuration, composition, and execution. A code generator kind therefore is as follows (Req. *TRQ-3.1*):

Definition 9 (Code Generator Kind). *A MontiArcAutomaton code generator kind consists of the following information provided and required to configure, compose, and execute the implementing code generator:*

1. *Name: A name uniquely identifying this generator kind in the employing application's context.*
2. *Input modeling languages: The modeling languages or language fragments the generator can process.*
3. *Well-formedness rules: Constraints restricting the processable models of the input languages.*
4. *Execution information: Information describing how the generator is invoked. With template-based code generators, this might refer to a main template, while GPL-implemented code generators might explicate a method and its signature.*
5. *Context information: Additional information that may be provided or required at generation time. This, for instance, might include the names of the artifacts to produce.*
6. *Output properties: Information on the generated artifacts (e.g., GPL or run-time environment the generated artifacts conform to).*

All generators participating in MontiArcAutomaton must provide a unique generator kind. However, the concept of generator kinds and their exploitation for generator composition is not limited to the three generator kinds defined in the scope of this thesis. In [RRRW14], we introduce a *factory generator* kind, which produces object instantiation mechanisms following the factory pattern [GHJV95]. Further generator kinds for various specific translation aspects are conceivable, such as code generators for serialization or communication issues. Furthermore, generators conforming to the generator kinds `ComponentGenerator` and `BehaviorGenerator` require to provide input modeling language, execution information, context information, and output properties. Well-formedness rules are optional for both. While behavior generators may specify a single input modeling language only, component generators may specify additional input modeling languages to reflect their ability to provide behavior generator capabilities (which allows modeling partly monolithic generators with MontiArcAutomaton as well). For generator descriptions representing generators conforming to the `DataTypeGenerator` kind, the required information is a single input modeling language and execution information only. Well-formedness rules are optional and output properties are prohibited.

Table 7.1: Each generator description requires specific information depending on the referenced generator kind (cf. Figure 7.2).

Kind	Mandatory	Optional	Prohibited
Component Generator	execution information, input modeling language, output properties	additional input modeling languages, well-formedness rules	
Behavior Generator	execution information, input modeling language, output properties	well-formedness rules	additional input modeling languages
DataType Generator	execution information, input modeling language	well-formedness rules	output properties, additional input modeling languages

Table 7.1 summarizes which generator description elements are required, allowed, and prohibited depending on the implemented generator kind.

MontiArcAutomaton identifies the participating code generators by their generator description models. Each generator description represents a code generator, for which it declares conformance to a single generator kind and provides information corresponding to the information required and provided by the generator kind. For each description, an implementation is generated. This implementation encapsulates the information the code generator provides and requires as defined in its description. MontiArcAutomaton instantiates this implementation to use the represented code generator. Afterwards, it first invokes code generation of data types, as this generation process is unrelated to component generation. It is in the application modeler's duty to ensure the data type artifacts and component artifacts produced by the code generators she selected are compatible. Once the data type generation has finished, MontiArcAutomaton configures the component generator. Besides passing the AST to generate code for, this includes passing the instantiated behavior generators as only the component generator is aware of their integration. MontiArcAutomaton then invokes the component generator which first checks the component models applying its own well-formedness rules and the well-formedness rules of the behavior code generators it received. If the AST is well-formed, the component generators traverses the passed AST and invokes behavior generators when necessary (i.e., when it visits an AST node of a behavior language).

The code generator kinds are realized in form of interfaces that prescribe the existence of methods providing or requiring the characterizing information of the respective generator kind for MontiCore code generators (Section 2.2.3). In this context, input modeling languages are productions in MontiCore grammars (cf. Req. *TRQ-5*), starting points are either FreeMarker templates or factory methods, well-formedness are realized as context conditions, and output properties are references to run-time environments. Hence, for MontiArcAutomaton, the code generator interfaces are as illustrated in Figure 7.2. Each

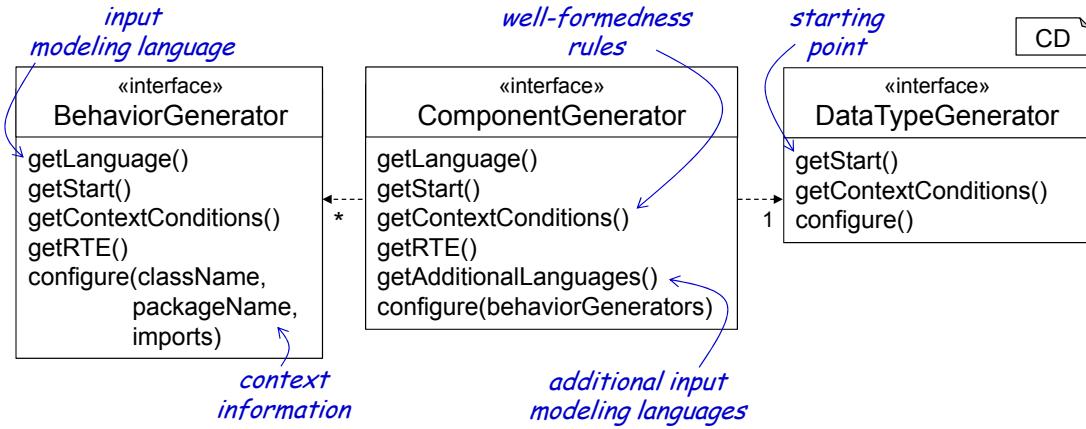


Figure 7.2: An overview of the generator interfaces employed by MontiArcAutomaton with their most important methods.

of these interfaces provides the method `getLanguage()` to retrieve the input modeling language, the method `getStart()` to return its execution information, the method `getContextConditions()` that returns its generator well-formedness rules. These well-formedness rules are not part of the language definition but restrict the models that can be translated to the target GPL. For instance, the Java code generators cannot resolve underspecification or non-determinism. Hence, they provide corresponding well-formedness rules. The interfaces representing component generators and behavior generators also impose a method `getRTE()` to retrieve the run-time environment the generated artifacts conform to (part of the output properties). The interface of component generators also imposes implementation of the method `getAdditionalLanguages()` to retrieve the component behavior languages processable by the component generator itself. All generator kinds also provide a method `configure()` to pass the context information required to generate proper GPL artifacts. The parameters expected by this method are specific to the represented generator kind.

These interfaces wrap all information required to enable code generator composition with MontiArcAutomaton. The composition mechanisms of MontiArcAutomaton can be applied to other interfaces and hence other C&C ADLs as well. Overall, the relations between compositional artifact code generators, generator description models, generator interfaces, and generator kinds are as depicted in Figure 7.3. A compositional artifact code generator contains exactly one generator description models. That model declares conformance to exactly one generator interface, which represents exactly one code generator kind.

Before composing code generators based on their kinds, the content of the respective interfaces' implementations has to be defined. Therefore, Section 7.2 presents the code generator description language to provide the interfaces' implementations.

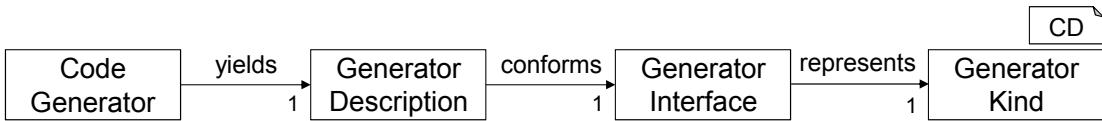


Figure 7.3: The relations between code generators, generator description models, generator interface, and generator kinds.

7.2 Code Generator Description Language

MontiArcAutomaton code generators must exhibit which generator kind they conform to and provide information on the properties required by that generator kind. Therefore, Section 7.2.1 presents the code generator description language elements, which provide a concise formalism to describe that information. Afterwards, Section 7.2.2 presents the language’s symbol table and Section 7.2.3 its context conditions.

7.2.1 Language Elements

The code generator description language elements reflect the properties of generator kinds in terms of their MontiCore realization, i.e., for each code generator kind property as realized with the corresponding interface, there is a corresponding language element. Listing 7.1 illustrates most of the code generator description language elements with the generator description ComponentsJava for a generator that translates component models into Java artifacts.

Generator descriptions are structured in packages (l. 1) and begin with the keyword `generator` followed by a name, the keyword `conforms`, and the name of the generator interface the represented generator implements (l. 3). Afterwards it describes three properties of the represented code generator (ll. 5-9). First it describes how the represented generator is invoked (l. 5). Afterwards, it describes which the input modeling language the represented code generator can process (l. 7), which is identified by the rule `MAAComponent` of the MontiArcAutomaton ADL MontiCore grammar (cf. Listing A.2). Finally, it describes that the represented code generator produces artifacts conforming to a specific run-time environment, which, for instance, describes how generated GPL artifacts and handcrafted GPL artifacts interact (l. 9) as presented in Section 7.4.

The following paragraphs detail the individual language elements and their properties before subsequent sections describe the language’s symbol table and context conditions.

Generator Description Declaration

A generator description begins with the keyword `generator`, followed by the description’s name. After the keyword `conforms` it references the interface of the type the represented code generator implements. This construction restricts each generator description, and hence each code generator, to a single generator kind. The referenced

```
1 package generators.componentsjava;
2
3 generator ComponentsJava conforms ComponentGenerator {
4
5   start generators.componentsjava.MainTemplate;
6
7   language languages.adl.MontiArcAutomaton.MAACComponent;
8
9   rte runtimes.javatimesyncdelegation;
10 }
```

GD

Listing 7.1: The code generator description model ComponentsJava represents a code generator that translates component models into Java artifacts.

interface's name must be fully qualified¹ and after this name, curly brackets delimit the generator description's body.

```
1 generator AutomataPython
2   conforms generators.BehaviorGenerator {
3     // ...
4 }
```

GD

Listing 7.2: The generator description AutomataPython implements the generator interface generators.BehaviorGenerator.

Listing 7.2 illustrates a generator description of name AutomataPython (l. 1) that represents a behavior generator. The generator description establishes the latter by declaring conformance to the generator interface generators.BehaviorGenerator representing the behavior generator kind (l. 2).

Starting Point

Each code generator must be invoked and the ways to invoke code generators differ. While most MontiCore generators employ a FreeMarker template as starting point, there is neither a convention on that template's name or a general consensus that a code generator must start with a template. Therefore, MontiArcAutomaton enables specifying factory methods of Java classes as starting point as well. The `start` element of generator description models thus may refer to a template or a static method only and does so by stating its complete name.

¹The interface's name is used once only and hence importing it, which requires referencing it via its fully qualified name, provides no advantage.

```

1 generator ClassdiagramPython
2   conforms generators.DataTypeGenerator {
3
4   start generators.cpython.MainTemplate;
5   // ...
6 }
```

GD

Listing 7.3: The generator description `ClassdiagramPython` defines that the represented generator implements the `DataTypeGenerator` generator interface (l. 2) and that it is started via the template `generators.cpython.MainTemplate` (l. 4).

Input Modeling Language

To enable proper composition, each code generator must explicate which models it can process. Therefore, the generator description characterizes the processable models by the modeling language that defines these models. MontiArcAutomaton code generators generally process models of MontiCore languages. Therefore, MontiArcAutomaton identifies this modeling language with a MontiCore grammar production and code generator descriptions refer to these productions.

Requiring a concrete production instead of a complete grammar allows selecting an arbitrary production that grammar. This reflects that language embedding also does not require to embed complete languages into the MontiArcAutomaton ADL. Each code generator description thus must contain the keyword `language`, followed by the name of a production of a MontiCore grammar. This name consists of the referenced grammar's package name concatenated with the grammar's name and the name of the production defining the language. Listing 7.4 illustrates this with a reference to the production `TableContent` of the grammar

```

1 generator IOTablePython
2   conforms generators.BehaviorGenerator {
3
4   start generators.componentspython.MainTemplate;
5
6   language languages.iotable.IOTable.TableContent;
7   // ...
8 }
```

GD

Listing 7.4: Generator description `IOTablePython` explices that the represented generator can process models derivable from production `TableContent` of grammar `languages.iotable.IOTable` (l. 6).

Well-formedness Rules

Code generators may additionally restrict the processable models with well-formedness rules, which enables preventing code generation for unsuited models, such as the translation of non-deterministic automata to Java. For MontiArcAutomaton code generators, well-formedness has the form of context conditions. Code generator descriptions declare context conditions as sets of qualified names referencing MontiCore context conditions.

As illustrated in Listing 7.5, generator description models may declare the represented generators' context conditions in two ways and both begin with the keyword `contextconditions`. If this keyword is followed by a set of qualified names that reference the context conditions' Java classes (ll. 10-13), these must fully identify the respective classes. However, as these usually are part of the represented code generator and reside in the same or similar packages, a common prefix of the package name may be extracted to abbreviate the individual references and ease generator description modeling. Therefore, the keyword `contextconditions` may be followed by the keyword `in` and a qualified name representing a prefix of the package the context conditions reside in. The second context condition declaration block of generator description `RobotArmPython` (ll. 15-18) uses this pattern to define that both context conditions `AtMostTenLocations` and `determinism.NoUnderspecification` (l. 17) reside in the package `generators.robotarmpy`. The qualified name after `in` must not describe the complete package of the following context conditions as illustrated with `NoUnderspecification`. This context condition ultimately resides in the package `generators.robotarmpy.determinism`. MontiArcAutomaton determines the complete set of code generator context conditions by collecting all `contextconditions` blocks and calculating the complete qualified name of each referenced context condition. Specifying multiple `contextcondition` blocks allows declaring multiple logically distributed context conditions conveniently.

Run-Time Environment

Run-time environments are means to provide common mechanisms to applications of MontiArcAutomaton without generating these for each application anew. The issues tackled by run-time environments include, but are not limited to, compatibility of generated and handcrafted artifacts, component communication, serialization, and error handling. MontiArcAutomaton, for instance, considers artifacts for component structure and for component behavior compatible if they agree on interacting each other using the same run-time environment. Therefore, generator descriptions explicate which run-time environment the represented generators conform to. These declarations start with the keyword `rte`, followed by the qualified of a run-time environment (cf. Section 7.4.1). The generator description `ComponentsPython` depicted in Listing 7.6 represents a code generator that transforms MontiArcAutomaton components (l. 6) into Python artifacts. To enable checking whether the behavior generators it might be composed with are compatible, it declares that the represented code generator produces artifacts compatible to the run-time environment `runtimes.pythonuntimed` (l. 8).

```

1 generator RobotArmPython
2   conforms generators.BehaviorGenerator {
3
4     start generators.robotarmpy.EmbeddedMain;
5
6     language ra.RobotArm.RobotArmProgam;
7
8     rte runtimes.pythontimesync;
9
10    contextconditions {
11      generators.robotarmpy.NoComplexDataTypes,
12      generators.robotarmpy.RestrictedProgramSize
13    }
14
15    contextconditions in generators.robotarmpy {
16      AtMostTenLocations,
17      determinism.NoUnderspecification
18    }
19 }
```

GD

Listing 7.5: Generator description RobotArmPython contains two sets of context conditions (ll. 10-18) describing the contained conditions directly (ll. 10-13) and via abbreviation (ll. 15-18).

```

1 generator ComponentsPython
2   conforms generators.ComponentGenerator {
3
4     start generators.cpython.MainTemplate;
5
6     language languages.MontiArcAutomaton.MAAComponent;
7
8     rte runtimes.pythonuntimed;
9 }
```

GD

Listing 7.6: The code generator represented by description ComponentsPython produces artifacts compatible to the runtimes.pythonuntimed runtime environment (l. 8).

Additional Input Modeling Languages

Component code generators may provide means to produce artifacts for component behavior languages as well. This, for instance, is useful, when the base language, prior to component behavior language embedding, already provides means to model component behavior. Similar to the input modeling language, the supported component behavior modeling languages are identified via their defining grammar productions. The corresponding production starts with the keyword behaviors followed

by a comma-separated list of input modeling languages identified by their complete path, grammar name, and production to be embedded. The generator description `ArchitecturePython` depicted in Listing 7.7 illustrates this by representing a component code generator (l. 2) that transforms components of `MontiArcAutomaton`'s production `MAAComponent` (l. 4), via start method `arcpy.Main.generate()` (l. 6) into artifacts compatible to the run-time environment `runtimes.pythonuntimed` (l. 8). The represented generator also can process components with embedded `TableContent` models and embedded `RobotArmProgram` models (ll. 10-13).

```
1 generator ArchitecturePython
2   conforms generators.ComponentGenerator {
3
4   language languages.MontiArcAutomaton.MAAComponent;
5
6   start arcpy.Main.generate();
7
8   rte runtimes.pythonuntimed;
9
10  behaviors {
11    iotable.IOTable.TableContent,
12    ra.RobotArm.RobotArmProgam
13  }
14 }
```

GD

Listing 7.7: Generator description `ArchitecturePython` describes that the represented code generator can translate models of the behavior languages `TableContent` and `RobotArmProgram` (ll. 10-13).

7.2.2 Symbol Table

Properties of the generators to be integrated need to be checked prior to composition. For instance, the processable modeling languages must be checked against the modeling languages employed in the software architecture to be processed. This requires integration of the `MontiArcAutomaton` language family with the generator description language. Enabling such integrated well-formedness checking without coupling both languages ASTs requires integration on symbol table level. `MontiArcAutomaton` integrates the generator description language via language aggregation, as models of both languages remain independent artifacts, but are interpreted together. Thus, the generator description language must provide a symbol table. The generator description symbol table consists of a single entry type that provides all information relevant to context condition checking and further analyses. The properties of `GeneratorDescriptionEntry` entries, depicted in Figure 7.4, directly translate to generator description model properties, but omit AST details, such as whether the context conditions are modeled as a single set or multiple sets.

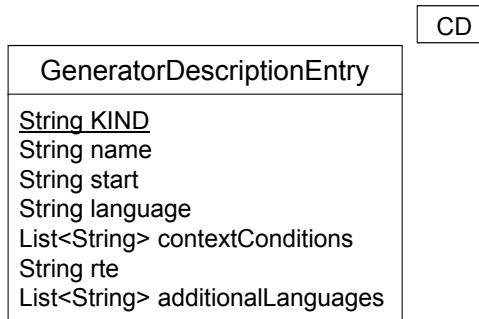


Figure 7.4: Entries of type `GeneratorDescriptionEntry` provide all information from the corresponding generator description model.

All properties of `GeneratorDescriptionEntry` symbol table entries are names, which must be interpreted correctly for context condition checking. For instance, the property `language` references to a production of a grammar. The context conditions checking the validity of this reference must interpret the contained name properly.

7.2.3 Context Conditions

The generator description context conditions ensure the well-formedness of generator descriptions' elements. This includes ensuring the uniqueness of specific elements, the adherence of certain conventions, the elements' referential integrities, and their type correctness. As many properties of generator description models reference other artifacts, checking their well-formedness requires resolving these artifacts using MontiCore's symbol table mechanisms (cf. Section 2.2.1). The following presents the uniqueness conditions, convention conditions, referential integrity conditions, and type correctness conditions of generator description models.

Uniqueness Conditions

Uniqueness conditions ensure that certain elements of generator descriptions occur at most once per model. Violated context conditions may produce errors or warnings depending on the duplicated elements.

GU1: Only context condition blocks and behavior blocks may occur multiple times.

All generator description language elements but context condition blocks and behavior blocks (i.e., `start`, `language`, and `rte`) may occur only once. Declaring multiple of these elements would entail inconsistent code generator representations. Only `contextconditions` and `behaviors` may be declared multiple times. Violating this raises errors as depicted in Listing 7.8, where the generator description `ComponentsWithAutomata` declares multiple `start` elements (ll. 4-5) and multiple `contextconditions` elements (ll. 7-8). The former produces errors as displayed.

```

1 generator ComponentsWithAutomata
2   conforms generators.ComponentGenerator {
3
4   start cwa.Main.run(); ✖ // Multiple start elements.
5   start cwa.EmbeddedMain; ✖ // Multiple start elements.
6
7   contextconditions { generators.cwa.NoInnerComponents }
8   contextconditions { generators.cwa.NoGenericType }
9   //...
10 }
```

GD

Listing 7.8: The generator description `ComponentsWithAutomata` is not well-formed as it declares two start elements (ll. 4-5).

GU2: No duplicate context conditions.

Executing a context condition multiple times produces new warnings or errors for each execution. However, as MontiArcAutomaton considers the entirety of context conditions declared in generator description models as a single set of context conditions, each context condition will be executed at most once per model. Thus, specifying a context condition multiple times does not change the resulting number of warnings or errors. Therefore, this context condition raises warnings for generator description models mentioning the same context condition multiple times.

```

1 generator IOTablesGroovy
2   conforms generators.ComponentGenerator {
3
4   contextconditions {
5     iotgen.NoInnerComponents ⚠ // Duplicate context condition.
6   }
7
8   contextconditions in iotgen {
9     NoInnerComponents, ⚠ // Duplicate context condition.
10    NoGenericType
11  }
12  // ...
13 }
```

GD

Listing 7.9: The generator description `IOTablesGroovy` mentions the context condition `iotgen.NoInnerComponents` two times: once in the first `contextconditions` block (l. 5) and once in the second block (l. 9).

GU3: No duplicate behaviors.

For component generators, declaring an additional input modeling language multiple times is prohibited as well. As MontiArcAutomaton treats these input modeling languages treated as a set, such redundancy does not raise errors, but warnings only.

Convention Conditions

Convention conditions facilitate modeling by enforcing conventions on modeling elements, thereby reducing the effort of comprehending models. For instance, a typical convention of many object-oriented GPLs is that the names of classes begin with a capital letter. Hence they are easier to distinguish from non-class names and reduce the need to look up names. As the generator description language references existing artifacts only, these names are governed by the artifacts' respective type (i.e., a reference to a Java class must conform to the requirements for naming Java classes). Hence there is only a single generator description convention condition, which ensures the well-formedness of generator description names and raises an error.

GC1: Generator descriptions start with a capital letter.

To help distinguishing the names of generator descriptions other elements, their names must start with a capital letter. Listing 7.10 illustrates this with a generator description of name `flowchartPython` (l. 1) that violates this rule.

```
1 generator flowchartPython    ✘ // Generator descriptions
2   conforms BehaviorGenerator { // start upper-case.
3     // ...
4 }
```

GD

Listing 7.10: The generator description `flowchartPython` violates GC1 as its name starts with a lower-case letter (l. 1).

Referential Integrity Conditions

Context conditions regarding referential integrity ensure that names referenced in generator description models refer to existing artifacts and that they are used correctly. Their existence is crucial and, hence, violation of referential integrity conditions raises errors.

GR1: The referenced generator interface exists.

Each generator description must refer to a generator interface to define the type of generator it represents. As MontiArcAutomaton composes generators based on their interfaces the referenced interface must exist and be available to the code generator.

Otherwise, generator configuration, composition, and execution will fail. The generator description `SequenceDiagramPython` references the interface `ArcGenerator` (l. 2), which is unavailable to the code generator. Consequently, `MontiArcAutomaton` rises the depicted error.

```
1 generator SequenceDiagramPython
2   conforms ArcGenerator { ❌ // Inexistent interface
3     // ...
4 }
```

GD

Listing 7.11: The interface referenced by generator description `ArcGenerator` cannot be resolved as it does not exist or is unavailable.

GR2: The referenced starting point exists.

Similarly, the starting point of the represented generator must exist and be available. For the generator description `StatechartPython` of Listing 7.12, this does not hold, as the referenced template `scp.Embedded` (l. 4) is not available. Hence, `MontiArcAutomaton` produces an error.

```
1 generator StatechartPython
2   conforms generators.BehaviorGenerator {
3     // ...
4   start scp.Embedded; ❌ // Inexistent template 'scp.Embedded'.
5 }
```

GD

Listing 7.12: The generator description `StatechartPython` references a template as starting point (l. 4) that is unavailable to the generator it represents.

GR3: The referenced input modeling language exists.

The third mandatory constituent of each code generator description is the input modeling language of the represented generator. If the referenced input modeling language is unavailable to the generator, `MontiArcAutomaton` raises an error as depicted in Listing 7.13. Here, the behavior generator `RecipePython` referenced the rule `Main` of the grammar `Recipe`, which is unavailable to the represented generator.

GR4: The referenced starting point exists.

Generator description models can either reference main templates or Java factory methods as entry points. Whichever it references must exist. For templates, this is

```

1 generator RecipePython
2   conforms generators.BehaviorGenerator {
3
4   language Recipe.Main; ✖ // Inexistent language 'Recipe.Main'.
5 }
```

GD

Listing 7.13: The generator represented by description RecipePython cannot access the referenced language Recipe.Main (l. 4).

validated on file-basis. For factory methods, the referenced type is parsed using the Java/P modeling language and the existence of the specified method is validated.

GR5: The referenced context conditions exist.

Whenever context conditions are declared, these must exist as well. Please note that for multiple references to the same missing context condition, MontiArcAutomaton raises multiple errors. Listing 7.14 illustrates this error with the context condition tap.RequireDeterminism (ll. 4-5).

```

1 generator TimedAutomataPython
2   conforms generators.BehaviorGenerator {
3   // ...
4   contextconditions in tap {
5     RequireDeterminism; ✖ // Inexistent context condition
6     } // 'tap.RequireDeterminism'.
7 }
```

GD

Listing 7.14: The generator description TimedAutomataPython references the missing context condition tap.RequireDeterminism (ll. 4-5).

GR6: The additionally provided behavior languages exist.

The existence of the behavior input modeling languages of component generators is crucial as well. Consequently, MontiArcAutomaton produces errors if any of these is missing. The generator description PlainComponents of Listing 7.15 references the missing language lng.Automata.Body. Consequently, MontiArcAutomaton produces the depicted error.

Type Correctness Conditions

The generator description language features a single type correctness condition. This condition ensures that all required information for the referenced generator interface is defined in the generator description, so that MontiArcAutomaton can generate proper

```
1 generator PlainComponents
2   conforms generators.ComponentGenerator {
3     // ...
4
5   behaviors { ✘ // Inexisting language 'lng.Automata.Body'.
6     lng.Automata.Body,
7     lng.IOTable.Table,
8     lng.RobotArm.Progam;
9   }
10 }
```

GD

Listing 7.15: The generator represented by `PlainComponents` cannot access the referenced language `lng.Automata.Body` (l. 6).

implementations of the referenced interface. As this also is crucial to generator composition, violating this condition raises an error as well.

GT1: Generator descriptions provide the information required for implementing the referenced generator kind.

Each generator description references a generator interface that represents the generator type the represented code generator should conform to. MontiArcAutomaton generates proper generator description implementations (cf. Figure 7.1) to be part of the generator to utilize these for composition. Hence, the code generator description models must provide all information required by the referenced generator interface as summarized in Table 7.1.

Listing 7.16 shows the exemplary generator description `AutomataLisp` that describe a component behavior code generator that translates automata to Lisp artifacts. As the represented generator should implement the interface `BehaviorGenerator` (l. 2), MontiArcAutomaton expects the generator description to provide information on the run-time environment the generated artifacts conform to. For the same reason, designating additionally supported behavior languages (ll. 8-10) is prohibited.

7.3 Code Generator Composition

Generating integrated component artifacts with reusable component generators and behavior generators requires ensuring compatibility of the generators and the produced artifacts as well as their execution for whichever model parts they are responsible. To ensure the compatibility of generators, we have introduced code generators types. These types govern how and when conforming generators are executed. Artifact compatibility instead is based on conformance to run-time environments. Generator composition in MontiArcAutomaton is realized as the execution of responsible code generators while traversing the AST nodes of architectures with embedded behavior language models.

```

1 generator AutomataLisp ✖ // Run-time system missing.
2   conforms generators.BehaviorGenerator {
3
4     start gen.lisp.Starter.run();
5
6     language lng.Automata.AutomatonContent;
7
8     behaviors { ✖ // Prohibited for behavior generators.
9       lng.SimpleArc.Component
10     }
11 }
```

GD

Listing 7.16: The generator description AutomataLisp declares to represent a BehaviorGenerator (l. 2), but omits designation of a run-time environment. Also it specifies additionally supported behaviors (ll. 8-10), which is prohibited for behavior generators.

The implementations generated from generator description models are the prime constituents of MontiArcAutomaton code generator composition. These implementations provide all information required for composition and yield mechanisms to enable structured information exchange between code generators. Each MontiArcAutomaton code generator is deployed with such an implementation that can be identified via naming convention based on the generator descriptions name. Hence, composition requires to instantiate, configure, and execute the code generators represented by these implementations in the correct order.

Actual composition of component structure generators and component behavior generators can be performed in three ways that differ in the composition controlling elements and the granularity of code generation control required:

1. The generator composition framework traverses the AST itself and invokes the responsible generators for each AST node.
2. The component generator traverses the AST and passes each component behavior AST node it encounters and additional context information to the composition framework. The framework delegates generation of proper behavior artifacts to the respective behavior generators and returns control to the component generator again.
3. The component generator traverses the AST and delegates calls to behavior generators itself.

The first alternative requires very broad interfaces to code generators that allow specification of required information per AST node type and passing each AST subtree to the component generator requires it manage these trees properly. For instance, the component generator developer has to decide whether to generate something from the AST

subtree representing the interface of a component or from the individual interface element subtrees. Usually, code generators process the AST elements as required and do not need to manage invocation with redundant AST elements. Enforcing such management complicates code generator development. In the second alternative, component code generators traverse the AST and generate component structure artifacts as required. Whenever the generator encounters a behavior AST node, it collects all information required to produce a compatible artifact from the behavior node and passes that, together with the behavior node, to the composition framework. The latter configures and invokes the behavior generators that produce according artifacts. This alternative requires only small interfaces between component code generators and the composition framework, but produces redundancy as the composition framework delegates invocation to behavior generators without adding information. Removing this indirection leads to the third alternative, in which the component generators invoke behavior generators directly. MontiArcAutomaton implements the third alternative. Here, the infrastructure provides all necessary information, such as the configured behavior generators and the component generators invoke these. This leaves control at the component generators without composition framework interferences and enables optimizing code generation.

In the following, Section 7.3.1 describes the process to develop MontiArcAutomaton code generators, which includes production of these implementations. Afterwards, based on participating code generators selected by the application modeler, the generators' implementations must be instantiated and executed. Therefore, Section 7.3.2 explains how MontiArcAutomaton instantiates, configures, and executes code generators.

7.3.1 Developing MontiArcAutomaton Generators

MontiCore code generators employ M2T transformations powered by the FreeMarker² template engine. Usually, these code generators employ dedicated start templates that configure and control code generation. However, these start templates do not specify which information the generator requires to produce proper artifacts and the only means to specify such is in form of template commands. This requires the generator developer of a component generator to understand the internals of templates of participating behavior generators – which assumes the white-box view on code generators MontiArcAutomaton aims to avoid. MontiArcAutomaton code generators also employ FreeMarker, support template operators and template helpers, and may begin with a start template, but require a more sophisticated machinery to enable composition. With MontiArcAutomaton, application modelers may use monolithic code generators (cf. Req. *TRQ-4*) for specific language aggregates as well as modular code generators where each translates a subset of the participating languages into compatible GPL code. Composition of modular code generators is aligned with MontiArcAutomaton's language extension mechanisms and exploits the three code generator kinds introduced earlier (cf. Section 7.1).

Development of according code generators follows the activities depicted in Figure 7.5. First, the generator developer decides which generator kind the generator under devel-

²Website of the FreeMarker template engine: <http://freemarker.org/>

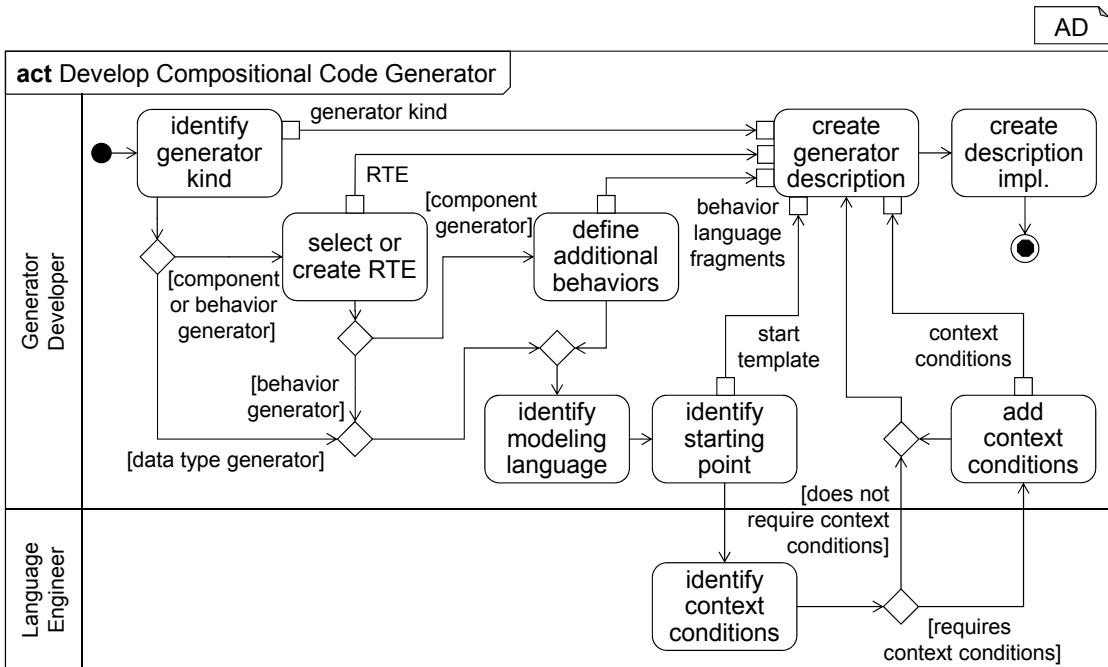


Figure 7.5: The activities required to develop a compositional MontiArcAutomaton code generator.

opment implements and selects the proper generator interface. For component generators and behavior generators, she afterwards selects a run-time environment the generated code will conform to. If no such RTE exists, the run-time environment developer must provide one. If the generator under development is a component generator, she also defines which additional input modeling languages the generator supports (cf. Section 7.2.1). Afterwards, invariant of the generator's kind, she identifies its input modeling language and its starting point (either a FreeMarker template or a Java factory method). If the generator requires context conditions - a question answered in collaboration with the language engineer of the generator's input modeling language, she develops these as well. Then, she documents the selected generator kind, run-time environment, input modeling languages, starting point, and context conditions in the generator description model. Finally, she creates an implementation from this model.

While MontiArcAutomaton supports handcrafted generator description implementations, it also yields a code generator for this. The generator representation generator processes generator description models and transforms these into Java classes implementing the referenced interfaces such that the implemented methods return the values of their related properties (e.g., `getStart()` of the implementation returns the qualified name of the start template or start method as declared via `start` by the generator description model). The generated implementation also persists the configuration information, which enables the represented generator to access this information at run-time.

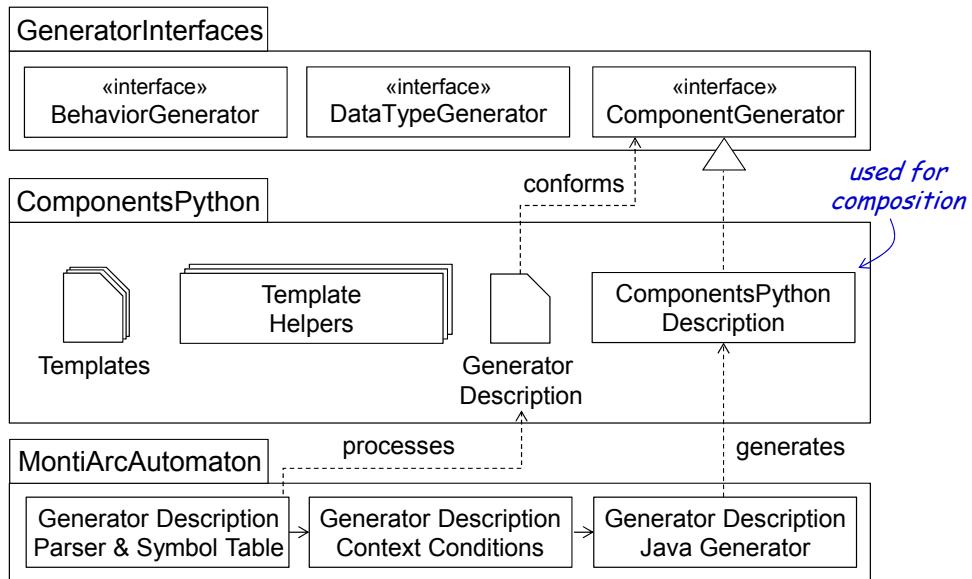


Figure 7.6: The constituents of the component generator presented in Listing 7.6. (cf. Figure 7.1).

Considering the generator description `ComponentsPython` depicted in Listing 7.6, the generator description infrastructure of `MontiArcAutomaton` will create a proper implementation of this interface as depicted in Figure 7.6. This implementation contains all information of the generator description model and, as the generator description declares that the represented generator implements the interface `ComponentGenerator`, implements that interface as well. The code generator thus comprises templates that describe the target artifacts, template helpers that perform complex calculations, a generator description model, and the generated description implementation that can be exploited for composition.

7.3.2 Instantiating and Executing Composable Generators

The application modeler selects the code generators appropriate for the target platform the software architecture should run with. To this end, she selects a component generator, multiple behavior generators, and a data type generator per target platform (cf. Figure 3.8). After selecting the participating generators, their qualified names are passed to the `MontiArcAutomaton` *generator orchestrator*.

The latter instantiates the generator description implementation of each referenced generator via the corresponding generator kind's interface to avoid dependencies of `MontiArcAutomaton` to the participating generators. Afterwards, it configures the component generator with the participating behavior generators via its `configure()` method. The component generator persists this information for generator execution.

Figure 7.7 depicts the participating components, models, and implementations trans-

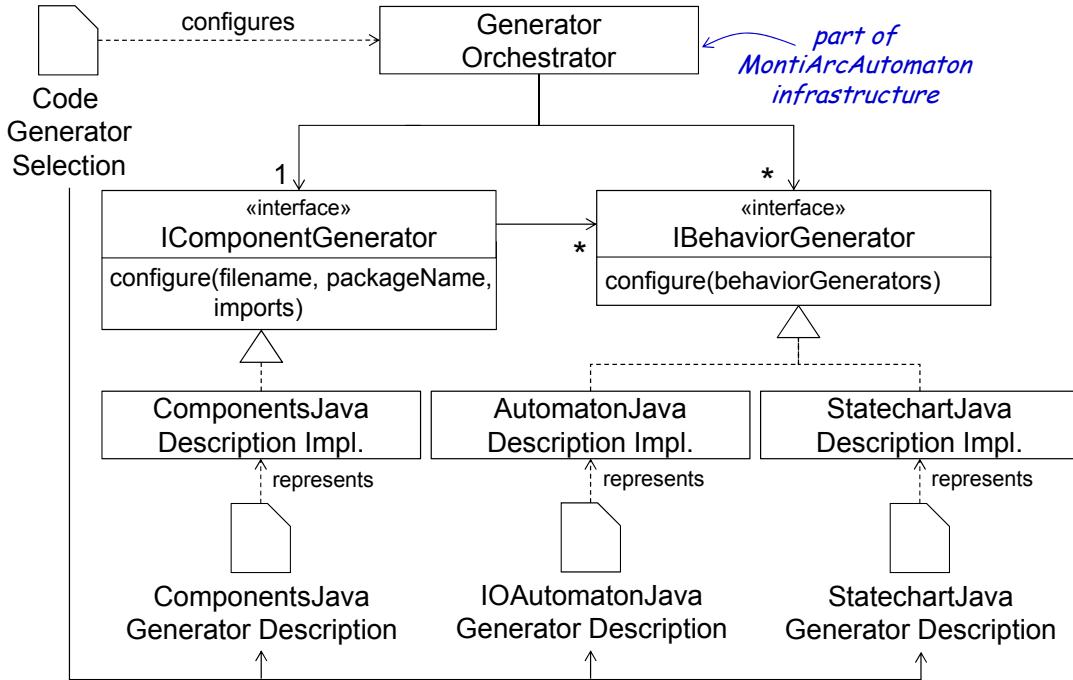


Figure 7.7: The MontiArcAutomaton generator orchestrator instantiates generators via the interfaces representing their kinds and implemented by their generated description implementations.

lation of components with embedded automata and Statecharts to Java. The generator selection is input to the generator orchestrator, which knows the MontiArcAutomaton code generator interfaces and how to compose these. The generator selection also references the three code generators required to translate such components by their generator description models. MontiArcAutomaton generates a Java implementation of the interface referenced in each model that contains all information from the respective model. The generator orchestrator can instantiate these generators via their interfaces and execute these to produce Java artifacts. To this end, the generator orchestrator starts code generation by invoking the starting point of the data type generator as explicated by its description implementation as depicted in Figure 7.8. This generation phase is independent of component generation and behavior generation as the participating code generators do not need to exchange information at generation-time. Afterwards, the data type generator has produced data type artifacts and the generator orchestrator invokes the component generator via its starting point. After passing the instantiated behavior generators to it, the component generators traverse the AST of MontiArcAutomaton components during which they also visit contained embedded behavior models. If the currently visited AST node represents an element of the MontiArcAutomaton ADL, it is processed by the component generator. If it is an element of a behavior language, the component generator identifies the responsible behavior generator via the input model-

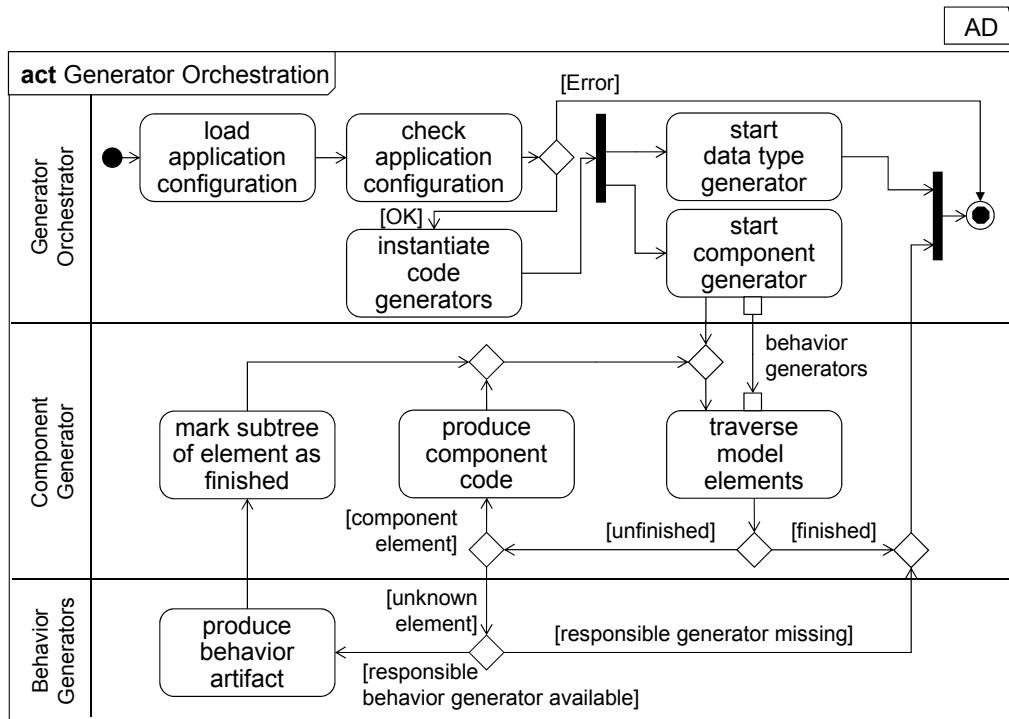


Figure 7.8: Generator orchestrator, component generators, and behavior generators interact to compose their activities.

ing language explicated by the respective generator's description implementation. While there may be more than one generator per behavior language, composition currently supports only a single behavior generator per behavior language. MontiArcAutomaton informs about invalid generator selections.

In case a responsible behavior generator is available, the component generator configures it with the information required to produce a compatible component behavior artifact. In case of the interface `BehaviorGenerator`, this amounts to passing the name of the resulting class, the package it should reside in, and the import statements of the component model via the interface's `configure()` method. The behavior generator's description implementation persists this information and the component generator invokes the behavior generator via its starting point, thereby passing the behavior model's AST node. The behavior generator traverses the behavior model's AST and may access information persisted in its description implementation to produce compatible artifacts. After processing the behavior model's AST, each behavior generator produces a valid GPL artifact (cf. Def. 8) and returns control to the component generator. The component generator marks the currently processed behavior model AST node and its subtree as processed and continues traversing until all nodes have been processed.

Figure 7.8 illustrates the activities of generator orchestrator, component generator and behavior generators during composition. The main composition activity is part of

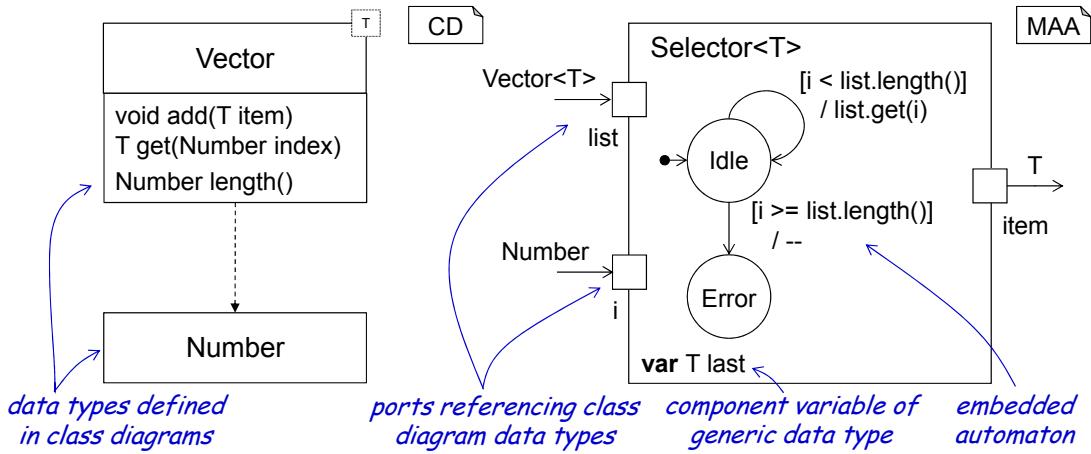


Figure 7.9: The atomic component `Selector` returns the i -th element of the passed list using the depicted class diagram data types.

the component generator, which processes the components surrounding the embedded behavior models. Therefore, of the three generator kinds, only the component generator knows the context information (cf. Req. 5) required to configure the behavior generators. Consequently, it manages execution of responsible behavior generators.

7.4 Two Compositional Code Generator Families

MontiArcAutomaton code generators are of specific kinds and implement the respective interfaces. They do so by providing generator description models that hold corresponding information and are translated into generator description implementations. These implement the generators' interfaces and provide means to persist context information for generator composition.

This section presents two code generator families. A *code generator family* is a set of code generators comprising at least one component code generator, at least one data type generator, and arbitrary many behavior generators that produce artifacts conforming to the same run-time environment. Hence, these artifacts and, by design of the generator interfaces, the generators can be composed. The families comprise five code generators: Three of these conform to a Java RTE and translate MontiArcAutomaton with embedded AUTOMATA models that reference UML/P class diagrams for data types into artifacts compatible to Java 1.8. The remaining two translate the same modeling language family into Python 3.3 artifacts compatible to ROS Indigo³. While the Java generators are completely modular, i.e., there are separate code generators for models of MontiArcAutomaton ADL, AUTOMATA, and UML/P class diagrams, the Python/ROS component generator takes care of creating artifacts for AUTOMATA models as well.

³ROS Indigo website: <http://wiki.ros.org/indigo>

Running example for both generator families is the component `Selector` depicted in Figure 7.9. This component employs an embedded automaton to select elements from a list. To this end, it features an input port `list` of class diagram type `List<T>` and another input port `i` of class diagram type `Number` (provided by MontiCore). If the received index `i` is within the list's length. The embedded automaton returns the `i`-th element. Otherwise, it switches to an error state that cannot be left.

7.4.1 A Code Generator Family for Java Systems

The code generator family for Java translates MontiArcAutomaton models into Java classes without further requirements on target platforms. A monolithic code generator for this translation was presented in [RRW13b]. The language family produces similar artifacts but differs in comprising modular code generators and integrating handcrafted artifacts and behavior artifacts differently.

While composability of the participating generators is enabled by design of the generator interfaces and MontiArcAutomaton's composition procedure (cf. Section 7.3), the compatibility of generated artifacts depends on the run-time environment they are compatible to. Therefore, the next sections describe a run-time environment for Java that component generators and behavior generators conform to, present the family's component code generator with its generator description and the behavior generator responsible for translating embedded AUTOMATA models into compatible artifacts, and describe the family's class diagram generator, which extends the generator presented in [Sch12].

The Run-Time Environment

MontiArcAutomaton run-time environments are sets of target GPL modules that provide solutions to common component tasks. This includes, for instance, communication, serialization, persistence, and integration of handcrafted artifacts. As such, it is tied to a specific GPL and target platform properties. For the latter, it may assume the existence of certain, APIs, framework, or libraries. Correct usage of these modules must be ensured via convention and documentation.

The `javats` run-time environment for this family of Java generators describes the interfaces and interaction of components, ports, and variables work, as well as the integration of handcrafted and generated behavior implementations with component implementations. Although only the component generator produces artifacts depending on components, ports, and variables at generation time, bundling these with the component generator would introduce ties from the generated system to it. Instead, these types are part of the run-time environment, which generated system and component generator depend on. Thus, the `javats` run-time environment comprises of Java classes and interfaces to represent components, ports, variables, and behavior implementations the family's generator developers agree on.

Figure 7.10 shows the constituents of `javats`, which are an abstract class `Component`, the two classes `Port` and `Variable` that both extend the abstract class `DataSource`, and the three interfaces `Computable`, `Result`, and `Input`. Code generators whose

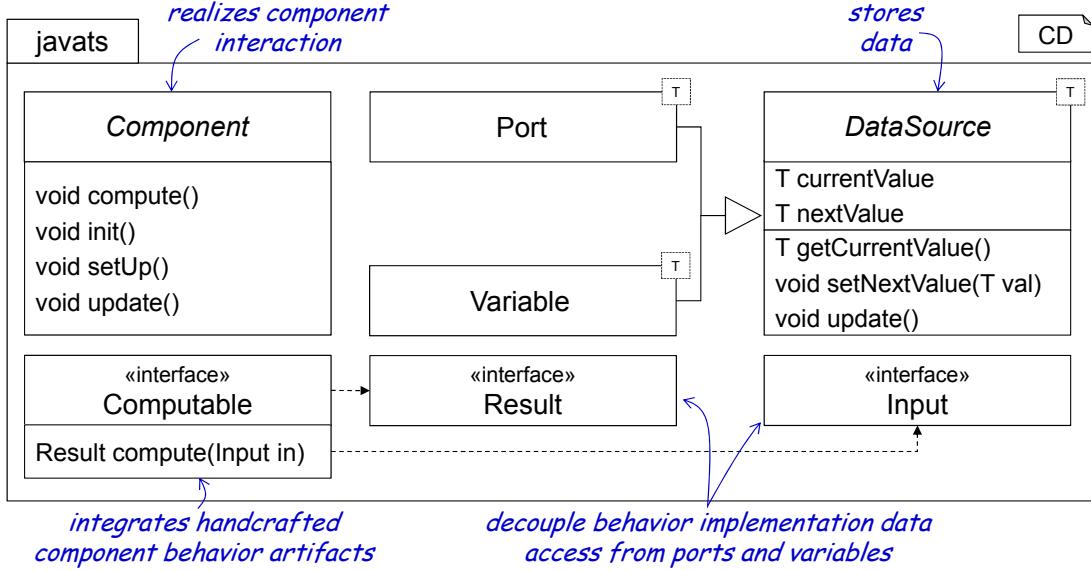


Figure 7.10: The `javats` run-time environment for Java component implementations.

descriptions declare conformity to `javats` are expected to produce component artifacts that extend the class `Component`. The class `Component` is unaware of `Port` and `Variable` instances. As the actual ports and variables of a component model are available at code generation time only, `Component` cannot know these apriori. Instead of managing lists of ports and variables, the code generators producing subclasses of `Component` are expected to produce proper realizations. This enables, for instance, producing individual members per port and variable. Furthermore, this decouples the notion of components from ports and variables and enables component generators to employ their own types for that. The types `Port` and `Variable` wrap functionalities of ports and variables. In consequence, they provide the similar features and behavior, wherefore they extend the abstract class `DataSource`. The class `DataSource` provides access to two values of generic type `T`: the current value of this data source and its next value.

Atomic components conforming to `javats` are expected to integrate handcrafted and generated component behavior implementations via the delegator pattern [GHJV95], where the delegate implements the interface `Computable`, to fulfill Req. *TRQ-2*. Employing the delegator pattern is a feature of this run-time environment and using other integration patterns [GHK⁺15] is feasible as well. This interface yields two generic type parameters `T` and `U` that require type arguments to implement the interfaces `Input`, and `Result` respectively. As behavior generators conforming to `javats` are expected to produce component behavior implementations that implement `Computable` as well, the integration of handcrafted and generated component behavior implementations can be treated uniformly for both. The interfaces `Result` and `Input` are expected to be

implemented by data structures generated from the components' ports and variables. At design-time, these data structures separate the concerns of application programmers of handcrafted component behavior implementations from component generator concerns by providing the single means of communication between component artifacts and component behavior artifacts. This liberates application programmers from learning about component generation details (such as the correct use of ports) and shields the component internals from application programmers. At run-time, the data structure implementing `Input` conveys the current values of incoming ports and variables and `Result` conveys the resulting values after component behavior calculation.

In conclusion, the component generator of this language family produces Java classes that extend the class `Component` and employ `Port` and `Variable` to represent ports and variables, respectively. Furthermore, atomic components produced with this component generator yield a member that implements `Computable`. For components with behavior models, the family's behavior generators produce Java classes implementing this interface and conforming either to naming conventions of following the containing component's implementation reference (cf. Section 4.1.1). For atomic components without behavior models, application programmers provide according Java classes implementing the `Computable` interface.

The `javats` Component Code Generator

The component code generator `ComponentsJavaTS` translates composed and atomic components with their ports, variables, subcomponents, and embedded behavior models into Java classes. For the latter, it delegates generation the available behavior generators. As the component generator must produce artifacts compatible to the `javats` run-time environment, it produces up to four Java classes per component:

1. A component implementation class that extends the abstract class `Component` and owns multiple ports and variables according to the corresponding model. If the model is an atomic component, this class also has a member of type `Computable`, otherwise, it contains the references to the model's subcomponents.
2. For atomic component models, it also generates a class implementing `Input` wrapping component's input ports and variables with their current values. Instances of it are passed to the component's behavior implementation which performs calculations based on these values.
3. A similar class for output ports and variables implementing `Result`. This also is generated for atomic components only.
4. A class implementing the factory pattern [GHJV95] for the component implementation class. Component implementations of composed components rely on this class to create subcomponent instances. Employing this pattern allows exchanging the actually created factory products for testing purposes.

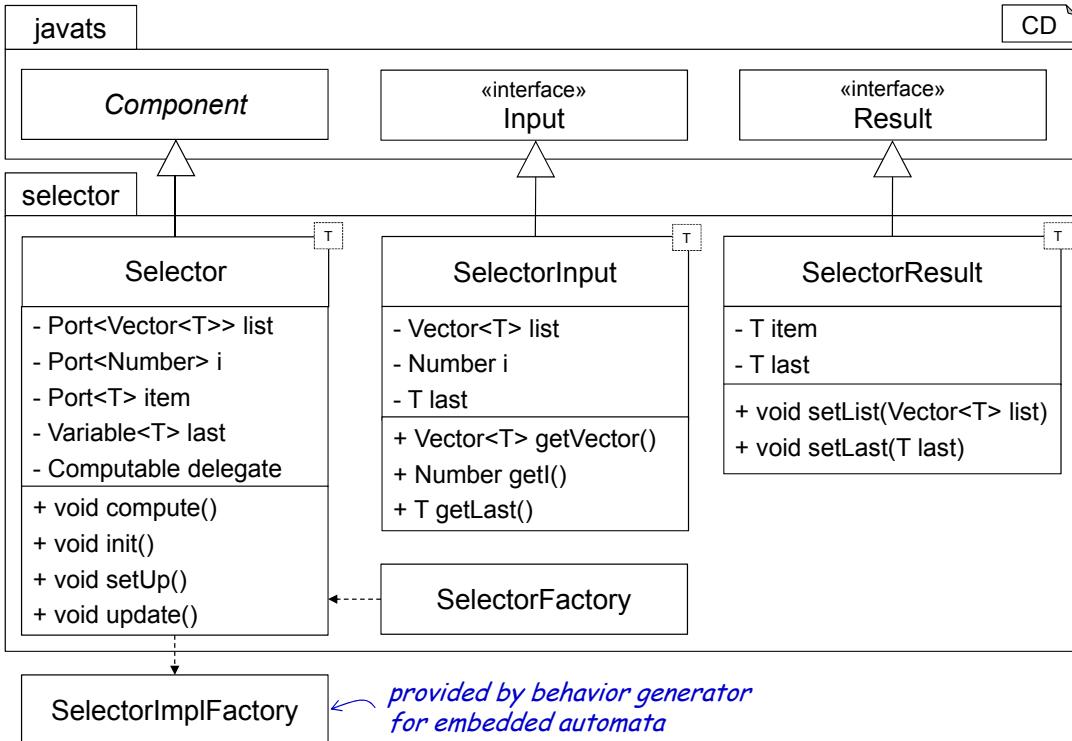


Figure 7.11: Java classes resulting from applying component generator ComponentsJavaTS to component Selector.

For the component type `Selector` of Figure 7.9, the generator produces the four classes as depicted in Figure 7.11. To achieve this, the component generator employs 29 templates and 9 template helpers. Most templates are responsible for small parts of the resulting class to enable reuse with other code generators (as discussed in [RRW13b]). For instance, there are individual templates for each of the methods illustrated in Figure 7.11 for atomic components and composed components. Furthermore, there are templates for connectors, parameters, ports, and variables. The helpers, for instance, collect various component constituents into data structures convenient for processing by templates.

The generated class `Selector` of Figure 7.11, for instance, represents the component type `Selector` and can be reused for every subcomponent instance of this type (cf. Req. *TRQ-8*). It has a member of type `Computable` to which the component implementation delegates `compute()` calls. To instantiate this member, component implementations rely on factories provided by behavior generators, such as the class `SelectorImplFactory`. Its name is derived from the behavior implementations name and passed to the behavior generator via its generator description implementation. The behavior generator responsible for translating automata models is hence also responsible for providing a proper factory. This requires the component generator's integration into MontiArcAutomaton's generator composition. Therefore, the component generator's de-

scription model is defined as depicted in Listing 7.17. The generator description resides in the generators root package (l. 1) and has the name ComponentsJavaTS (l. 3). The generator description declares that its description implementation implements the interface ComponentGenerator (l. 4) and that its start template is templates.Main (l. 5). The represented generator also processes the language defined by MAAComponent of grammar MontiArcAutomaton and produces artifacts compatible to run-time environment javats.

```
1 package d.m.generators.componentsjava;
2
3 generator ComponentsJavaTS
4   conforms d.m.c.i.ComponentGenerator {
5     start d.m.generators.componentsjava.templates.Main;
6     language d.m.c.l.adl.MontiArcAutomaton.MAAComponent;
7     rte de.montiarcautomaton.runtimes.javats;
8 }
```

GD

Listing 7.17: Generator description of the ComponentsJavaTS component generator with abbreviated package names.

The **javats** Automata Code Generator

The AutomataJavaTS code generator provides means to translate AUTOMATA models embedded in components into Java classes compatible to the run-time environment javats. The generator produces three artifacts per AUTOMATA model:

1. An enumeration to represent the automaton's states.
2. The actual behavior implementation realizing the automaton. This class implements the BehaviorGenerator interface and relies on the surrounding component's Input and Result data structures (cf. Figure 7.11). These are wrappers to receive current port and variable values (Input) and to emit new values for ports and variables (Result). This liberates component behavior generator developers and application programmers from these structure implementation details.
3. A factory [GHJV95] for the behavior implementation class.

For the component type Selector of Figure 7.9, AutomataTS produces the enumeration type SelectorState with values Idle and Error, the behavior implementation class SelectorImpl, which implements the interface Computable, and the behavior factory SelectorImplFactory as used by the class Selector. Figure 7.12 illustrates these classes and shows that SelectorImpl relies on the data structures SelectorInput and SelectorResult provided by the component generator. Furthermore, it shows that the behavior artifacts are created in the same package as the other component artifacts. This decision is made within the component generator and

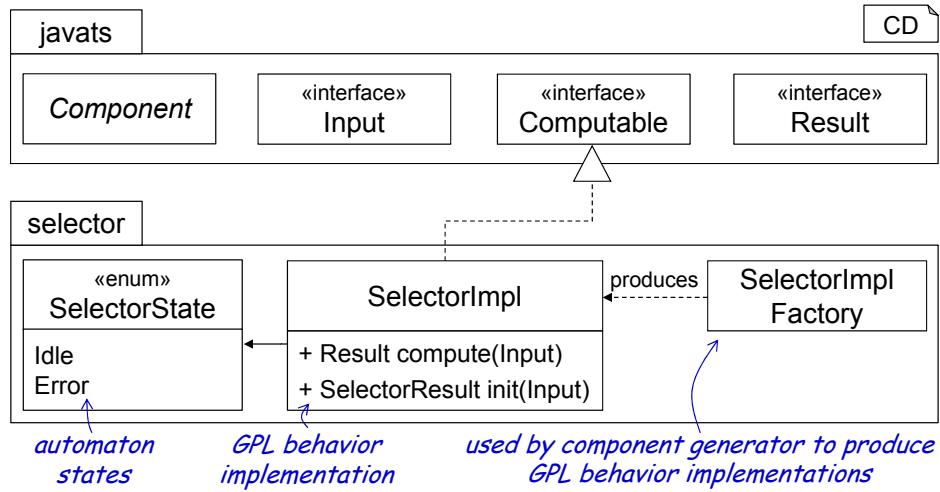


Figure 7.12: The AutomataJavaTS behavior generator produces three classes per embedded AUTOMATA model.

passed to AutomataJavaTS via its generator interface (cf. Figure 7.2). This decision is arbitrary and extending the behavior generator interface to receive a package path would enable the behavior generators to produce artifacts in different package paths.

AutomataJavaTS translates AUTOMATA models that allow underspecification, for instance in form of defining multiple initial states to Java classes, multiple transitions leaving a state on the same stimulus, or multiple transitions leaving a state on the same guard expression. However, Java does not support such non-determinism and therefore some resulting behavior artifacts will not adhere to the behavior of the input model. To prevent this, AutomataJavaTS provides context conditions to reject problematic models. While these can identify AUTOMATA models with multiple states and identical stimuli on the same state easily, evaluating whether two guards are enabled for the same inputs is generally undecidable. This, however, is a problem of verification and beyond the scope of this thesis.

Nevertheless, AutomataJavaTS must explicate its context conditions to MontiArc-Automaton. Therefore, its generator description must declare these as introduced in Section 7.2.1. The description of AutomataJavaTS is as depicted in Listing 7.18, which, after declaring a package (l. 1) and its name (l. 3), denotes that AutomataJavaTS is a behavior generator (l. 4). Consequently, it must declare a start template (l. 5), the modeling language it processes (l. 6), and compatibility to a run-time environment (l. 7) – in this case javats. Additionally, it declares the context conditions `MultipleInitialStates` and `MultipleIdenticalStimuliInState` (ll. 8-11).

```
1 package d.m.generators.automatajava;
2
3 generator AutomataJavaTS
4   conforms d.m.c.i.BehaviorGenerator {
5     start d.m.generators.automatajava.EmbeddedAutomaton;
6     language d.m.c.l.automata.Automata.AutomatonContent;
7     rte de.montiarautomaton.runtimes.javats;
8     contextconditions in d.m.g.a.cocos {
9       MultipleInitialStates,
10      MulitpleIdenticalStimuliInState
11    }
12 }
```

GD

Listing 7.18: Generator description of behavior generator AutomataJavaTS with abbreviated package names.

The javats Data Type Code Generator

For the translation of UML/P class diagrams into Java artifacts, the language family employs the code generator introduced in [Sch12]. This code generator is template-based and utilizes FreeMarker as well. However, it does not provide a generator description and hence is not composable. To amend this, the ClassdiagramJava generator wraps that generator's functionality and adds the description model depicted in Listing 7.19.

```
1 package de.montiarautomaton.generators.cdjava;
2
3 generator ClassdiagramJava
4   conforms d.m.c.i.DataTypeGenerator {
5     start mc.umlpa.arc.ClassCodegen;
6 }
```

GD

Listing 7.19: A composable class diagram generator based on the generator presented in [Sch12].

The ClassdiagramJava generator description consequently refers to the wrapped generator's start template mc.umlpa.arc.ClassCodegen instead of providing its own. The generator's translation of class diagrams to Java and the integration of handcrafted method implementations is described in [Sch12].

7.4.2 A Code Generator Family for ROS Python Systems

The pythonts code generator family produces Python artifacts relying on the robot operating system (ROS) [QGC⁺09] as middleware. To this effect, it employs concepts similar to the Java code generator family. As the Python component generator takes care of translating embedded AUTOMATA models as well, a behavior code generator is

not necessary. Also, as Python differs from Java, the Python implementations differ accordingly. The most important differences are the lack of interfaces in Python and that the language is dynamically typed. Consequently, the members produced for configuration parameters, ports, and variables are untyped in resulting Python artifacts. Furthermore, ROS [QGC⁺09] considers software systems as graphs of nodes communicating over typed message buses. These nodes especially do not support hierarchical decomposition, wherefore this generator translates composed components accordingly.

The next sections introduce ROS, describe the family's run-time environment, its component code generator and data type generator.

ROS

ROS [QGC⁺09] is a communication middleware and build system for distributed robotics applications multiple GPLs, including C++ and Python. ROS applications consist of nodes and topics. Nodes are stand-alone processes that perform computations, may not be hierarchically composed, and communicate via topics. Topics are ordered, typed message buses. Nodes can register subscribers to get informed about new messages on topics and publishers to send message to topics. Such communication can be established dynamically, which allows multiple nodes to send messages to a single topic. ROS expects topic types to be defined in msg⁴ models. These models resemble simplified UML class diagrams (or C++ struct types) without visibilities, methods, and inheritance. Besides these, ROS project require so-called launch files⁵, describing which nodes to start, and manifests characterizing the ROS application. For instance, the data type MotorCMD of Figure 2.8 can be translated into a msg model as depicted in Listing 7.20.

```

1 # MotorCMD message format.
2 # Sending component
3 string sender
4 # Enumeration constants
5 uint8 FORWARD
6 uint8 BACKWARD
7 uint8 STOP
8 # Data type enumeration
9 uint8 data

```

ROS msg

Listing 7.20: The data type MotorCMD of Figure 2.8 as a ROS msg.

Please note that this msg model declares the field `sender`, which identifies the sending subcomponent. As ROS allows that multiple nodes send messages to a topic, but Monti-ArcAutomaton allows only a single sender per incoming port, this is used to ensure component implementations deny messages from wrong senders. However, this cannot prohibit malevolent ROS nodes from faking sender information and publishing messages

⁴rosmsg website: <http://wiki.ros.org/rosmsg>

⁵roslaunch website: <http://wiki.ros.org/roslaunch>

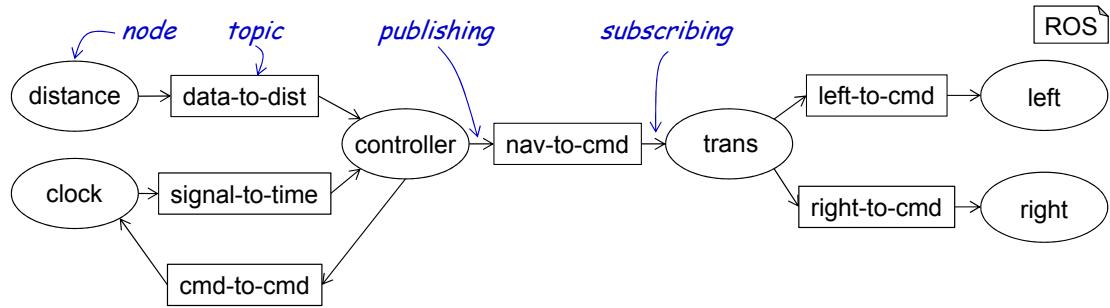


Figure 7.13: A ROS graph representing the `ImprovedBumperBot` software architecture of Figure 4.1. Atomic components are translated to nodes, composed components to configuration information stored in launch files (not depicted), and connectors to topics.

to topics resembling connectors. Also ROS msg models are strongly typed but do not support generics. The data type generator of this family takes care of proper translation.

Figure 7.13 illustrates the ROS graph for the `ImprovedBumperBot` software architecture of Figure 4.1. Here, each atomic component is represented by a node and each connector by a topic of the corresponding ROS msg data type. The composed components `ImprovedBumperBot` and `Navigation` are absent and have been translated into a launch file starting their subcomponents' processes.

The issues in translating MontiArcAutomaton ADL models into ROS systems therefore originate in employing Python's type system and the lack of node decomposition. Also allowing arbitrary many-to-many communication and demanding topic types in ROS msg models must be considered. Furthermore, producing proper configuration files and starting subcomponents as individual processes is crucial.

The following sections explain the code generation from MontiArcAutomaton to ROS, the interaction of nodes and components, as well as the translation of architectures with embedded automata into multiple artifacts including configuration files.

Run-Time Environment

The `pythonts` run-time environment of the ROS-Python code generator family resembles the `javats` run-time environment in employing similar concepts. It also features classes to describe components, ports, and variables. Python component implementations also integrate GPL behavior implementations via delegation and hence also rely on classes to represent these with their input and output. Ports and variables also extend a common base class. The run-time environment itself is unaware of ROS as the component implementations are executed by manager classes that start ROS nodes, connect to topics, and transmit data to and from ports.

The constituents of `pythonts` are the seven classes depicted in Figure 7.10. As with `javats`, there is the abstract base class `Component`, which defines methods all component implementations have in common. This besides the methods reflecting the

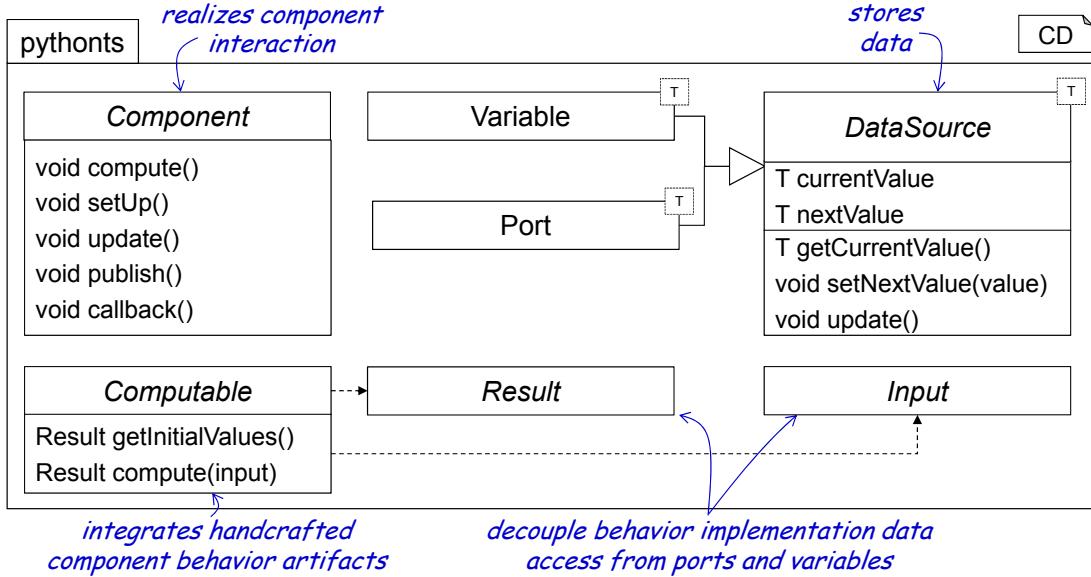


Figure 7.14: The run-time environment `pythonts` supports execution of Python components that communicate via ROS.

component life-cycle, implementations of this class also must define the two methods `callback()` and `publish()` which connect messaging to the ROS infrastructure. The run-time environment also contains the two classes `Port` and `Variable`, which inherit from the abstract class `DataSource` to provide access to ports, variables, and their values. Although `DataSource` conceptually relies on the generic type parameter `T`, its implementation does not. As the types of Python members are determined dynamically, it relies on duck typing [Bea09] to determine data types. Thus it omits syntax to handle generic types. With `pythonts`, GPL behavior implementations are integrated via delegation [GHJV95] as well. Hence, it also contains the three abstract classes `Computable`, `Input`, and `Output` (Python does not support interfaces). For the same reasons as with `javats`, the Python `Component` class is oblivious of port and variable implementation details.

This run-time environment prescribes component life-cycles, their interaction, and integration of GPL behavior implementations similar to `javats`. Employing the full power of ROS, distributed component implementation processes exchanges messages via topics. Hence, their distribution is transparent to the user. To enable this, this code generator family must eliminate composed components, produce proper ROS msg models for port data types, start ROS nodes, and connect these to the ROS infrastructure. The next section presents the family's component generator, which takes care of this.

The `pythonts` Component Code Generator

The `pythonts` component code generator produces Python artifacts and ROS configuration artifacts implementing component structure and automata behavior using FreeMarker. To this effect, it first eliminates composed components from the architecture and lifts their subcomponents to the next higher level until all subcomponents are direct subcomponents of the architecture root. For instance, regarding the software architecture `ImprovedBumperBot` of Figure 4.1, the subcomponents `trans`, `left`, and `right` become subcomponents of `ImprovedBumperBot`. Once all subcomponents reside in the architecture root, it is eliminated as well. The resulting software architecture is a graph of components for which the generators produces component implementations and further ROS artifacts.

Similar to the `javats` component generator, this generator produces at least five classes per component type (Figure 7.15). The component implementation extends from `Component` of the run-time environment and comprises members representing ports and variables. It interfaces handcrafted or generated GPL behavior implementations via delegation to an instance of `Computable`. Therefore, each time a message on the topics it is connected to arrives, these value are stored in its incoming ports and variables. Every `compute()` step, it compiles current port and variable data into an instance of `SelectorInput` and passes it to the behavior implementation of `SelectorImpl`. The behavior implementation computes one transition in its automaton implementation, compiles the resulting port and variable values into an instance of `SelectorResult`, and passes that back to the component implementation. On receiving this data, `Selector` assigns the contained values to its ports and variables and finishes computation. Every `update()` step, it publishes the results to ROS. Receiving data from a topic via `callback()` and sending data to a topic via `publish()` includes marshalling the data into ROS msg types. As this is part of the generated component implementation, application programmers do not have to deal with ROS msg data types. The method `callback()` also takes care of denying messages from senders not connected to the component's ports. The component generator also produces a manager to create, execute, and manage component instances of type `Selector`. Its main purpose is creation of a component instance and managing its life-cycle.

After producing Python artifacts, the component generator also generates the launch file referencing the component managers to start the respective components' nodes. Furthermore, it produces the build configuration files (`makefile`, `CMakeLists`, and `manifest`) required by the `rosbuild`⁶ infrastructure to create ROS packages.

The code generator description of this generator (Listing 7.21) consequently conforms to the component generator interface (l. 4). It also declares a start template (l. 5), to process models of the MontiArcAutomaton ADL language (l. 6), and to produce artifacts conforming to the `pythonts` run-time environment (l. 7). As this generator produces artifacts for embedded automata models as well, it declares the additional input modeling language `AutomatonContent` (ll. 9-11). Consequently, it employs the

⁶rosbuild website: <http://wiki.ros.org/rosbuild>

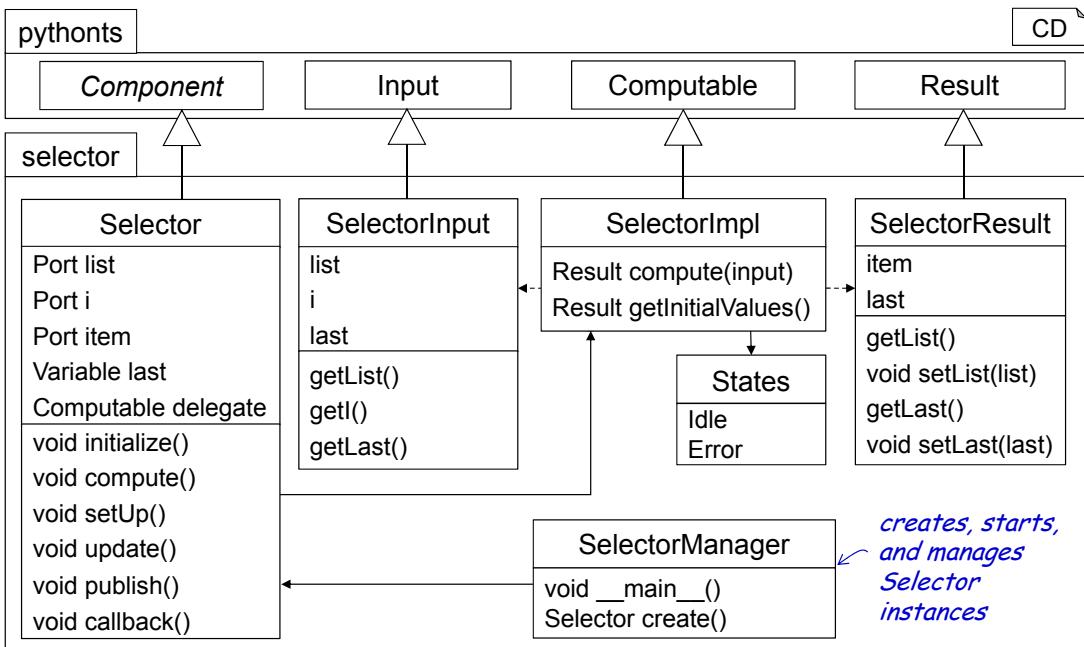


Figure 7.15: The `pythonts` component generator produces five classes per component without behavior model (atomic or composed).

same context conditions as the AutomataTS generator regarding multiple initial states and identical stimuli (ll. 13-16).

The `pythonts` Data Type Code Generator

The `pythonts` data type generator translates UML/P class diagram types into two representations: First, it creates Python data types for the component-internal representation. Second, it also creates ROS msg data types for the communication between nodes. The translation of UML/P class diagrams into plain Python data types translates CD classes into Python classes under consideration of Python idioms: there are no visibilities but conventions, and Python does not support interface types, and method overloading is prohibited as well. The code generator translates UML/P classes into similar Python representations, i.e., visibilities are translated into member names according to typical Python conventions, interfaces are translated into abstract classes (as Python supports multiple inheritance), and method overloading is prohibited by a generator context condition. The part of this generator translating to plain Python was prepared in a Bachelor's thesis [Deu14]. Translation into ROS msg models is slightly more complex as both, UML/P CD and ROS msg are strongly typed, but ROS msg (cf. Listing 7.20) does neither support generics, nor interfaces, inheritance, generic types, static members, or methods. The lack of interfaces and methods is unproblematic as component implementations take care of marshalling ROS msg types into plain Python types sup-

```
1 package d.m.generators.componentsros;
2
3 generator ComponentROS
4   conforms d.m.c.i.ComponentGenerator {
5     start d.m.g.c.MainTemplate;
6     language d.m.c.l.adl.MontiArcAutomaton.MAAComponent;
7     rte de.montiarcautomaton.runtimes.pythonts;
8
9   behaviors {
10     d.m.c.l.automata.Automata.AutomatonContent;
11   }
12
13   contextconditions in d.m.g.c.cocos {
14     MultipleInitialStates,
15     MulitpleIdenticalStimuliInState;
16   }
17 }
```

GD

Listing 7.21: Description of a composable component generator that produces Python implementations interfacing the ROS middleware.

porting both. Inheritance, and static members are prohibited by context conditions. For UML/P classes yielding generic type parameters, the data type generator calculates the actual types of members apriori and produces corresponding ROS msg types with the generic types replaced. Thus, for a component instantiating two subcomponents of type Selector with different arguments for its generic type parameter, this generator produces two different ROS msg types to communicate with these instances.

```
1 package d.m.generators.rospythontypes;
2
3 generator DataTypesROS
4   conforms d.m.c.i.DataTypeGenerator {
5     start d.m.g.rospythontypes.Main;
6
7     contextconditions in d.m.g.rospythontypes.cocos {
8       NoMethodOverloading,
9       NoInheritance,
10      NoStaticMembers
11    }
12 }
```

GD

Listing 7.22: A composable data types generator for the production of Python classes and ROS msg types.

DataTypesROS code generator description of Listing 7.22 represents this data type generator (l. 4) for use with MontiArcAutomaton. To this effect, it declares a start

template (l. 5), and multiple context conditions (ll. 7-11) regarding method overloading, interfaces, inheritance, and static members to realize the discussed restrictions.

7.5 Discussion and Related Work

The assumption of these three specific generator kinds is tightly coupled to the C&C nature of MontiArcAutomaton and cannot be generalized to code generator composition challenges in different contexts. Approaches to software composition that could be applied to code generators are, for instance, GenVoca [BO92, BST⁺94] and the Genesys [Jö13] extension of the application building center [SMN⁺07] as well as generator composition via aspect-oriented programming [ZR11], or feature-driven MDE [TBD07]. Their relation to the MontiArcAutomaton approach has been discussed in [RRRW14]: being generic, these approaches are agnostic to properties of the composed generators. As such they introduce additional complexities to code generator development which can be reduced taking the C&C nature of MontiArcAutomaton its generators into account. While it may be useful to add other generator kinds - for instance a code generator kind for object instantiation using the factory pattern [GHJV95] has been introduced in [RRRW14] - these are not required to separate of MontiArcAutomaton concerns.

Code generator composition in MontiArcAutomaton relies on exploiting the generator description implementations generated from the generator description models that implement interfaces according to the three generator kinds. Hence, the validity of a specific generator description model depends on the interface the represented generator should implement. In consequence, changes to generator interfaces entail changes to the generator description modeling language and their context conditions as well. This derivation could be automated by parsing the generator interfaces using the Java/P and generating a proper grammar according to naming and data type conventions (such as lists in the interfaces translated to lists in the grammar).

Furthermore, generator composition also relies on artifact compatibility as expressed in terms on run-time environments. Such run-time environments may comprise multiple classes and artifacts for different purposes. Hence, developing a code generator that produces conforming artifacts requires an understanding how this conformance is achieved (for instance by producing artifacts implementing specific interfaces). Introducing a modeling language to describe such dependencies may reduce the need for such coordination. Whether it can be eliminated completely needs to be investigated.

The actual execution of behavior generators is performed by the component generators. This imposes requirements on the development of compositional MontiArcAutomaton code generators. As component generators already must translate all component model parts besides any embedded behavior models, they must traverse and process almost the complete AST anyway (only the subtree containing the behavior model is not processed by the component generator). Although the generator orchestrator could traverse the AST itself - and call either the component generator or an appropriate behavior generator - this would require a very broad component generator interface and complicate component generator development.

Development of the `pythonts` generator family that interfaces the ROS [QGC⁺⁰⁹] middleware via ROS msg data types has shown that using class diagrams as data type modeling language may pose issues when translating to less expressive formalisms. Multi-platform applications requiring such translation must either restrict data types to the minimum expressiveness available on all platforms or employ code generators supporting proper data type translation.

The code generators presented in this chapter produce GPL implementations base on component types. Although this helps to reduce the number of generated artifacts, depending on the target GPL, generation based on component instances might be easier to implement. As MontiArcAutomaton does not prescribe how generators work - besides from employing certain generator kinds - this decision is left to the generator developers

Chapter 8

Describing Component & Connector Applications

All models are wrong but some are useful.

GEORGE E. P. BOX

Employing software architectures (Chapter 4), bindings (Chapter 6), and code generators (Chapter 7) requires their integration. Bindings and code generators are both platform-specific but also interdependent: Bindings map platform-independent interface components to platform-specific components, which are tied to GPL implementations conforming to a specific run-time environment. Code generators produce GPL artifacts from platform-independent components that conform to a run-time environment as well. Hence, specifying bindings or generators within the software architecture models ultimately ties it to platforms compatible to the run-time environments referenced by bound components or selected code generators. Instead, we decouple the platform-independent software architecture to be (re-)used from code generators and bindings by defining both in application configuration models. This loose coupling between an application's constituents allows reusing code generators and architecture models in different applications without modification. Platform-independent architectures, implementation libraries, and selected code generators can be referenced by multiple applications and in different combinations. Figure 8.1 depicts the typical constituents of a MontiArcAutomaton application. MontiArcAutomaton applications are characterized by *application configuration* models that reference the platform-independent software architecture, import types of implementation libraries, describe how the interface components of the referenced architecture are bound to implementation library components (cf. Chapter 6), and define which code generators should be applied to the resulting architecture (cf. Chapter 7). Thus, the MontiArcAutomaton language family comprises the MontiArcAutomaton ADL, the generator description language, and the application configuration language.

Therefore, Section 8.1 presents the MontiArcAutomaton application configuration language with its language elements, symbol table, and context conditions. Afterwards, Section 8.2 describes how MontiArcAutomaton processes MontiArcAutomaton applications before Section 8.3 describes their usage by example. Finally, Section 8.4 discusses findings and related work.

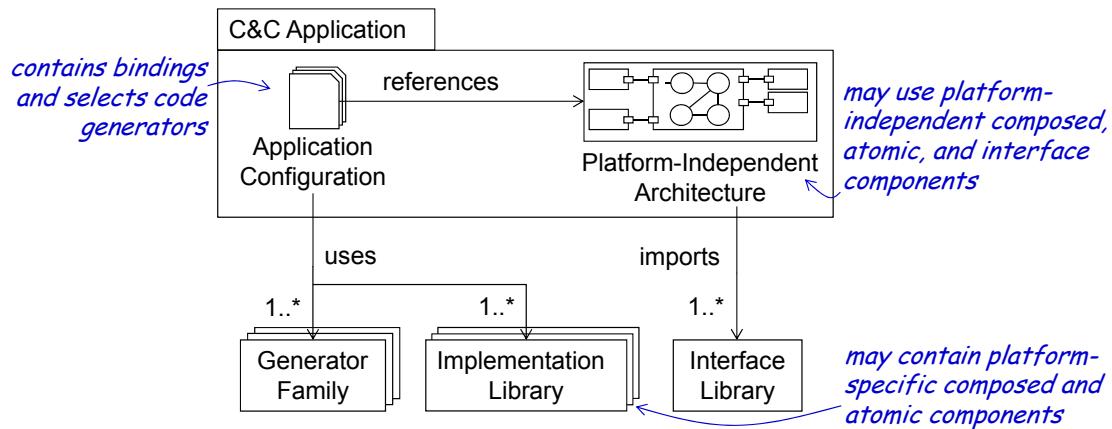


Figure 8.1: The constituents of a typical MontiArcAutomaton application are multiple application configuration models referencing a single platform-independent architecture and multiple generator families as well as implementation libraries to translate the architecture into multiple platform-specific implementations.

8.1 Application Configuration Language

The application configuration modeling language combines a single software architecture with bindings and code generators for a single target platform. Therefore, a software development project employing MontiArcAutomaton may contain multiple application configuration models. The following sections present the language's modeling elements, symbol table, and context conditions.

8.1.1 Language Elements

Application configuration models have a name, reference a software architecture, contain a set of bindings, and select a set of code generators. The application configuration modeling language defines language elements to represent this information properly. Listing 8.1 shows the application configuration `NavigationBotNXTJava` for the `NavigationBot` software architecture depicted in Figure 6.2. The model first declares a package (l. 1) and subsequently imports the `NXTJava` component types and data types depicted in Figure 6.8 (l. 3).

The application configuration begins with the keyword `application`, followed by a name, the keyword `for`, the qualified name of the architecture this application configuration refers to, and application configuration's body in curly brackets (ll. 5-6). The body may contain bindings and references to code generators in arbitrary order. The application configuration of Listing 8.1 first contains three bindings (ll. 8-10) followed by four qualified names referencing code generators (ll. 12-15). The bindings reference the subcomponent instance names of the interface subcomponents of `NavigationBot` and describe how their types should be replaced. The remainder of this section introduces

the language elements for application configuration declaration, definition of bindings, and selection of code generators in detail.

```

1 package navigationbot;
2
3 import NXTJava.*;
4
5 application NavigationBotNXTJava
6   for architecture.NavigationBot {
7
8   bind sensor to NXTUltrasonic(SensorPort.PORT_1);
9   bind navcontrol.left to NXTMotor(MotorPort.PORT_A);
10  bind navcontrol.right to NXTMotor(MotorPort.PORT_B);
11
12 componentgenerators.ComponentsJava;
13 datatypegenerators.ClassdiagramJava;
14 behaviorgenerators.AutomataJava;
15 behaviorgenerators.IOTableJava;
16 }
```

App

Listing 8.1: The application configuration `NavigationBotNXTJava` references the `NavigationBot` software architecture and defines three bindings.

Application Configuration Declaration

MontiArcAutomaton supports multiple application configurations per project, hence, it must be able to distinguish these. The application configuration declaration provides means to do so by requiring a name. It begins with the keyword `application` followed by the application's unqualified name as illustrated in Listing 8.2.

```

1 application BumperBot
2   for architecture.NavigationBot {
3   // ...
4 }
```

App

Listing 8.2: The application configuration `BumperBot` (l. 1) references the software architecture `architecture.NavigationBot` (l. 2).

In conjunction with the package, its name identifies the application configuration model (l. 1). After the name, the keyword `for` declares which software architecture model this application configuration relates to – restricting it to a single architecture (l. 2). The architecture's name must either be fully qualified or imported. Curly brackets follow the architecture's name to delimit the application configuration's body.

Bindings

Bindings describe a mapping from a subcomponent path in the transitive subcomponent hierarchy of the referenced software architecture denoted by $s_0.s_1 \dots s_n$ to a parametrized component type. Here, $s_0.s_1 \dots s_n$ is a dot-separated sequence of subcomponent names that unambiguously identifies a subcomponent in the subcomponent hierarchy of the referenced architecture (cf. Section 2.4). The identified subcomponent must be an interface component to be replaced. Forfeiting this restriction would allow replacing arbitrary components and, hence, would allow to change the software architecture beyond comprehension. Whether this is more useful than harmful requires investigation. T is a component that extends the interface component of s_n and a_0, \dots, a_m are proper arguments for T (cf. Section 6.1). Application configuration models specify such bindings in the form

```
bind s0.s1...sn to T(a0,...,am)
```

The application configuration BoundBumperBot depicted in Listing 8.3 defines three bindings (ll. 4-6). Each begins with the keyword bind, followed by a dot-separated sequence of names indicating a path in the subcomponent hierarchy of NavigationBot. After this, the keyword to, and the parametrized component type to replace the interface subcomponent follow.

```
1 application BoundBumperBot
2   for architecture.NavigationBot {
3
4     bind sensor to ROSUltrasonic("ultra", "front_us");
5     bind navcontrol.left to ROSMotor("lm", "motor_1");
6     bind navcontrol.right to ROSMotor("rm", "motor_2");
7     // ...
8 }
```

App

Listing 8.3: The application configuration BoundBumperBot defines three bindings (ll. 4-6) from interface subcomponents of the component type architecture.NavigationBot to platform-specific components.

The bindings of BoundBumperBot describe how the interface subcomponents sensor of NavigationBot and left and right of component type NavigationControl (cf. Figure 6.2) should be replaced. As NavigationBot instantiates the component type NavigationControl as navcontrol, the qualified name of the latter two bindings begins with navcontrol accordingly. Each binding's right-hand side refers to a component type of the PythonROS implementation library (cf. Figure 6.9) and parametrizes if correctly. The bound component types may be imported and referenced by their unqualified name or their qualified name if not imported. Bindings may provide arguments for parameters not set by the software architecture only. This prohibits for bindings to overwrite architecture properties as required by Req. *MRQ-8.2*.

Code Generators

Application configuration models also select the code generators to apply. To this effect, application configuration models may contain a set of qualified code generator names. In contrast to an earlier version of this language presented in [RRRW14], there are no keywords for the selection of individual generator kinds. Instead, the context conditions take care, that the selection is valid and MontiArcAutomaton derives the referenced generator's types via their symbol table entries (cf. Section 7.2.2).

```

1 application BoundBumperBotPythonROS
2   for architecture.NavigationBot {
3
4     bind sensor to ROSUltrasonic("ultra", "front_us");
5     bind navcontrol.left to ROSMotor("lm", "motor_1");
6     bind navcontrol.right to ROSMotor("rm", "motor_2");
7
8     componentgenerators.ROSPythonComponents;
9     datatypegenerators.PythonCD;
10    behaviorgenerators.AutomataPython;
11    behaviorgenerators.IOTablePython;
12 }
```

App

Listing 8.4: Application configuration `BoundBumperBotPythonROS` selects four code generators for translation of components (l. 8), class diagrams (l. 9), and two behavior languages (ll. 10-11).

The application configuration `BoundBumperBotPythonROS` of Listing 8.4 selects four code generators by their qualified names (ll. 8-11). These names refer to the generators' configuration models (cf. Section 7.2), which provide all information to determine whether this selection is valid.

8.1.2 Symbol Table

The application configuration language symbol table defines a single entry type: the `ApplicationConfigurationEntry`. It comprises all information required to check the well-formedness of application configuration models: the `MAAComponentEntry` of the referenced architecture (cf. Figure 4.3), the `GeneratorDescriptionEntry` symbol table entries (cf. Figure 7.4) of the participating generators, and a map from subcomponent paths to subcomponent entries that represents bindings. This integration is enabled by the language aggregation of MontiCore and allows reasoning about models of other languages (i.e., components and code generator descriptions) from the perspective of a MontiArcAutomaton application.

Figure 8.2 depicts the application configuration symbol table entries and their relations. Note that this structure ties the application configuration symbol table and the application configuration language to the symbol table entries of the MontiArcAutomaton

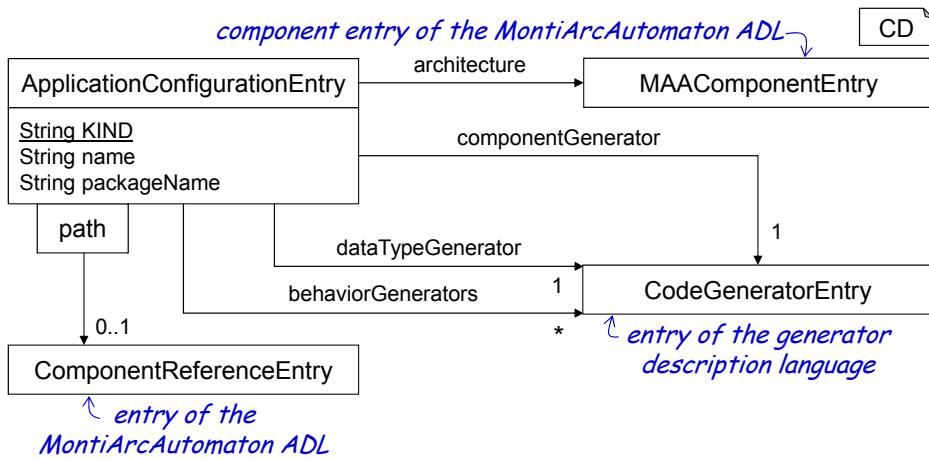


Figure 8.2: The symbol table of the application configuration language references entries of the MontiArcAutomaton ADL and of the generator description language.

ADL and the code generator description language. However, as long as their symbol table entries remain unchanged, changes to their respective abstract syntax trees do not affect the application configuration language.

8.1.3 Context Conditions

The application configuration context conditions ensure the models' well-formedness by checking whether the referenced architecture, subcomponents, and code generators exist, and whether the code generator selection and bindings are valid. To this effect, the application configuration context conditions are separated into uniqueness conditions, convention conditions, referential integrity conditions, and type correctness conditions. The following sections describe these context conditions. Some of the context conditions operate in the context of the application configuration's software architecture. For the following examples, we assume the ExplorerBot architecture illustrated in Figure 3.2.

Uniqueness Conditions

The application configuration uniqueness conditions check uniqueness of bindings and participating code generators. Restrictions on code generators are derived from the nature of embedding behavior languages into components: there may be only a single data type generator, a single component code generator per application, and multiple behavior generators. The structure of the application configuration's grammar already restricts application configuration's to reference a single software architecture, hence, this requires no context condition. Violation of uniqueness conditions raises errors.

CU1: All subcomponents are bound unambiguously.

Defining two bindings to the same subcomponent introduces underspecification that must be resolved prior to code generation. Such underspecification is unsolicited and processing such an application configuration creates unexpected results as the order of bindings is not intended to imply an order of their application. Hence, binding the same subcomponent of the architecture's *subcomponent hierarchy* multiple times is prohibited.

The application configuration BumperBotNXTJava of Listing 8.5 violates this condition by binding the subcomponent `sensor` multiple times (ll. 4-8). It specifies that the type of `sensor` should be replaced with `NXTUltraSonic` (ll. 4-5) and with `NXTColor` (ll. 7-8), which is impossible in the target software architecture.

```

1 application BumperBotNXTJava
2   for arc.ExplorerBot {
3
4     bind sensor ⑥ // Component 'sensor' bound multiple times.
5       to NXTUltraSonic(PORT_1);
6
7     bind sensor ⑥ // Component 'sensor' bound multiple times.
8       to NXTColor(PORT_2);
9     // ...
10 }
```

App

Listing 8.5: The `BumperBotNXTJava` application configuration binds the subcomponent `sensor` twice (ll. 4-8).

CU2: Exactly one code generator for component structure.

Each application describes transformation and code generation for exactly one software architecture and one target platform. Hence, each application configuration may reference only a single component code generator. Although referencing, and thus executing, two component generators might not raise issues, it is unspecified whether executing the second code generator interferes with (artifacts produced by) the first generator. Therefore, this context condition prohibits such configurations.

In case the generators `CompPythonV272` (l. 4) and `CompPythonV330` (l. 5) as referenced in Listing 8.6 are valid component generators, i.e., their generator descriptions are valid and declare that the represented generators implement the interface `ComponentGenerator`, the application configuration `BumperBotROSPython` violates this context condition. This context condition is checked against the application configuration's AST, as its symbol table does not support multiple component generators (cf. Section 8.1.2). Generator names are resolved as `GeneratorDescriptionEntry` instances (cf. Section 7.2.2). If two or more symbol table entries claim to contribute a component generator, this context condition raises the error depicted in Listing 8.6.

```
1 application BumperBotROSPython
2   for arc.ExplorerBot {
3
4     CompPythonV272; ❌ // Multiple component generators.
5     CompPythonV330; ❌ // Multiple component generators.
6     // ...
7 }
```



Listing 8.6: This application configuration is invalid as it contains multiple code generators responsible for the same language fragment (ll. 4-5).

CU3: Exactly one code generator for data types.

For the same reasons application configuration models may reference only a single component generator, they may reference a single data type generator only. The procedure to check whether the generator selection contains two or more data type generators also checks the models' AST and resolves all generators to find multiple generator models referencing data type generators. In case the context condition finds such generators, it raises the errors depicted in Listing 8.7, where CDJava (l. 4) and CDPython (l. 5) are both data type generators.

```
1 application BumperBotROSNative
2   for arc.ExplorerBot {
3
4     dtg.CDJava; ❌ // Multiple data type generators.
5     dtg.CDPython; ❌ // Multiple data type generators.
6     // ...
7 }
```



Listing 8.7: The application configuration BumperBotROSNative contains two code generators responsible for transformation of data types (ll. 4-5) and, hence, is invalid.

CU4: Exactly one code generator per behavior language.

The MontiArcAutomaton code generator composition also requires that there is exactly one code generator per behavior language embedded in the referenced software architecture. To check this, the context condition resolves all components of the architecture and collects their behavior languages. If there is any behavior language for which no code generator is responsible, it raises the error illustrated with application configuration ExpJava depicted in Listing 8.8. This application configuration references the architecture ExplorerBot as depicted in Figure 3.2 on page 32, which employs embedded AUTOMATA models. As the application configuration does not provide a code generator

responsible for the embedded AUTOMATA grammar production `Automaton.Body`, it raises the depicted error (ll. 16-17).

```

1 generator ADComponentsJava
2   implements ComponentGenerator {
3     start ComponentMain;
4     language MontiArcAutomaton.MAACComponent;
5     runtime javats;
6     behaviors ActivityDiagram.Body;
7   }
8
9 generator ActivityDiagramsJava
10  implements BehaviorGenerator {
11    start ActivityDiagram;
12    language ActivityDiagram.Body;
13    runtime javats;
14  }
15
16 application ExpJava for ExplorerBot {           // 'Automaton.Body'.
17   ADComponentsJava; // Redundant generator for
18                           // 'ActivityDiagram.Body'.
19   ActivityDiagramJava; // Redundant generator for
20                           // 'ActivityDiagram.Body'.
21
22   ClassdiagramJava;
23
24 }
```

AC

Listing 8.8: This application configuration is missing a code generator for the language `Automata.AutomatonContent` and provides too many code generators for the language fragment `ActivityDiagram.Body`

Overspecification of generators responsible for a specific language fragment is prohibited as well. As component generators can contribute behavior language generation capabilities, this context condition considers behavior generators and component code generators for provided behavior language translation capabilities.

The two example behavior generators depicted in Listing 8.8, `ADComponentsJava` (ll. 1-7) and `ActivityDiagramsJava` (ll. 9-14), for instance, support translation of the UML/P activity diagram language fragment `ActivityDiagram.Body` (ll. 6,12). Hence, the application configuration `ExpJava` (ll. 16-25), which employs both code generators, is invalid and this context condition produces the displayed errors (ll. 19-24).

Convention Conditions

Application configuration models resemble singleton [GHJV95] classes of object-oriented systems in the sense that they cannot be instantiated multiple times for different projects. Therefore, the single naming convention requires that the names of application configuration models begin with a capital letter as well.

CC1: The names of application configuration models begin with a capital letter

The names of application configuration models should begin with a capital letter. Otherwise, this context conditions produces a warning as displayed in Listing 8.9. Here, the illustrated application configuration `expRob` begins with a lower-case letter, which produces the displayed warning (ll. 1-2).

```
1 application expRob  // Application configurations
2   for rob.Explorer { // start upper-case.
3     ...
4 }
```

App

Listing 8.9: Application configuration `expRob` begins with a lower-case letter and consequently raises a warning (ll. 1-2).

Referential Integrity Conditions

The application configuration language's referential integrity conditions check whether the constituents referenced by name exist and whether these references are valid. This comprises checking that all subcomponents of the referenced architecture are bound and that the code generators are compatible.

CR1: The referenced architecture exists.

An application configuration is meaningful only in the context of a software architecture. Hence, it is crucial that the referenced software architecture exists. In case the architecture is missing, the context conditions regarding bindings are meaningless and some of the conditions related to code generators are as well. Considering `ExplorerBot` (Figure 3.2) and related component types being the only component types available, the application configuration `BBRobotino` of Listing 8.10 is invalid as the referenced software architecture `arc.XumperBot` cannot be resolved. Hence, this context condition raises the depicted error (ll. 1-2).

```

1 application BBRobotino  // Inexistent architecture
2   for arc.XumperBot {           // 'arc.XumperBot'.
3     // ...
4 }
```

App

Listing 8.10: This application configuration references the nonexistent software architecture `arc.XumperBot`, which raises the error depicted.

CR2: All interface subcomponents of the referenced architecture are bound.

Code generation for software architectures with interface components either cannot produce executable systems or must produce executable systems with underspecified behavior. This is due to the interface components omitting any executable behavior. While individual code generators may produce executable systems from interface components, MontiArcAutomaton cannot assume this in general. Therefore, application configurations that reference architectures with interface components, but do not bind all of these components are considered erroneous and are rejected by this context condition.

```

1 application TurtleBumper  // Unbound interface
2   for arc.ExplorerBot {           // component 'sensor'.
3
4     bind left to TurtleBotMotor();
5     bind right to TurtleBotMotor();
6
7     generators.ComponentsWithAutomataPython;
8     generators.CDPython;
9 }
```

App

Listing 8.11: This application configuration does not bind the interface component `sensor` of the referenced software architecture, which gives rise to the depicted error.

The application configuration `TurtleBumper` of Listing 8.11 references the software architecture `ExplorerBot` depicted in Figure 3.2, which contains the three interface subcomponents `sensor`, `navigation.left`, and `navigation.right`. As it only provides bindings for the latter two, this context condition raises the error depicted in the listing (ll. 1-2).

CR3: All bound subcomponents exist and are of interface components.

Application configurations binding subcomponents that do not exist in the referenced software architecture or that are no interface components are obviously considered erroneous as well (cf. Def. 4). Permitting bindings to other component types would allow replacing all subcomponents of the referenced software architecture, and therefore

render the initial architecture meaningless. Listing 8.12 illustrates both errors: The application configuration `DistributedBB` for the platform-independent architecture `arc.ExplorerBot` binds the subcomponent `controller`, which is of non-interface kind (l. 4). The binding of subcomponent `logger` (l. 7) raises another error as that subcomponent does not exist in the software architecture.

```
1 application DistributedBB
2   for arc.ExplorerBot {
3
4     bind controller to JavaCtrl(); ✗ // Subcomponent 'controller'
5                           // is no interface component.
6
7     bind logger to JavaLogger(); ✗ // Inexistent subcomponent
8       // ...
9 }
```

App

Listing 8.12: The application configuration `DistributedBB` binds a subcomponent of non-interface type (l. 4) and a nonexistent subcomponent (l. 7). Therefore, it gives rise to two errors.

CR4: All implementation components types used for bindings exists.

As the bound subcomponents must exist in the software architecture, so must the bound platform-specific implementation component types exist as well. Otherwise, trying to apply the binding transformation (cf. Listing 6.2) will fail. Consequently, such application configurations are prohibited and rejected by this context condition. The application configuration `ScalaBumperBot` of Listing 8.13 demonstrates the error raised by context condition CR4 on a binding from interface subcomponent `sensor` of `ExplorerBot` to the nonexistent component type `NXTLadar`. Therefore, this context condition raises the error depicted (ll. 4-5)

```
1 application ScalaBumperBot
2   for arc.ExplorerBot {
3
4     bind sensor to NXTLadar(); ✗ // Inexistent component type
5       // ...
6 }
```

App

Listing 8.13: Binding the existing interface subcomponent `sensor` to an nonexistent component type raises the depicted error.

CR5: All referenced code generators exist.

Obviously, all code generators referenced in an application configuration must exist. Otherwise, checking code generator related context conditions and proper code generation are impossible. Considering only the Java generators AutomataJavaTS, ClassdiagramJava, and ComponentsJavaTS (introduced in Section 7.4), the application configuration EmbeddedJavaBot of Listing 8.14 is invalid as it references the nonexistent code generator SmallComponentsJava (l. 4).

```
1 application EmbeddedJavaBot
2   for arc.ExplorerBot {
3
4     SmallComponentsJava; ✖ // Generator does not exist.
5     // ...
6 }
```

App

Listing 8.14: The application configuration EmbeddedJavaBot references an nonexistent code generator (l. 4) and, thus, is erroneous.

CR6: All referenced code generators conform to the same run-time environment.

Application configuration models select multiple code generators to produce an integrated, executable system. To ensure proper integration, the participating component code generator and the behavior generators must produce compatible artifacts. With MontiArcAutomaton, this is expressed as conformance to the same run-time environment. Considering the Java code generators AutomataJavaTS, CDJava, and the Python code generator ComponentsPython (Section 7.4), the application configuration model SimulationBot of Listing 8.15 is invalid: the generators AutomataJavaTS and ComponentsPython produce artifacts conforming to different run-time environment (ll. 1-2).

```
1 application SimulationBot ✖ // Generators 'AutomataJavaTS'
2   for arc.ExplorerBot { // and 'ComponentsPython' use
3     AutomataJavaTS; // different run-time environments.
4     CDJava;
5     ComponentsPython;
6 }
```

App

Listing 8.15: Two of the code generators used by application configuration SimulationBot rely on different run-time environments (ll. 1-2).

CR7: Code generators and bound component types conform to the same run-time environment.

Considering the component generator ComponentsPython of Listing 7.6 and the platform-specific components of the javats library depicted in Figure 6.8, the application configuration OfficeBot of Listing 8.16 is erroneous. The component generator produces artifacts that conform to the run-time environment pythonuntimed (cf. Listing 7.6, l. 8), while subcomponent left is bound to component type NXTMotor conforming to the run-time environment java-timesync. Applying bindings and code generators would produce a GPL implementation of component type ExplorerBot, which, for instance, expects behavior implementations to conform to the run-time environment pythonuntimed – which is incompatible to the behavior implementation of component type NXTMotor.

```
1 application OfficeBot
2   for arc.ExplorerBot {
3
4     bind left to NXTMotor(PORT_A); ✗ // Bound component type does
5                               // not use the component
6     ComponentsPython;           // generator's RTE.
7     // ...
8 }
```



Listing 8.16: The application configuration OfficeBot employs a code generator and a binding of incompatible run-time environments.

Type Correctness Conditions

The type correctness conditions of the application configuration language check the type correctness of bindings. This includes checking that interface subcomponents are bound to platform-specific components only and that they are parametrized properly.

CT1: The bound implementation's component type is platform-specific.

Binding interface components to interface components does not contribute behavior to the architecture. In consequence, this is prohibited and context condition CT1 performs the according checks. To this end, it resolves all implementation component types referenced in bindings and checks whether these are of interface components. The application configuration GroovyExplorer shown in Listing 8.17 contains such a binding (l. 4) that maps the interface component sensor to the interface component DistanceSensor (cf. Figure 6.7). Context condition CT1 detects this and produces the displayed error.

```

1 application GroovyExplorer
2   for arc.ExplorerBot {
3
4   bind sensor to DistanceSensor(); ✘ // Cannot bind to interface
5   // ...
6 }
```

App

Listing 8.17: The component type `DistanceSensor` is interface and, therefore, cannot be bound (l. 4).

CT2: The bound subcomponent's type is a super type of the component type to be replaced.

To ensure interface compatibility, the types of interface subcomponents may only be replaced with inheriting component types. Context condition CT2 ensures this by traversing the inheritance hierarchy of each replacing type mentioned in a binding and raises an error if that hierarchy does not contain the bound subcomponent's type. Assuming, for instance, that component type `NXTMotor` does not extend - directly or transitively - the component type `DistanceSensor` (cf. Figure 6.7), the application configuration `ROSPythonNXT` as depicted in Listing 8.18 violates this context condition as it binds the subcomponent `sensor` to `NXTMotor` (l. 4).

```

1 application ROSPythonNXT
2   for arc.ExplorerBot {
3
4   bind sensor to NXTMotor(); ✘ // NXTMotor does not extend
5   // ...
6 }
```

App

Listing 8.18: The depicted application configuration binds the subcomponent `sensor` to the component type `NXTMotor`. As `NXTMotor` does not descend from `DistanceSensor`, this is invalid.

CT3: Arguments for all new implementations component types' configuration parameters without default values are provided.

Bindings introduce new component types to the software architecture, which must extend the bound subcomponent's type. The extension mechanisms of MontiArcAutomaton allow to extend the list of configuration parameters inherited from a super component, which is exploited to specify additional platform-specific configuration parameters (as required by Req. *MRQ-8.1*). If these parameters do not provide default values, the bindings must specify arguments for these. However, as arguments for some of these

parameters may be already defined in the software architecture, these are considered paramount and may not be redefined by bindings (cf. Req. *MRQ-8.2*). This follows the idea that the original software architecture is decisive and therefore, bindings may give arguments for the parameters of implementation component types that are not already given in the software architecture only.

In consequence, this context conditions can raise two different errors: First, bindings can omit arguments required by the bound component type. Second, they can overwrite arguments already specified in the software architecture. Both errors are illustrated in the application configuration *BumperBotCPP* depicted in Listing 8.19, which binds the subcomponents *sensor* (l. 4) and *timer* (l. 6). The component type *NXTUltrasonic* bound to *sensor* extends *DistanceSensor* (cf. Section 6.2.1) and introduces the configuration parameter *port* to describe which physical port of the NXT robot its behavior implementation is connected to. The binding omits to specify an argument for this parameter and thus, this context condition raises the displayed error (l. 4). The second binding from *timer* to *JavaTimer* specifies an argument for the parameter *delay*, which *JavaTimer* inherits from *Timer*. This is prohibited as the software architecture already defines an argument for it, which is considered paramount. Hence, this context condition raises the second error (l. 6).

```
1 application BumperBotCPP
2   for arc.ExplorerBot {
3
4     bind sensor to NXTUltrasonic(); ✖ // Missing argument.
5
6     bind timer to JavaTimer(2); ✖ // Overwriting argument
7     // ... // of component 'timer'.
8 }
```



Listing 8.19: Application configuration *BumperBotCPP* contains two bindings with missing (l. 4) and redundant (l. 6) arguments.

CT4: Arguments for all generic type parameters of bound subcomponents are provided and correct.

Extending platform-specific implementation component types may add new generic type parameters or reduce these by assigning values to the super component's generic type parameters. Generic type parameters can govern types of ports. To retain the validity of connectors, ports of the platform-specific component type must be of the same types than the ports of the bound interface component. Therefore, this context condition requires that for each port of the interface subcomponent, there is a port of the same name and type in the subcomponent derived from applying the bindings' generic type arguments to the replacing component type. This is a stronger requirement than extension and suffices to ensure the integrity of the connectors.

```

1 interface component GenericSensor<T> {
2   port
3     out T data;
4 }
5
6 component RGBASensor<U> extends GenericSensor<U> {
7   // Component modeling elements.
8 }
9
10 component ExtendedBumperBot {
11   port
12     out Integer reading;
13
14   component GenericSensor<Integer> gs;
15   connect gs.data -> reading;
16   // Additional components and connectors.
17 }
```

MAA

Listing 8.20: The composed component `ExtendedBumperBot` instantiates `RGBASensor` and defines its generic parameter to be `Integer`.

Consider the component types `GenericSensor`, `RGBASensor` and their usage with component type `ExtendedBumperBot` as depicted in Listing 4.12. The interface component `GenericSensor` declares the generic type parameter `T` to define the type of its outgoing port `data`. Hence, the possible connectors are restricted by the actual argument for parameter `T` passed to it at subcomponent instantiation. The component type `RGBASensor` extends `GenericSensor` and employs a generic type parameter `U` which it expects to be defined a instantiation and passes to its `GenericSensor` part (l. 6). Hence, instantiating `RGBASensor` with a specific data type entails that it defines a port `data` of that type. The component type `ExtendedBumperBot` yields an interface consisting of the single `Integer` port `reading` (l. 12) and instantiates a subcomponent `gs` of type `GenericSensor` with type argument `Integer` (l. 14), which entails that this subcomponent's port `data` is of type `Integer`. It furthermore connects that port to its outgoing `Integer` port `reading`. In this context, the application configuration of Listing 8.21 is invalid: If the new component type of `gs` should become `RGBASensor<Float>`, its port `data` will assume the type `Float` and, hence, cannot be connected to port `reading`. Thus, this context condition raises an error as depicted and rejects this application configuration.

8.2 Processing MontiArcAutomaton Applications

MontiArcAutomaton MontiArcAutomaton applications are projects containing a MontiArcAutomaton ADL software architecture and at least one application configuration model. MontiArcAutomaton processes applications in two phases: First, it processes

```
1 application RobotinoJava
2   for ExtendedBumperBot {
3
4     bind gs to RGBASensor<Float>(); ✘ // Invalid type parameter.
5     // ...
6 }
```



Listing 8.21: Application configuration RobotinoJava produces an error by binding sensor to component type `RGBASensor<Float>` (l. 4), hence, changing the type of port data from `Integer` to `Float` and, thus, violating the integrity of `ExplorerBot`.

the application configuration model, checks its context conditions, applies the bindings, and stores the selected code generators. Afterwards, it processes the transformed software architecture, checks its context conditions, applies further transformations, starts generator composition, and ultimately generates GPL artifacts. Enabling this procedure, requires integration of the participating languages. To this end, MontiArcAutomaton employs the MontiCore language integration mechanisms as follows:

- Behavior languages are embedded into the MontiArcAutomaton ADL and aggregated into the MontiArcAutomaton ADL language family.
- The MontiArcAutomaton ADL language family is aggregated with the generator description language and the application configuration language to the overall MontiArcAutomaton language family.

This coupling results in the overall MontiArcAutomaton language family as depicted in Figure 8.3. Hence, the MontiArcAutomaton can resolve application configurations, as well as the referenced architecture and generator descriptions.

In processing applications in two phases, executing MontiArcAutomaton differs from typical MontiCore executions (cf. Figure 2.3). Fully processing a MontiArcAutomaton application requires participation of the toolchains of the application configuration language, the generator description language, and the MontiArcAutomaton ADL.

First, the application configuration infrastructure parses the application's configuration model as depicted in Figure 8.4. This may include loading the symbol table entries of referenced generators, which requires the generator description infrastructure to provide these symbol table entries. Providing these entries may include parsing the generator description models of participating generators, checking their context conditions, and creating their symbol table entries. Similarly, the MontiArcAutomaton ADL infrastructure must produce the component symbol table entries for the software architecture. This may also include parsing, context condition checking, and symbol table creation. Afterwards, the symbol table for the processed application configuration model can be created and the language's context conditions are checked. Subsequently, the binding transformation (cf. Section 6.3) of the application configuration language transforms the

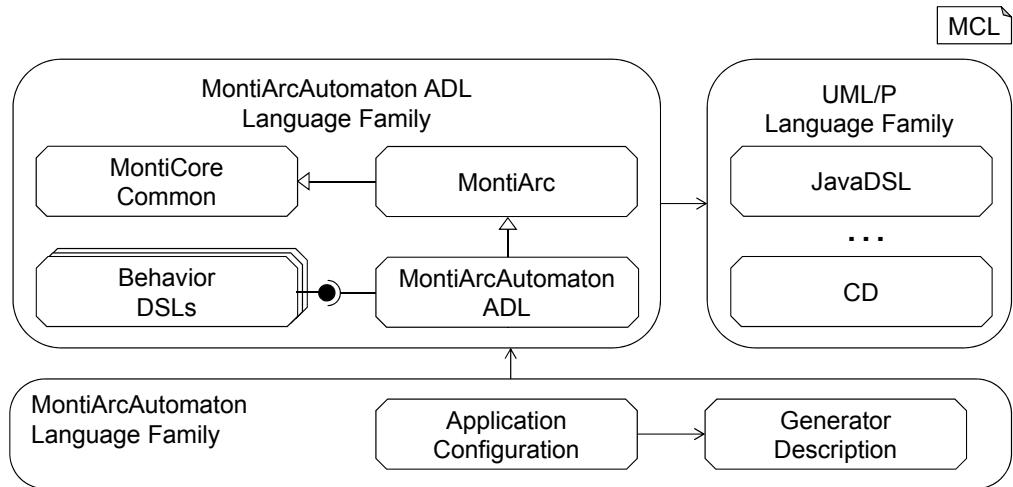


Figure 8.3: The MontiArcAutomaton language family combines software architectures, code generator descriptions, and application configurations.

referenced software architecture and passes it to the MontiArcAutomaton ADL infrastructure. The latter checks the well-formedness of the architecture model, composes the participating code generators, and ultimately generates the target GPL artifacts.

8.3 Modeling MontiArcAutomaton Applications

Reusing a single platform-independent architecture with two platforms employing different GPLs (cf. Section 3.1) with C&C applications builds on the modeling languages presented in Chapter 4, the bindings and libraries of Chapter 6, and the code generation framework of Chapter 7.

Figure 8.5 depicts the constituents of the MontiArcAutomaton application comprising a platform-independent MontiArcAutomaton software architecture `ExplorerBot` and the artifacts to derive two platform-specific implementations from it. Creating these two implementations from the `ExplorerBot` software architecture of Figure 3.2 (interface subcomponents are shaded blue), requires the application modeler to provide corresponding application configuration models. Therefore, she integrates a behavior language provided by a language engineer into MontiArcAutomaton as presented in Section 4.2. Afterwards, she models the software architecture as required, but uses the interface components `DistanceSensor`, `Timer`, and `Motor` from the `BumperBotModels` library. To transform the platform-independent `ExplorerBot` into platform-specific variants for the robot platforms depicted in Figure 3.1, she selects proper implementation libraries for both platforms. For the scenario, these are the `JavaNXT` library of Section 6.2.2 for the Lego NXT platform and the `PythonROS` library presented in Section 6.2.3 for the Pioneer 3-AT platform. Given the `ExplorerBot` software architecture containing interface components and their platform-specific realizations, she creates two application

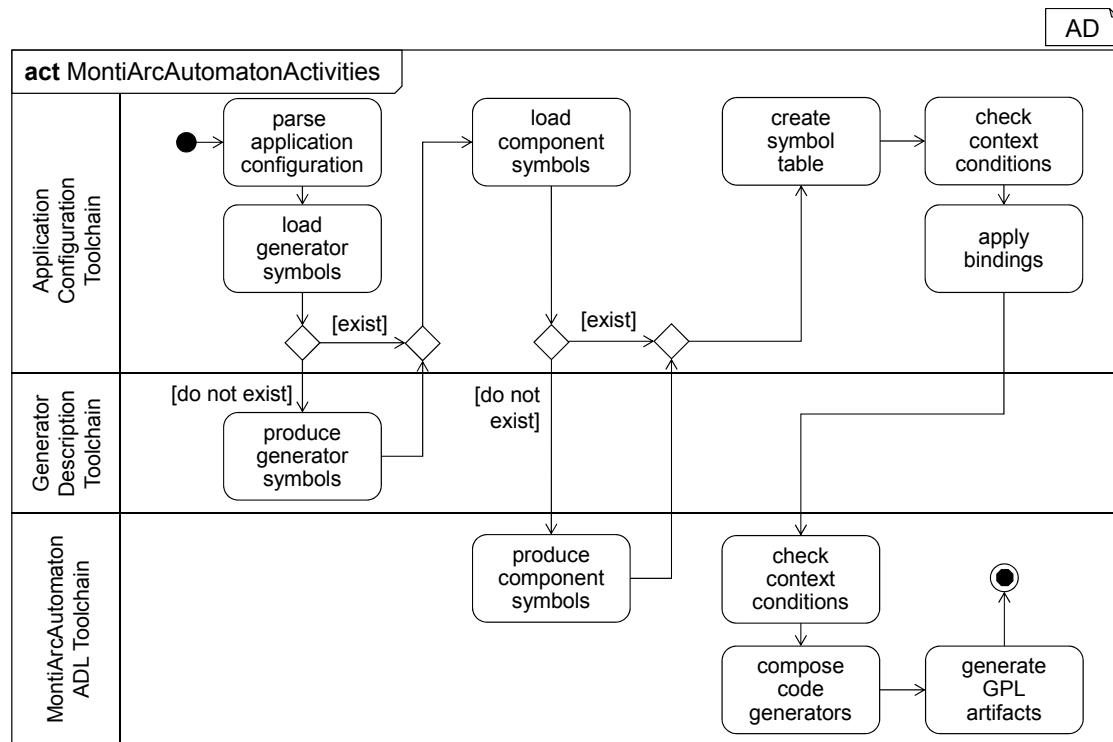


Figure 8.4: A typical MontiArcAutomaton execution involves the toolchains of all participating languages.

configuration models (cf. Chapter 6), one for each platform and models proper bindings for each platform. Afterwards, she selects appropriate combinations of run-time environments and code generators as presented in Chapter 7. From these application configuration models MontiArcAutomaton produces platform-specific architecture models (arrow “A”), which are translated into platform-specific GPL implementations (arrow “B”) afterwards.

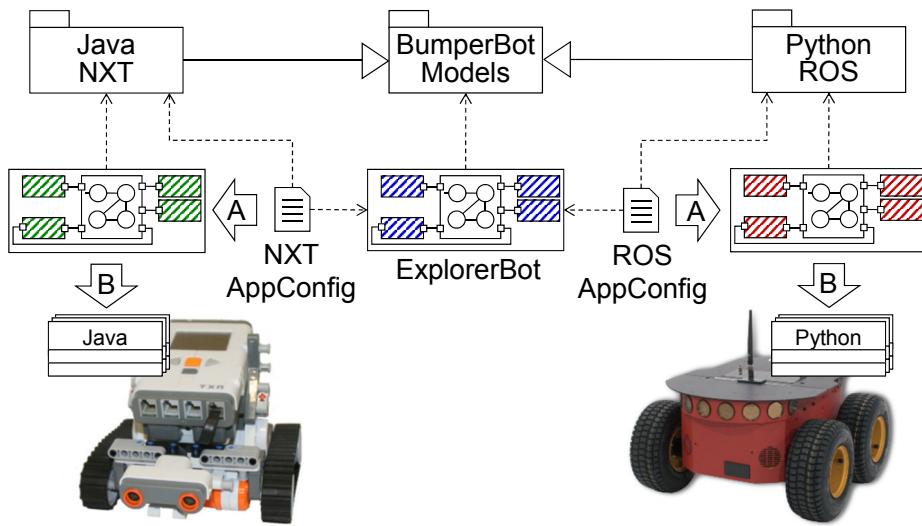


Figure 8.5: Constituents of a reusable, platform-independent C&C application, their prime relations, and produced artifacts.

This supports non-invasive reuse along six dimensions:

1. Single components: Both platform-independent and platform-specific components can be reused with corresponding architectures in a black-box fashion. The former with all platform-independent architectures, the latter with all platform-specific architectures for the same target GPL and RTE.
2. Complete architectures: Platform-independent architectures with interface components can be reused in a black-box fashion with similar applications as well. It suffices to provide required platform-specific components and to specify corresponding bindings.
3. Behavior languages: Behavior modeling languages can be reused in different language combinations with little effort.
4. Component generators: Generators that produce artifacts for the representation of structural architecture elements with respect to a specific target GPL and RTE can be reused with all platforms supporting both.
5. Data type generators: The production of data types is decoupled from RTEs, hence for each platform supporting a data type generators target GPL, reusing it amounts to simply selecting it in an application configuration.
6. Behavior generators: For the same combination of target GPL and RTE, reusing behavior generators also amounts to selecting these with a compatible generator family as well.

Furthermore, if a new target platform requires implementations in a yet unsupported GPL, the effort to develop required infrastructure parts is well separated between the participating experts: The run-time environment developer creates a RTE and the generator developers provide proper code generators. The implementation library provider provides platform-specific components and their implementations for this new GPL. Afterwards, the application modeler integrates these by simply defining a new application configuration model.

8.4 Discussion and Related Work

Architecture configuration models tie together software architecture, implementation libraries, and code generators of a C&C application. To this effect, they specify bindings and select code generators in a concise syntax close to the MontiArcAutomaton ADL. With these, reusing a single software architecture with multiple target platforms requires minimal effort. However, to the best of our knowledge, other architecture modeling languages consider software architectures the only development models [MDEK95, GMW97, GMW00, VVKM00, PM03, NDZR04, DSLT05, BCL⁺06, LPJ10, FG12]. Hence, expressing features about a software architectures, such as how to replace subcomponents, or how to generate code for it, pushed into the modeling framework and rarely documented. An exception to this is ArchJava, which enriches Java with architecture modeling elements and hence is tied to Java [ACN02]. Where code generation is supported [KGO⁺01, BGBK08, FG12, SSL11, DKS⁺12] this is tied to a few fixed code generators and neither their reuse, nor their composition are taken into account.

Also, MontiArcAutomaton currently does not support composition of multiple code generators that support the same input modeling languages (e.g., a component generator and a behavior generator that both process the same behavior language). While development of a heuristic or selection mechanism for this is easy, introducing new elements to the code generator selection mechanism (Section 8.1.1) would complicate the modeling language and introduce additional notational noise [Wil01]. Whether employing a more general modeling language for this (such as feature diagrams [SHT06]) is better suitable to express the requirements in generator composition is to investigate.

Application configuration models define model transformations from single platform-independent software architecture into single platform-specific implementations. Supporting multiple related blocks of bindings and code generators in a single application configuration is generally possible. However, such multi-application configurations would entail updating this central artifact whenever a new target platform is included or platform-specific components for a single target platform change. Changes to a single target platform in configurations consisting of multiple models do not interfere with the other application configuration models and, hence, reduce the risk of introducing errors by side effects.

Chapter 9

Experiments

Man prefers to believe what he prefers to be true.

Francis Bacon

We have evaluated MontiArcAutomaton with various component behavior languages on multiple robot platforms ranging from simulators to educational Lego NXT robots to complex service robots. To this effect, we have developed code generators for different target languages [RRW13b], which includes code generation to Mona [EKM98] for formal analysis, EMF Ecore [SBMP08] for graphical editing, and Java and Python for deployment. Many applications require additional components to access platform-specific software and hardware. Hence, proper libraries for specific target platforms and applications were developed. This enables to use MontiArcAutomaton with

- Lego NXT robots and simulators using ROS [QGC⁺09],
- Festo Robotino robots using ROS [QGC⁺09], SmartSoft [SSL11],
- Lego NXT robots using LeJOS [LeJ], and
- the Simbad¹ simulator.

Additionally, we have developed GPL specific libraries to provide GPL functionalities (e.g., file I/O). The libraries comprise from 4 (ROS turtlesim²) to 21 (LeJOS NXT) components and can be easily imported by MontiArcAutomaton applications to use these with different platforms. In the following, Section 9.1 illustrates three evaluations in academic contexts before Section 9.2 presents three case studies in academic and industrial contexts. For each evaluation and case study, we describe the evaluated features of MontiArcAutomaton. For evaluations, we also discuss the results of surveys performed at different states of the projects.

9.1 Evaluations

We have performed three evaluations in lab courses at RWTH Aachen University. These lab courses were performed in different semesters, with students of different courses,

¹Simbad website: <http://simbad.sourceforge.net/>

²ROS turtlesim simulator: <http://wiki.ros.org/turtlesim>

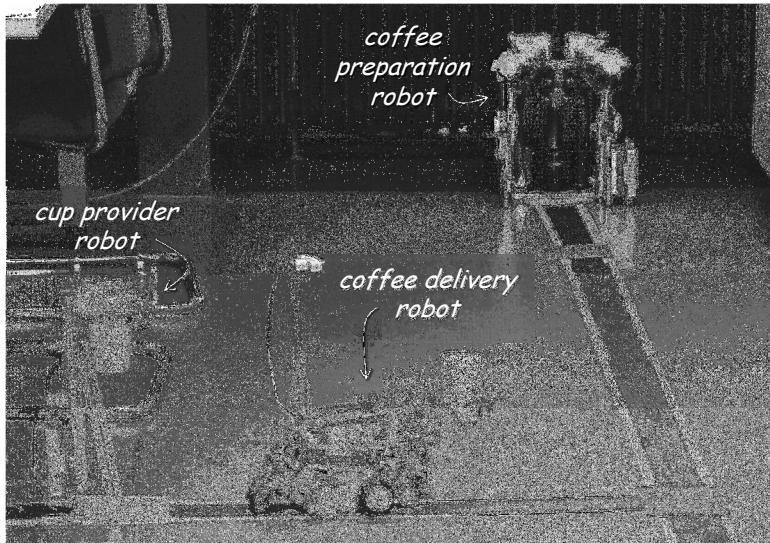


Figure 9.1: The distributed robotic coffee service implemented with Lego NXT robots using the LeJOS JVM as operating system.

and with different versions of MontiArcAutomaton. In each lab course, we conducted interviews with the participants and questionnaires. For each evaluation, the following sections discuss observations and threats to validity.

9.1.1 NXT Java Coffee Delivery

With the JavaNXT library presented in Section 6.2.2, and the javats code generator family presented in Section 7.4.1, we evaluated MontiArcAutomaton during a lab course in winter term 2012/13 with eight master level students [RRW13a]. We employed a development process following Scrum [Rub12]. The students acted as application modelers by developing a distributed robotic coffee service consisting of the three robots illustrated in Figure 9.1 and as application programmers for implementing the behavior of components not easily expressible as automata (cf. Req. TRQ-2). The run-time environment and system component implementations were provided.³ In this evaluation, the students used the MontiArcAutomaton ADL language family with embedded AUTOMATA to model a platform-specific software architecture and employed a monolithic code generator (cf. Req. TRQ-4) to reduce the number of Java artifacts of the resulting implementations (cf. Req. TRQ-8) for better performance on the Lego NXT robots.

With the system, users can issue coffee requests via a website. This website is hosted on a smart phone connected to the coffee preparation robot depicted in Figure 9.1 via Bluetooth. Once it receives a request, it commands the coffee delivery robot to fetch a plastic mug from the mug provider robot. Afterwards, the coffee delivery robot returns

³A video of the system in action is available via <https://www.youtube.com/watch?v=xvLYN-6awfk>

to the coffee preparation robot, instructs it via Bluetooth to prepare coffee and drives to the user who ordered the coffee. In lieu of sophisticated localization sensors, navigation was guided by black lanes with colored junctions. The system's software architecture consists of 23 component type definitions used in 60 component instances. Of these, 39 are components imported from various libraries.

The lab course was divided into three stages: First, the students prepared presentations of the course's topics ranging from MontiCore to MontiArcAutomaton to development infrastructure. Afterwards, they produced the coffee delivery system and finally, they worked on different aspects of the MontiArcAutomaton infrastructure. After the second stage, the students participated in a questionnaire on the modeling tools and their complexities [RRW13a]. The detailed survey results are available in Section B.1 of the appendix.

Observations

We observed that behavior modeling using automata was considered most complex (consumed 37% of the students' time), followed by physical robot construction (24%), Java behavior implementation (20%), and structure modeling (19%). Consequently, the students were more confident in using the other students' Java classes and component models (70%), than their automata (60%) and estimated the complexity of their team members' artifacts accordingly. Figure 9.2 (a) illustrates the estimates of the latter. Although the students considered AUTOMATA models more complex (6.3 points out of 10 points) than Java artifacts (4.3 points), they were similarly confident with both as displayed in Figure 9.2 (b). Exposing the same confidence to MontiArcAutomaton components as to Java, which the students learned in the first semester and used many times afterwards, highlights its benefits regarding comprehensibility.

The students of this lab course modeled atomic and composed MontiArcAutomaton components to describe the three robots' software architectures. In the development stage, they changed the behavior description of most atomic components from Java artifacts to behavior models. We therefore assume that MontiArcAutomaton is suitable for modeling the software architecture of complex, distributed software systems. Further observations have been published in [RRW13a].

Threats to Validity

This evaluation was performed in an educational context at RWTH Aachen University and the eight master level students participated to obtain a rated certificate. Due to the complexity of the task at hand and the size of the group, we could not separate the lab course into an experimental group and a control group. This raises threats to the evaluation's internal validity (causality) and to its external validity (generalizability).

Threats to this evaluation's internal validity arise from the student's lack of previous modeling experience compared to experience with Java as prescribed by their curriculum. This selection bias manifested in the greater confidence in Java artifacts compared to automata. Furthermore, the instruments to evaluate MontiArcAutomaton are subject to

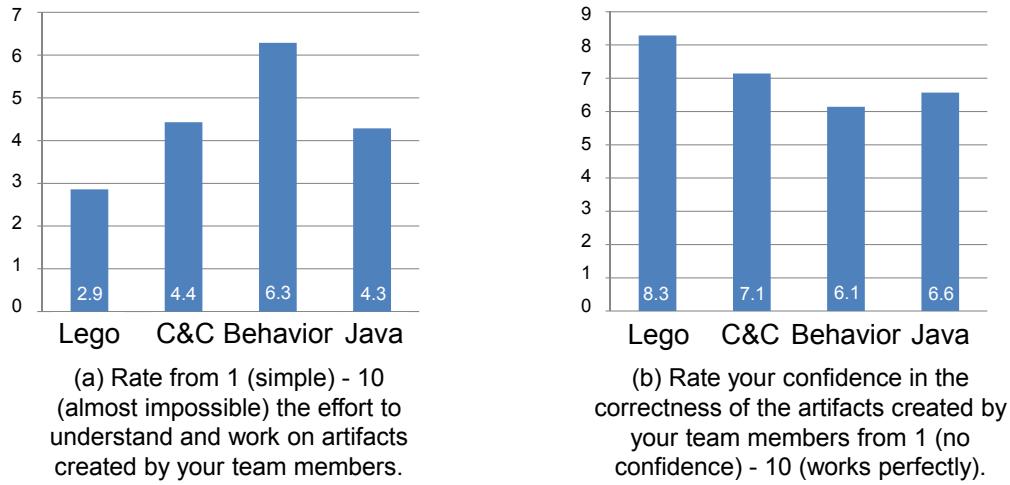


Figure 9.2: The students estimated AUTOMATA behavior modeling most complex and had the least confidence in atomic components with AUTOMATA models developed by their team members.

issues themselves. For instance, participants interpret the scales of questionnaire answers differently and give answers based on their self-perception only. We tried to alleviate this bias with interviews. However, as this lab course was on model-driven development of robotics applications and the students knew that they were assessed, the study may also be subject to compensatory rivalry. As the study did not comprise a control group and an experimental group, the effect of such rivalry can be hardly estimated.

Threats to external validity arise from the small number of participating students and from the interviews being performed by the advisors instead of interviewers unrelated to the desired certificate's grade. Whether the results of this case study are reproducible with other groups of similar master students or whether the same students would perform differently in another context is hardly predictable in a university context. Lab courses are performed every semester and hence, the modeling languages and infrastructure advance from one lab course to another. Furthermore, master level students at RWTH Aachen University must participate in a single lab course only, hence assembling the same group for a similar task in another context is rarely possible.

9.1.2 Robotino ROS Python Transport Services

In the winter term of 2013/14, we evaluated MontiArcAutomaton with a Python code generator family (cf. Req. TRQ-1), a Python run-time environment, and the ROS Python Robotino modules in another lab course. In this course, nine master level students developed a model-driven logistics application for a Festo Robotino⁴ robot using a Kinect and speakers for user interaction. The group again employed a Scrum [Rub12]

⁴Robotino website: <http://servicerobotics.eu/robotino/>

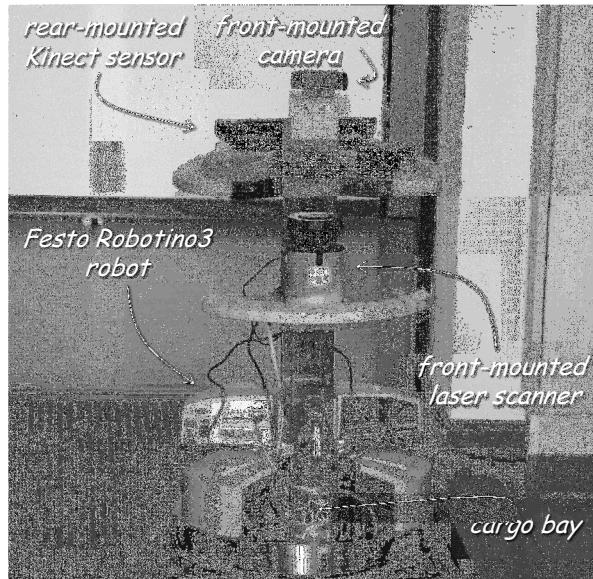


Figure 9.3: The Robotino ROS transport service robot with Kinect, speakers, and front-mounted laser scanner.

methodology.⁵ The robot, depicted in Figure 9.3, also used a front-mounted laser sensor and integrated distance sensors to navigate and avoid obstacles. The students used the robot operating system ROS [QGC⁺09] in its Python implementation to interface robot drivers and a version of AUTOMATA without embedded Java expressions. The former is interesting as the students curricula did not prescribe to learn Python, which we assumed to reduce the selection bias regarding previous experiences with one of the languages.

The software architecture was modeled with 31 components. Of these components, nine are atomic with automata and 17 are atomic with platform-specific implementations. The architecture’s top-level is depicted in Figure 9.4. The subcomponents Navigation, MapProvider, and TaskManager are composed from further components.

In this course, the students enacted the roles of application modelers and application programmers. The latter was necessary as, for example, the behavior of the ROS-specific components interfacing the Kinect sensor had to be handcrafted. In this system, a website is hosted on the robot, which receives tasks such as “fetch item X from room Y” from this website. Embedded AUTOMATA transform these tasks into motion and interaction commands. These commands are sent to the components Navigation and UserInterface, which transform these into platform-specific primitives.

This lab course consisted of two stages: In the beginning, the students prepared and held introductory talks on the course’s topics. Afterwards, they developed parts of the logistics software in groups of two or three students organized in a Scrum teams. We

⁵A short video documentation is available via <https://www.youtube.com/watch?v=u6LF8KjvDgM>

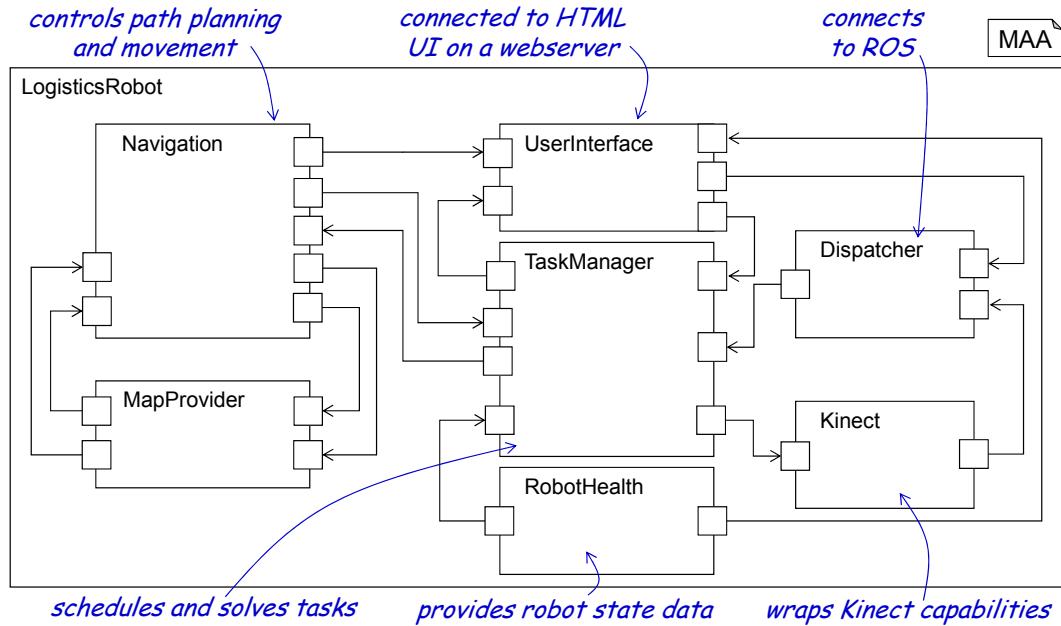


Figure 9.4: Software architecture of the Robotino ROS Python transport service. The components Navigation, MapProvider, and TaskManager are composed.

conducted two surveys in this lab course: the first after a few weeks in the development stage, the second shortly before the course ended. The first questionnaire contained 12 questions and the second contained 18 questions. The additional 6 questions focused on reuse and the applicability of Scrum in university courses. Both questionnaires and the aggregated results are available in Section B.2 of the appendix.

Observations

In the beginning, the students spent more time learning to use the ROS infrastructure (27% of their time) than modeling automata (22%) or plain components (21%). We assume this is partly due the complexity of the ROS infrastructure as well as due to the conciseness of the modeling languages. In the end, the components and automata consumed less time (14% and 15%, respectively), but the time required to understand ROS (44%) and the plain Python artifacts (16%) increased. This might be due to ROS hiding its complexity, i.e., simple and well-documented features are easy to use but once the software grows, the complexity of solving the underlying problems grows even faster. Figure 9.5 (a) illustrates this development. It also shows the actual development times (b) estimated by the students in both questionnaires. Initially, the students spent much time developing component & connector structures (35% of their development time) and developing Python artifacts (45% including ROS infrastructure). In the end, the development effort shifted towards Python development (64%), which hints at the C&C

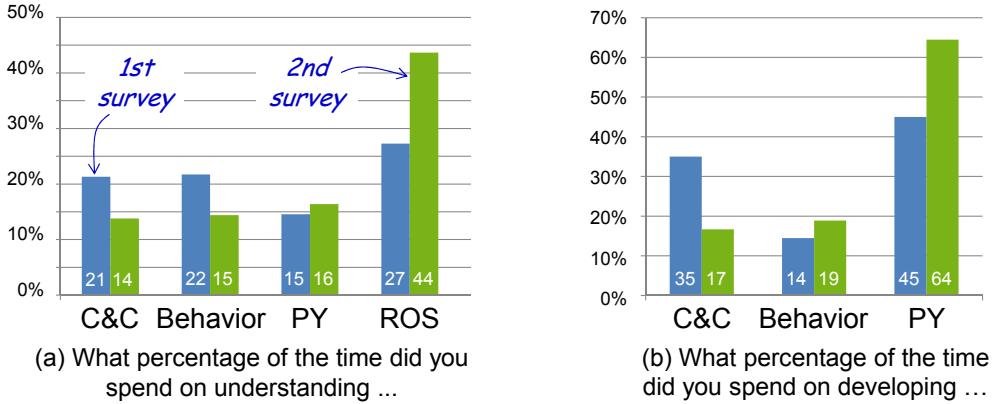


Figure 9.5: The estimated distribution of the students' time learning and developing with the respective languages. The left column represents survey results from the first questionnaire, the right column results from the second questionnaire. Numbers in the columns represent the actual values.

and automata models being more stable than the Python artifacts. We assume this is due to their reduced complexity compared to Python code.

We also asked the students to estimate the complexity of artifacts of their team members to reduce the bias of overestimating their skills. The results indicate that the students in the beginning underestimated the complexity of almost all participating technologies and languages as depicted in Figure 9.6 (a). It is notable that the students' estimate in the complexity of Python artifacts developed by their colleagues did not increase, where the estimated complexity of ROS nodes, AUTOMATA behavior models, and C&C models increased between 41% (AUTOMATA) and 81% (C&C structure). In the same vein, we asked the students to estimate their confidence in the correctness of artifacts created by their team members (Figure 9.6 (b)). The estimated correctness of C&C structure artifacts and AUTOMATA models remained stable while the confidence in Python artifacts and ROS nodes increased slightly. We assume this is due to the similarity of Python and Java (which is taught in the students' curriculum).

The students also estimated the correctness of the artifacts created by themselves. While the estimates regarding C&C models, behavior models, and Python artifacts remained stable between 6.8 points and 8.6 points (both out of 10 points), the confidence in the correctness of ROS nodes raised from 2.8 points in the first survey to 7.3 points in the second survey.

This is due to the inherent complexity of developing ROS nodes, selecting the correct nodes from their vast repository, and configuring the individual nodes properly with arguments determined by trial-and-error. Once selection and configuration are stable, ROS seems to perform reliably. The second questionnaire also revealed that the students assumed that they could have modeled twice as many components with automata than

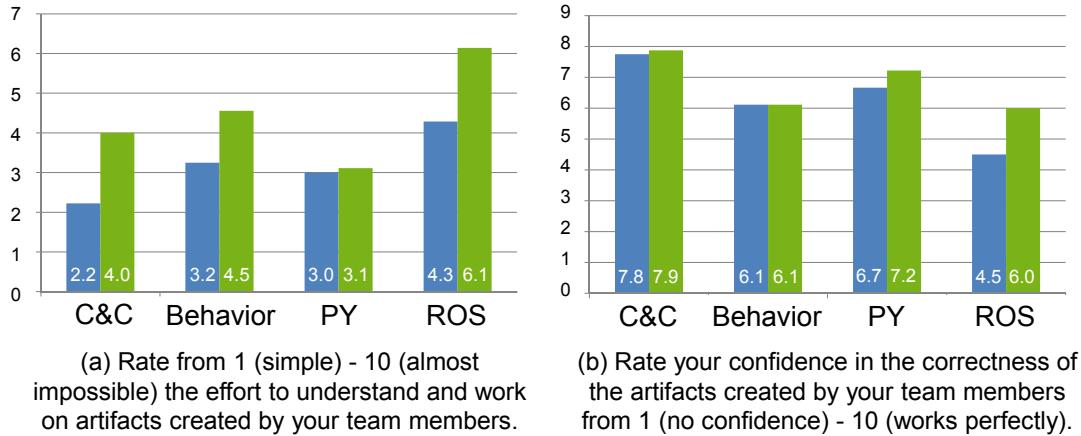


Figure 9.6: Estimated complexity of their team members' artifacts and the students' confidence.

they actually did and that ROS was considered the most complex technology (4.4 points out of 6 points) of that lab course. ROS was followed by the employed continuous integration toolchain (4.1 points), the automata behavior language (3.8 points), plain C&C models (2.9 points), and Python (1.9 points). This highlights that the object-oriented Python language was closer to Java, which the students had plenty experience with, than to MontiArcAutomaton. Nonetheless, the students assumed that reusing their solution with a different but similar robot is possible within 5 days of development. Regarding the application of Scrum, the seven of the eight students assumed that Scrum helped development and that sprint planning meetings and daily Scrum meetings were most helpful (9.11 points and 8.6 points out of 10 points, respectively).

Overall, the students understanding of both the MontiArcAutomaton C&C language elements as well as of AUTOMATA converged during the lab course, whereas their understanding of Python and ROS diverged. We assume that this is due to the restricted complexity of the modeling languages, which points at the benefit of modeling C&C software architectures with MontiArcAutomaton. This also correlates with the fact that the students increased AUTOMATA modeling activities towards the end. The decrease of C&C modeling activities during the lab course points at the expected increase of architecture stability. We thus assume that MontiArcAutomaton also facilitates development with more complex systems than presented in the previous lab course (Section 9.1.1). Furthermore, transforming MontiArcAutomaton ADL architectures to other GPLs than Java has shown to be feasible as well (cf. Req. TRQ-1).

Threats to Validity

Due to the task's complexity and the small group size, separating the students into a control group that uses only general-purpose programming languages and an experimen-

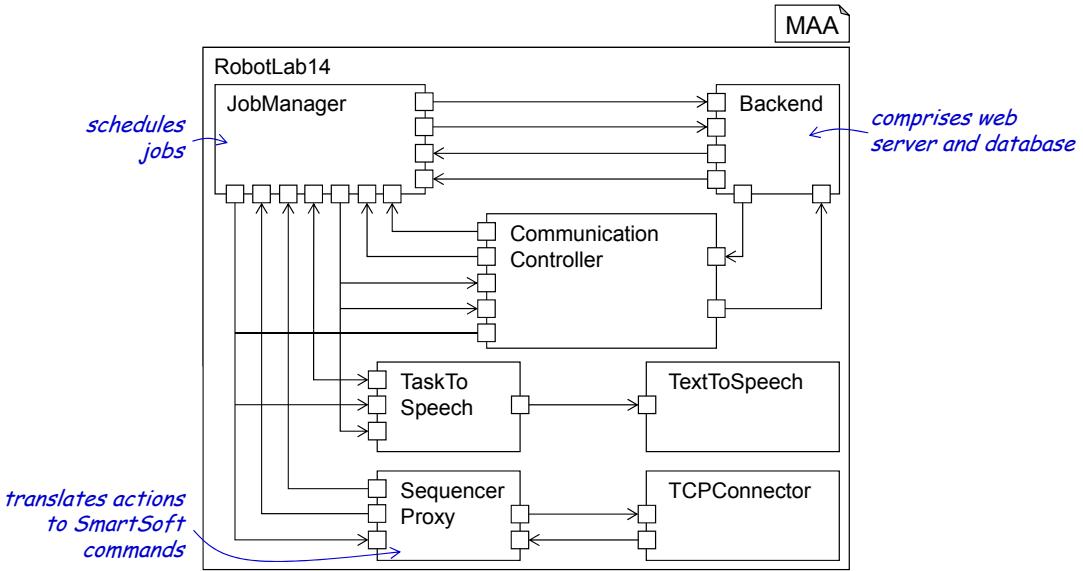


Figure 9.7: Top-level architecture of the logistics software system implemented in summer term 2014 for a Robotino robot controlled via SmartSoft and Python. Five of the displayed component types are composed.

tal group using mostly modeling languages was not feasible. This raises threats to both internal and external validity. Regarding the internal validity, there is a selection bias as all participants were well-trained in developing object-oriented systems, which supposedly influenced their estimates on the modeling techniques. Furthermore, questionnaires are an insufficient instrument as the answers are biased by their self-perception and regarding avoidance of extreme answers (avoiding the provided scales upper and lower ends). The external validity of this evaluation is threatened as the course had the topic model-driven development and the students received grades for their participation.

9.1.3 Robotino SmartSoft Java Transport Services

In a lab course of summer term 2014, we again assigned the task to develop a robotics logistics application. In this lab course, 14 students from different computer science courses participated.⁶ The students developed the software architecture with MontiArcAutomaton and used to the SmartSoft [SSL11] middleware to control the robot. The students employed Scrum [Rub12] again and used MontiArcAutomaton with integrated AUTOMATA models and the javats code generator family. The students acted as application modelers, application programmers, and implementation library providers and produced the architecture depicted in Figure 9.7. Of the depicted subcomponents, TextToSpeech and TCPConnector are atomic, the others are composed. In total, the software architecture consists of 28 component types. Of these, 6 are composed

⁶A video of the results is available at <https://www.youtube.com/watch?v=TIspANC9TY4>

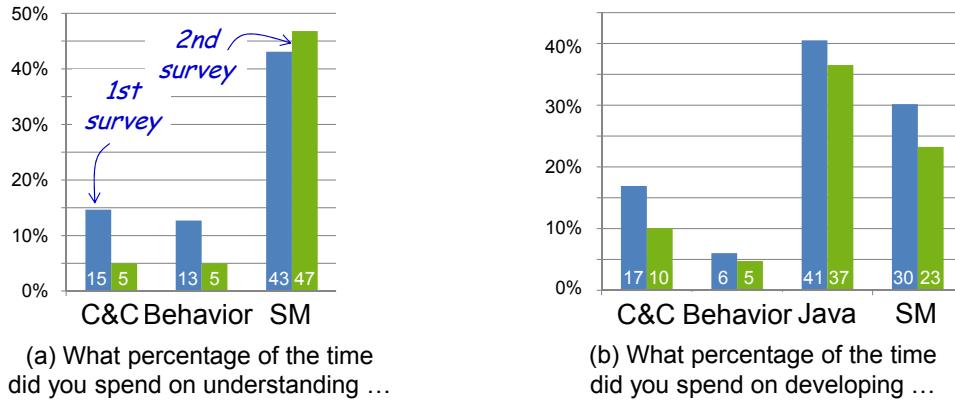


Figure 9.8: The students of this lab course spent most time understanding the SmartSoft middleware. The consumed time increased towards the end, while the time actually developing SmartSoft modules decreased.

containing 30 subcomponent instances. To communicate with the robot, a website and a tablet PC were used. Both were connected to the architecture via subcomponents of the component Backend. Logistics tasks are passed from the component Backend to the component JobManager. The latter translates these into commands send to the SmartSoft middleware via the component SequencerProxy.

Similar to the previous lab courses, we performed two surveys: one a few weeks after the course started and the other shortly before the course ended. Both surveys were conducted using questionnaires containing 12 (first) and 18 (second) questions again. The additional questions of the second survey are designed to measure the students' participation and the applicability of Scrum in this setting. Both questionnaires are available in Section B.3 of the appendix.

Observations

The observations during this course differ from the previous in various ways: First, the students spent more time understanding SmartSoft (SM) in the end of the course than in the beginning (see Figure 9.8 (a)), while the C&C structure and behavior languages consumed more time in the beginning than in the end. As SmartSoft and the modeling languages were introduced and worked with parallelly, we ascribe this to the intended specificness of MontiArcAutomaton. However, this contradicts with the second finding, that the students spent less time developing with SmartSoft in the end (Figure 9.8 (b)).

The students of this lab course also initially required more time in trying *conceptually wrong* approaches with Java than in the previous courses as depicted in Figure 9.9 (a). For both Java and SmartSoft (SM) this reduced clearly during the course, whereas the time spent for conceptually wrong approaches using the MontiArcAutomaton C&C modeling elements and AUTOMATA almost remained constantly low. Figure 9.9 (b) also shows that the students of this course students were more confident in modeling with

components and AUTOMATA models than in developing with Java or SmartSoft (SM). This also contradicts the findings from the previous courses. The correlation between making mistakes in Java and the confidence in our modeling languages might point to the benefits of DSLs over GPLs regarding the better comprehension, smaller complexity, and lower notational noise [Wil01] of models.

In the beginning of this course, the SmartSoft modules were revised more often (4.55 times) than in the end (2.80 times). This correlates with the observation that the students put more effort in understanding SmartSoft initially and, during this, revised the modules more often. All other constituents were refined more often in the beginning than afterwards. Another remarkable observation is that the students considered AUTOMATA (2.33 points out of 10 points) less complex than C&C modeling and Java (both 3.00 points), and SmartSoft (7.33 points). This changed towards the end into considering C&C modeling (3.83 points) less complex than developing Java implementations (4.25 points), modeling AUTOMATA (4.67 points), and developing SmartSoft modules (8.75 points). It is, however, consistent with the previous lab course that the students' complexity estimations increased over time.

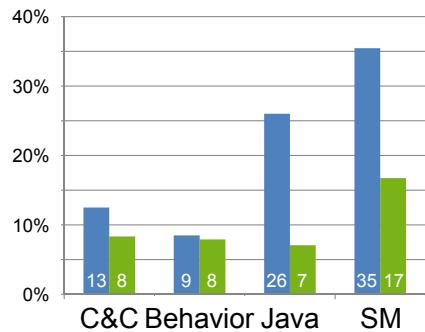
The fact that the students participating in this lab course had background in different computer science bachelor and master courses may be one reason for the different evaluation results compared to the previous lab courses. It is remarkable that the students' confidence in the artifacts created by their colleagues decreased, while the confidence in artifacts produced by themselves increased (see Section B.3 of the appendix). Nonetheless, this lab course also showed that the students grasped the concepts of MontiArcAutomaton and afterwards increased their modeling activities, which reinforces the assumption that MontiArcAutomaton is useful to facilitate the model-driven engineering of complex systems.

Regarding the applicability of Scrum, the students were less confident that it facilitated development than the students of the previous course. Nonetheless, the sprint planning meetings and daily Scrum meetings were considered most useful again.

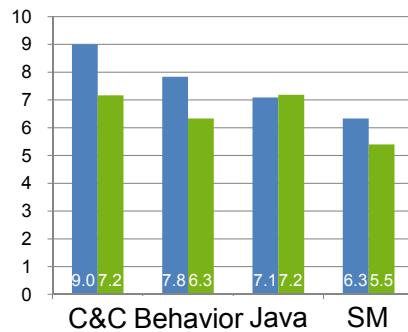
Threats to Validity

Both internal and external validity of this evaluation are threatened by its university context. Announcing the lab course as model-driven robotics software development and the students expecting grades gave rise to multiple biases. Furthermore, the complexity of the development effort prohibited separating the students into a control group and an experimental group. While the students' different backgrounds reduced the selection bias, all participating students have at least taken three semesters in a bachelor's computer science course. While this clearly is a bias, the modeling languages applied in this course are designed for software engineers, such that the selection bias should not impact this evaluation as much.

The threats to the evaluation's internal validity (causality) arise from the employing questionnaires (subject to statistical regression) and the lack of a control group. Both should be improved in future lab courses. Improving the former requires means to evaluate the students' experience with the participating modeling languages other



(a) What percentage (0% - 100%) of the time was wasted because you tried something that was *conceptually* wrong?



(b) Rate your confidence in the correctness of the artifacts created by your team members from 1 (no confidence) - 10 (works perfectly).

Figure 9.9: The students spent less time with conceptually wrong approaches to any of the development tools, but lost confidence in most of their colleagues' artifacts over time.

than interviews and questionnaires. The evolution's external validity (generalizability) is threatened by the students expecting grades, the expectation that model-driven approaches are taught because they are superior to GPL programming, and of course the group composition. Treating the latter requires a lab course big enough to separate into two groups where both groups' participants have similar background regarding GPL and MDE expertise. Unfortunately, this is hardly realizable in university contexts.

9.2 Case Studies

We conducted multiple case studies in different contexts to assess various features of the MontiArcAutomaton infrastructure. These case studies usually were performed by software engineering experts and took place either in academic or industrial contexts.

9.2.1 Lego NXT Distributed Toast Service

We examined MontiArcAutomaton with its behavior language integration and code generator composition mechanisms with a behavior language specific to the robot arm depicted in Figure 9.10. The arm's behavior is controlled by ROBOTARM language models that describe positions in the depicted arm's joint space and programs over these positions as presented in [RRRW14, RRW15a]. This language is embedded into components of a software architecture distributed over two Lego NXT bricks to control a total of three motors. After inserting a slice of toast into the toaster, the arm controlling brick invokes the second brick to start the motor pulling the toaster's lever. After a certain duration, the toast is retrieved from the toaster and delivered to a nearby plate.⁷ This case

⁷A video of the system in action is available from <https://www.youtube.com/watch?v=5EggJHtTg0c>



Figure 9.10: The toast service setup uses two Lego NXTs controlled by MontiArcAutomaton with embedded ROBOTARM programs to steer an arm with three degrees of freedom and a motor to pull the toaster's lever.

study's setup also employed a platform-specific software architecture and used the code generator composition features of MontiArcAutomaton to integrate a code generator for translation of ROBOTARM programs to Java artifacts.

The software architecture of the system was modeled using platform-specific components (cf. Req. *MRQ-2*) and translated into Java implementations compatible to the LeJOS⁸ JVM. Embedding the ROBOTARM component behavior language into the component controlling the Lego robot arm allowed to use a more specific and concise language than Java to control the arm (cf. Req. *MRQ-4*). With MontiArcAutomaton's code generator composition framework, we could reuse the Java code generator for AUTOMATA models without modifications (cf. Requirements *TRQ-3* and *TRQ-6*). Integration actually embedded the ROBOTARM parts considering locations in the arm's joint space and individual ROBOTARM programs only (cf. Req. *MRQ-5*) and required less effort than integrating the AUTOMATA language (Section 5.5.1) as less adapters were required.

9.2.2 Multi-Platform BumperBot

We also have evaluated a variant of the ImprovedBumperBot (as depicted in Figure 4.1) software architecture with Lego NXT robots using LeJOS and the Python

⁸LeJOS website: <http://www.lejos.org/>

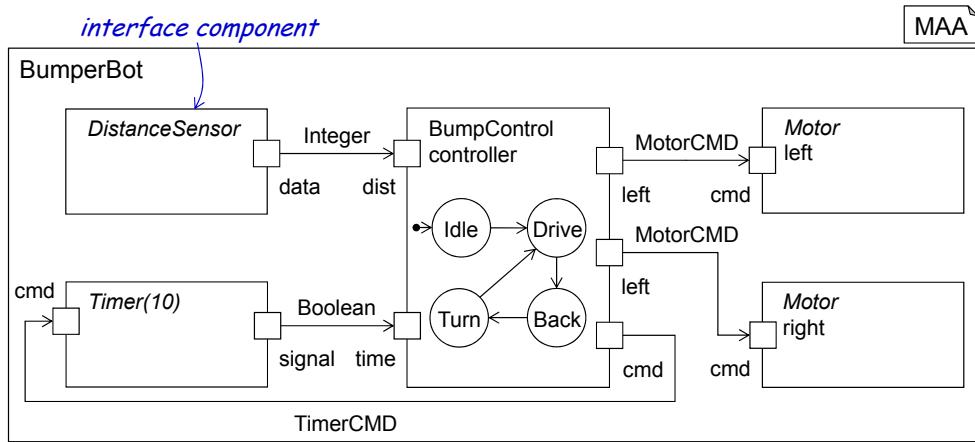


Figure 9.11: Platform-independent software architecture of the BumperBot system reused with platforms relying on different GPLs.

implementation of ROS. The BumperBot variant depicted in Figure 9.11 employs a similar structure to detect and avoid obstacles but, relies on a subcomponent of type `BumpControl` to describe its behavior, which directly communicates with both `Motor` instances. The software architecture is completely platform-independent (cf. Req. *MRQ-1*) and employs interface components to declare extension points for sensors and actuators. It also platform-independent class diagram data types (cf. Req. *MRQ-7*) and AUTOMATA models to describe the behavior of component type `BumpControl` and was translated into two platform-specific architectures using bindings (cf. Req. *MRQ-8*) and the interface libraries and implementation libraries presented in Section 6.2. These software architectures were translated into Java artifacts for the NXT robot using LeJOS, and into Python ROS nodes with corresponding configuration artifacts.

Reusing the platform-independent architecture required 4 interface components as extension points and 4 platform-specific components per platform. The bindings replaced the interface subcomponents. The code generator families for Java systems and ROS-Python systems (Section 7.4) translated the resulting architectures into GPL artifacts.

Listing 9.1 shows an excerpt of the Java code produced for the composed component `BumpControl`, which is a class `BumpControl` that implements the `Component` interface (l. 1) of the `javats` run-time environment (Section 7.4.1). The class yields a member of RTE class `Port` (cf. Figure 7.10) for each port of the component model (ll. 4-8) and a member of RTE interface type `Computable`, which it delegates behavior computation to (l. 11). The actual implementation of this member is produced by the Automatats code generator (Section 7.4.1). The class also contains two members implementing the `Input` and `Result` interfaces of the `javats` RTE, which wrap data of incoming and outgoing ports and variables (ll. 14-15). The method `compute()` of interface `Component` (ll. 17-30) saves the current input values, invokes computation of the behavior delegate, and saves the resulting values to its ports and variables. Whenever the component is invoked to communicate, it emits these values.

```

1 public class BumpControl implements Component {
2
3     // Ports
4     protected Port<Integer> distance;
5     protected Port<TimerSignal> signal;
6     protected Port<TimerCmd> timer;
7     protected Port<MotorCommand> right;
8     protected Port<MotorCommand> left;
9
10    // Behavior implementation
11    protected Computable delegate;
12
13    // Wrappers for port and variable values
14    private BumpControlInput input;
15    private BumpControlResult result;
16
17    public void compute() {
18        // Create data structure of incoming port values
19        input = new BumpControlInput(
20            distance.getCurrentValue(),
21            signal.getCurrentValue());
22
23        // Perform calculations
24        result = delegate.compute(input);
25
26        // Assign computation result to ports
27        timer.setNextValue(result.getTimer());
28        right.setNextValue(result.getRight());
29        left.setNextValue(result.getLeft());
30    }
31
32    // Additional methods, e.g., initialize() and update()
33 }
```

Java

Listing 9.1: An excerpt of the Java implementation generated for the component type BumpControl with its most important members and methods.

Overall, the Java implementation consists of 33 classes, of which 30 are implementations, component factories, implementation factories, input wrappers, and output wrappers of the respective component types. The remaining three classes are the starter of BumperBot and the data types MotorCMD and TimerCMD. An excerpt of the Python implementation of BumpControl is depicted in Listing 9.2. As Python does not support interfaces, the Python class BumpControl extends the abstract class Component of the pythonts run-time environment (Section 7.4.2). Its constructor (ll. 3-16) initializes a ROS node with the name bumperbot (l. 4). If ROS is running, it creates a member for each port (ll. 6-15). For incoming ports, it initializes a ROS Subscriber

with each port’s name, data type, and pointer to a callback function that should be executed whenever new data arrives for that port (ll. 7-10). For outgoing ports, it initializes a ROS Publisher with each port’s name and data type (ll. 11-13). This publisher allows sending messages of the port’s type to the corresponding topic. The Python component implementations also rely on delegation to compute component behavior and consequently, the implementation component type `BumpControl` owns a member delegate produced by the `BumpControlImplFactory` (l. 15).

Afterwards, the `BumpControl` class defines that whenever its `distanceCallback` method (assigned as callback to incoming port `distance`) is invoked with a new ROS message `msg`, the payload part of this message is assigned to the port `distance`. Similarly, the method `leftPublish()` employs a ROS message of type `MotorCommandMSG` to wrap the current value of port `left` and publishes it to a ROS topic.

The pythonts run-time environment also imposes a method `compute()` on components. For the Python implementation component type `BumpControl`, this method (ll. 26-38) behaves similarly to the Java implementation: after collecting the current input values from ports `distance` and `signal`, it passes these to its behavior implementation delegate (l. 33) and assigns the results back to its ports (ll. 38-38).

As the Python implementation of `BumperBot` employs similar patterns, it is little surprising that it produces a similar amount of Python classes. However, ROS does not support composed nodes, and hence, there is no implementation for the composed component `BumperBot`, but only a so-called launch file that describes how to start each of its subcomponents as a ROS node. To interface with ROS, the publisher and subscriber instances rely on ROS messages (a simple data type description language), which were created from the class diagram data types `MotorCMD` and `TimerCMD` accordingly.

9.2.3 The iserveU Hospital Logistics Project

The iserveU research project was conducted with partners from industry and academia to investigate pervasive mode-driven engineering for complex software systems. The industrial partners were the Robert Bosch GmbH, the Robotics Equipment Corporation (REC), and the Symeo GmbH. Academic expertise was contributed by Friedrich-Alexander-Universität Erlangen-Nürnberg, Clausthal University of Technology, RWTH Aachen University, and Ulm University of Applied Sciences. The project focused on model-driven engineering and deployment of robotics applications for real-world contexts. This 3-year project was funded by the German Federal Ministry of Education and Research (BMBF) to investigate pervasive model-driven engineering of service robotics applications. Here, we used MontiArcAutomaton to model parts of architecture and behavior for a logistics service robotics application deployed to a complex hospital environment. In this project, MontiArcAutomaton served as ADL for a high-level controller [HNR⁺15] that interfaces SmartSoft [SSL11] and all development roles were distributed among the consortium.

Figure 9.12 shows the core component `BehaviorController` of the iserveU top-level software architecture. This component contains six subcomponents of the types `TaskManager`, `ActionExecuter`, `StateProvider`, `Planner`, `PlanVerifier`, and

```

1 class BumpControl(Component):
2
3     def __init__(self):
4         rospy.init_node('bumperbot')
5         if not rospy.is_shutdown():
6             # Ports are connected to ROS topics
7             self.distance = Port()
8             rospy.Subscriber("~distance",
9                             IntegerMSG,
10                            self.distanceCallback)
11            self.left = Port()
12            self.leftPub = rospy.Publisher("~left",
13                                         MotorCommandMSG)
14            # Behavior implementation
15            self.delegate = BumpControlImplFactory.create()
16
17    def distanceCallback(self, msg):
18        self.distance.setNextValue(msg.data)
19
20    def leftPublish(self):
21        value = self.left.getCurrentValue()
22        msg = MotorCommandMSG()
23        msg.data = numpy.uint8(value)
24        self.leftPub.publish(msg)
25
26    def compute(self):
27        # Create data structure of incoming port values
28        input = BumpControlInput(
29            self.distance.getCurrentValue(),
30            self.signal.getCurrentValue())
31
32        # Perform calculations
33        result = self.delegate.compute(input)
34
35        # Assign computation result to ports
36        self.timer.setNextValue(result.getTimer())
37        self.right.setNextValue(result.getRight())
38        self.left.setNextValue(result.getLeft())

```

Python

Listing 9.2: An excerpt of the Python implementation generated for the component type BumpControl with its most important members and methods.

PropertyCalculator. The TaskManager subcomponent receives tasks and decomposes these into individual goals that are passed to the component task Planner. The latter is capable of reasoning about the goals using the Metric-FF planner [Hof02] to reach goals based on the current situation and the actions available to the robot. To deduct a valid plan it may access properties of the robot and its environment

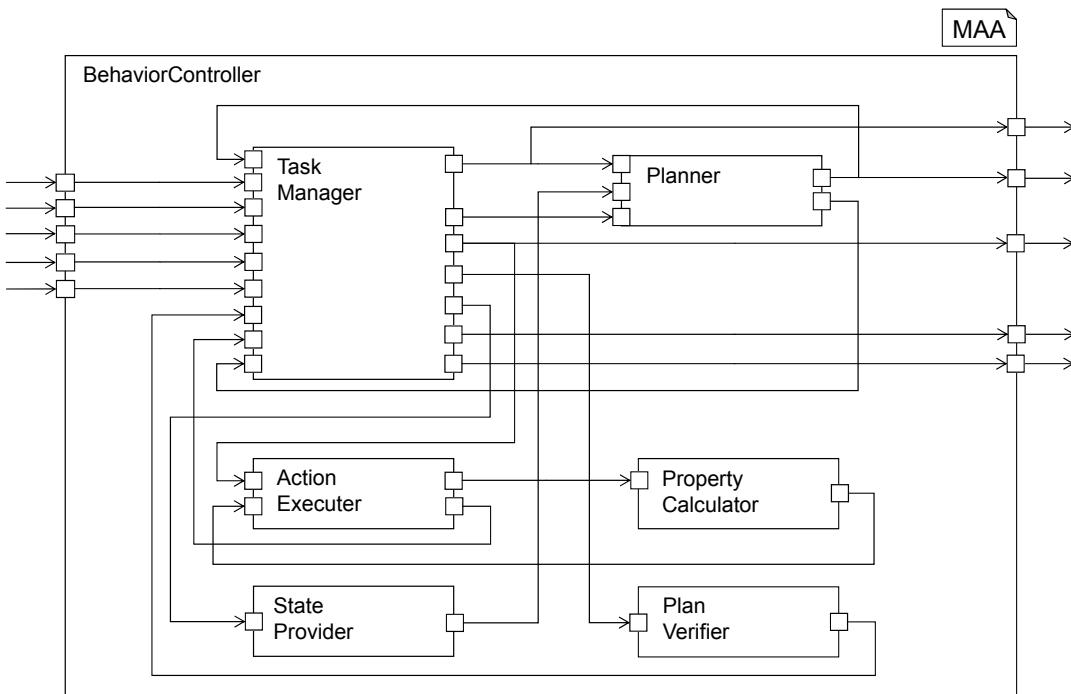


Figure 9.12: The core software architecture of the high-level robot controller employed in the iserveU project.

via `StateProvider`. Once a plan is deduced and verified by `PlanVerifier`, the controller executes the plan's actions via `ActionExecuter`. The latter utilizes the `PropertyCalculator` to check whether the environment has remained stable enough during planning, such that the plan, and hence the action to be executed, is still valid.

Afterwards, the action is translated properly and sent to the underlying SmartSoft middleware via handcrafted GPL behavior implementations (cf. Req. TRQ-2). All component types depicted are composed from multiple subcomponents and in total, the software architecture employs 23 component types and instantiates 57 subcomponents.

Applying MontiArcAutomaton in a context with multiple stakeholders from industry and academia has shown that it scales to industrial requirements. Although discussions pointed out that the embedded AUTOMATA language is insufficient for various purposes (such as describing behavior of components interfacing system APIs), the ability to easily develop and integrate the most appropriate behavior languages redeems for that.

9.3 Discussion

All three lab courses were conducted in a university environment, which entails various threats to their validity. Most students had prior training in Java and object-oriented concepts, but only little experience in modeling techniques or state-based description formalisms. Furthermore, the students were graded, hence there was an intrinsic moti-

vation to accept the modeling tools presented by the lab course organizers. Also, due to the nature of this thesis, the lab courses conducted earlier were performed with less sophisticated versions of this infrastructure, which hampers their comparability. Future evaluations should also separate the participants into a test group employing the modeling techniques and into a control group employing GPLs. However, due to the size of lab courses of RWTH Aachen, this was not feasible.

The case studies were conducted with students, research assistants, and industrial partners and examined more complex infrastructure features than the evaluations. As these were of a more experimental nature, the results are merely observations from which no generalizable conclusions regarding the benefits of modeling over programming can be drawn. Nevertheless, they help to illustrate the features of MontiArcAutomaton.

Chapter 10

Conclusions and Future Work

*Better a little which is well done,
than a great deal imperfectly.*

Plato

The concepts presented in this thesis combine software language engineering with architecture modeling to enable pervasive model-driven engineering with an ADL that can be adjusted to specific requirements using the most appropriate DSLs and code generators. The insights gained are presented in 15 publications discussing individual aspects of MontiArcAutomaton at different development stages (Section 1.4). The presented state of MontiArcAutomaton combines these contributions and contributes improvement of various aspects as well as their integration.

This chapter concludes the thesis. First, Section 10.1 discusses its contributions relative to its goals as presented in Section 1.2. Afterwards, Section 10.2 sketches future work, before Section 10.3 concludes with a discussion of overall observations that span multiple concerns and parts of MontiArcAutomaton.

10.1 Contributions

We presented concepts for the development of complex, multi-platform software systems as C&C software architectures with exchangeable behavior languages. To this effect, we introduced a concept for the integration of component behavior modeling languages into a C&C ADL, which facilitates contribution by domain experts and language reuse. We also presented a concept for the development of platform-independent software architectures and their translation into platform-specific architectures, which relies on designated architecture extension points, model transformations, and code generator composition. These concepts are realized in the MontiArcAutomaton architecture modeling infrastructure that enables reusing component behavior languages, components, complete architectures, and code generators in different contexts and applications.

Concepts and realization are based on the requirements raised in Section 3.2. The basis to fulfilling these requirements is the extensible MontiArcAutomaton ADL, which supports modeling of platform-independent and platform-specific software architectures (Chapter 4) with extensible component behavior modeling languages. Off-the-shelf, MontiArcAutomaton provides the AUTOMATA language (Chapter 5) to model component

behavior platform-independently. Based on the MontiArcAutomaton ADL, a concept to derive platform-specific architecture models utilizing a distinction between platform-independent components, interface components, and platform-specific components is presented (Chapter 6). This concept relies on interface components, interface libraries and implementation libraries to maximize reuse in contexts with multiple target platforms. It uses bindings to describe replacement of interface components with platform-specific components and application configuration models to define these bindings. Defining bindings requires little effort as their syntax resembles the MontiArcAutomaton ADL's syntax and the platform-specific components are required without bindings as well. Most additional effort rises from developing interface components, which reuse the vocabulary of the MontiArcAutomaton ADL and can be developed with little effort.

Producing platform-specific implementations from the resulting software architectures requires code generators capable to translate architecture with exchangeable component behavior languages into GPL artifacts. Therefore, concepts and realization of a compositional code generation framework are presented in Chapter 7, which support black-box code generator composition and, thus, effortless combination of required code generators. This framework exploits the C&C nature of MontiArcAutomaton and identifies three code generator kinds for composition. Participating generators implement one of these types and explicate this via generator configuration models from which MontiArcAutomaton produces composable implementations.

MontiArcAutomaton ADL, bindings, and code generators are combined into a compact tool chain that can be used out-of-the-box or extended with new behavior modeling languages and code generators as required (Chapter 3). Different parts of this infrastructure were evaluated in different contexts throughout its development (Chapter 9) and results have been integrated subsequently.

Although the implementation of MontiArcAutomaton fulfills all requirements identified initially Section 3.2, the infrastructure comprises many modules that can be configured. This enables various forms of mis-configuration, ultimately leading to development issues, hindrances, and failures. However, off-the-shelf MontiArcAutomaton does not require configuration if developing platform-specific software architectures with target GPL Java as it already contains the AUTOMATA behavior language and the javats code generator family. Generating to different target platforms and GPLs requires proper code generators. If these are available, their selection requires only specifying their qualified names in an application configuration model - which requires only very little effort. The effort required to make the code generators under development compositional consists of providing a generator description model and invoking the corresponding generator - which is negligible as well. Integration of new behavior languages usually requires the application modeler to provide a simple language configuration model only. For languages with special integration requirements, MontiArcAutomaton supports to harness the full power of the MontiCore language workbench and provide means to specify the complete integration information in a single artifact.

Evaluations and case studies have shown that MontiArcAutomaton is suitable to develop complex applications with various component behavior modeling languages. The

MontiArcAutomaton ADL is an accessible architecture description language and integration of other languages enables modeling great parts of software architectures. Throughout the different projects, we could reuse components, complete architectures, modeling languages, and code generators with little effort.

The findings and insights gained in this thesis are generalizable to other modeling scenarios with varying degree depending on the employed architecture description languages and frameworks. Usually, integration of behavior languages into ADLs is either not anticipated or overly complicated. However, ADLs build on top of language workbenches generally could support some form of behavior language integration (via embedding, inheritance, or AST composition). The notion of bindings translates naturally to ADLs that distinguish component interfaces from component implementations, and hence, the binding transformation and the corresponding library concept can be translated to replacing component implementations instead. Similarly, the notions of platform-independent interface libraries and platform-specific implementation libraries could be translated. Most ADL modeling frameworks do not consider code generation for multiple target platforms a prime concern. Hence, they usually support only a fixed – and small – set of target GPLs. For frameworks employing language workbenches that consider code generation, development of new code generators might be feasible. To the best of our knowledge, code generator composition is not considered by any ADL modeling infrastructure. However, in contexts where the generator kinds are fixed apriori, applying the presented mechanisms is feasible as well.

10.2 Potential for Future Research

Although the presented MontiArcAutomaton infrastructure satisfies its goals regarding the development of multi-platform applications with little effort, future research could improve the MontiArcAutomaton ADL and other constituents. Previous sections have presented future work regarding MontiArcAutomaton ADL (Section 4.3), bindings (Section 6.4), and code generation (Section 7.5).

Aside from enhancements argued in those sections, the MontiArcAutomaton tool support could be improved. While there is a MontiArcAutomaton plugin¹ for the Eclipse IDE, this does not support configuration besides editors for the MontiArcAutomaton modeling languages. Also, for a previous version of MontiArcAutomaton, it supported visualization of components with automata [RRW13b]. For generic visualization of components with arbitrary behavior languages, language embedding must be reflected by integration of the respective languages’ editors and visualizers.

A central benefit in model-driven engineering is that abstract models are better amendable to analyses than GPL artifacts. The MontiArcAutomaton modeling languages implement various analyses regarding the well-formedness of models. There are many other analyses that could improve engineering with MontiArcAutomaton, such as refinement of components to ensure proper architecture evolution [Rin14], verification of structural

¹Available via <http://www.monticore.de/robotics/montiarcautomaton/>

and behavior architecture properties, or compliance of component implementations to the run-time environment the corresponding models specify.

Validating the integration of model processing infrastructure, models transformations, and code generators require inter-framework well-formedness checking. The context conditions checking whether exactly one code generator per behavior language is used is an effort into this direction. Nonetheless, there are many inter-framework relations and properties not yet validated. This includes checking whether the modeling languages' semantics match the generated artifacts semantics, whether new workflows and transformation produce artifacts processable by subsequently executed infrastructure parts, and whether the code generators produce artifacts conforming to the run-time environments.

Another issue in model-driven engineering with complex tool chains arises from metamodel evolution. If the grammar of any participating language changes, various parts of the infrastructure need to evolve as well. While research in metamodel evolution itself has produced some results [IK04, GG07], pervasive change tracing from metamodels to transformations to code generation has not.

Future work also requires evolution of MontiArcAutomaton in industrial-scale contexts with multiple target-platforms. As MontiArcAutomaton also aims at architecture modeling in academic contexts, further evaluation in this direction is required as well.

10.3 Conclusion

The research presented in this thesis aims to enable seamless model-driven multi-platform engineering with component & connector software architectures that support flexible integration of component behavior languages. This facilitates contribution by domain experts and increases reuse of components, software architectures, modeling languages, and code generators.

The MontiArcAutomaton ADL is an extensible C&C architecture description language that can express the most important elements common to C&C architecture without introducing too much notational noise [Wil01]. While its expressiveness is limited compared to industrial ADLs, it reduces the risk of accidental complexities [FR07]. Nonetheless, built upon the powerful language workbench MontiCore, it allows the extension and tailoring deemed crucial [MDT07, WHR14] for professional application. A unique feature of MontiArcAutomaton is its non-invasive extensibility with component behavior languages, which employs the language integration mechanisms of MontiCore. Language integration in MontiArcAutomaton is supported by a concise DSL, which reduces its complexity severely. With this, application modelers can develop platform-specific software architectures with the most-appropriate modeling languages easily.

Our approach to multi-platform reuse is of minimal effort in terms of new concepts and requires only interface components and application configuration models. It exploits being tailored to the MontiArcAutomaton ADL by reusing its instantiation syntax and enforces reuse by relying on explicit interface libraries and implementation libraries. In contrast to generic model transformation mechanisms [Sch91, FNTZ00, JABK08] or delta-modeling approaches [HKR⁺11, HRRS11], it is specific to replacing component

types and parameters of instances. This minimizes its complexity and is less error prone. Thus, application modelers targeting multiple platforms can engineer such applications with minimal effort.

The code generator composition framework of MontiArcAutomaton is built upon the code generation framework of MontiCore and reuses its concepts. Nonetheless, its composition is decoupled from templates and transformations, but relies only on generator kinds derived from the C&C nature of MontiArcAutomaton. Conformance to these types is expressed by generator description models providing information required by the respective types. MontiArcAutomaton transforms these models into implementations implementing the explicated interfaces. These implementations are exploited for code generator composition.

Finally, the application configuration language expresses C&C applications as combination of their constituents. It therefore integrates a single platform-independent software architecture with implementation libraries, bindings, and code generators to define platform-specific implementations. Based on these models, MontiArcAutomaton processes the corresponding artifacts and produces implementation artifacts. The separation of concerns into different modeling languages and artifacts enables reusing components with different architectures, complete architectures with different target platforms, component behavior languages within different language combinations, and code generators as required. We believe that combining software architecture modeling with software language engineering is a promising area and that our research produces useful results for the flexible model-driven engineering of C&C systems with domain-specific languages.

Bibliography

- [ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 187–197. IEEE, 2002. 1.1, 2.3, 4.3, 4.4, 8.4
- [ADvSP05] João Paulo Almeida, Remco Dijkman, Marten van Sinderen, and Luís Ferreira Pires. Platform-Independent Modelling in MDA: Supporting Abstract Platforms. In Uwe Aßmann, Mehmet Aksit, and Arend Rensink, editors, *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2005. 6.4
- [ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. 2.2
- [ASH⁺12] Andreas Angerer, Remi Smirra, Alwin Hoffmann, Andreas Schierl, Michael Vistein, and Wolfgang Reif. A Graphical Language for Real-Time Critical Robot Commands. In *Proceedings of the Third International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2012)*, 2012. 1.1, 4.4
- [ASK⁺05] Noriaki Ando, Takashi Suehiro, Kosei Kitagaki, Tetsuo Kotoku, and Woo-Keun Yoon. RT-Middleware: Distributed Component Middleware for RT (Robot Technology). In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 3933–3938, 2005. 1.1, 2.3, 4.4
- [ASK08] Noriaki Ando, Takashi Suehiro, and Tetsuo Kotoku. A Software Platform for Component Based RT-System Development: OpenRTM-Aist. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 87–98. Springer, 2008. 4.4
- [ASM04] Timo Asikainen, Timo Soininen, and Tomi Männistö. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. In *Software Product-Family Engineering*, pages 225–249. Springer, 2004. 2.3
- [AZ05] Paris Avgeriou and Uwe Zdun. Architectural Patterns Revisited - A Pattern Language. In *10th European Conference on Pattern Languages*

BIBLIOGRAPHY

- of Programs (EuroPlop 2005), Irsee.* Universitaetsverlag Konstanz, 2005.
2.2.3
- [Bal00] Helmut Balzert. Software-Entwicklung (Lehrbuch der Software-Technik, Band 1), 2000. 2.1
- [BBC⁺07] Davide Brugali, Alex Brooks, Anthony Cowley, Carle Côté, AntonioC. Domínguez-Brito, Dominic Létourneau, François Michaud, and Christian Schlegel. Trends in Component-Based Robotics. In Davide Brugali, editor, *Software Engineering for Experimental Robotics*, volume 30 of *Springer Tracts in Advanced Robotics*, chapter 8, pages 135–142. Springer Berlin Heidelberg, 2007. 1.1, 2.3, 4.4
- [BBH13] Johannes Baumgartl, Thomas Buchmann, and Dominik Henrich. Towards Easy Robot Programming: Using DSLs, Code Generators and Software Product Lines. *8th International Conference on Software Paradigm Trends (ICSOFT-PT'13)*, 2013. 1.1, 4.4
- [BCL⁺06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Software, Practice, and Experience*, 36(11-12):1257–1284, 2006. 2.3, 4.3, 6.1, 8.4
- [BCOR15] Jean-Michel Bruel, Benoit Combemale, Ileana Ober, and Hélène Raynal. MDE in Practice for Computational Science. In *International Conference on Computational Science (ICCS 2015)*, Reykjavík, Iceland, June 2015. 2.1
- [BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-Driven Software Engineering in Practice. *Synthesis Lectures on Software Engineering*, 2012. 2.1
- [BD99] Bernd Bruegge and Allen A Dutoit. *Object Oriented Software Engineering, Conquering Complex and Changing Systems*. Prentice Hall, 1999. 2.1
- [BDC02] Marco Bernardo, Lorenzo Donatiello, and Paolo Ciancarini. Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language. In *Performance Evaluation of Complex Systems: Techniques and Tools*, pages 236–260. Springer, 2002. 1.1
- [BDD⁺93] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The Design of Distributed Systems - An Introduction to FOCUS. Technical report, TUM-I9202, SFB-Bericht Nr. 342/2-2/92 A, 1993. 2.3

- [BDHN10] Jean-Christophe Baillie, Akim Demaille, Quentin Hocquet, and Matthieu Nottale. Events! (Reactivity in urbiscript). In *First International Workshop on Domain-Specific Languages and Models for ROBotic Systems*, October 2010. 1.1, 4.4
- [Bea09] David M. Beazley. *Python Essential Reference (4th Edition)*. Addison-Wesley Professional, 2009. 7.4.2
- [Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013. 2.2.4
- [BFBFR07] Ricardo Bedin Franca, Jean-Paul Bodeveix, Mamoun Filali, and Jean-Francois Rolland. The AADL behavior annex-experiments and roadmap. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems. Washington, DC: IEEE Computer Society*, pages 377–382, 2007. 2.3, 4.4
- [BG01] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE, 2001. 2.1
- [BGBK08] Simon Barner, Michael Geisinger, Christian Buckl, and Alois Knoll. EASYLab: Model-Based Development of Software for Mechatronic Systems. *2008 IEEE/ASME International Conference on Mechtronic and Embedded Systems and Applications*, pages 540–545, October 2008. 1.1, 8.4
- [BGM10] Barrett R. Bryant, Jeff Gray, and Marjan Mernik. Domain-Specific Software Engineering. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 65–68. ACM, 2010. 2.3
- [BGP⁺10] Rainer Bischoff, Tim Guhl, Erwin Prassler, Walter Nowak, Gerhard Kraetzschmar, Herman Bruyninckx, Peter Soetens, Martin Haegele, Andreas Pott, Peter Breedveld, et al. BRICS – Best practice in robotics. In *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)*, pages 1–8. VDE, 2010. 6.4
- [BGT05] Sven Burmester, Holger Giese, and Matthias Tichy. Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In *Model Driven Architecture*, pages 47–61. Springer, 2005. 2.3
- [BHH02] M Brian Blake, Gail Hamilton, and Jeffrey Hoyt. Using Component-Based Development and Web Technologies to Support a Distributed Data Management System. *Annals of Software Engineering*, 13(1-4):13–34, 2002. 2.3

BIBLIOGRAPHY

- [BHS99] Manfred Broy, Franz Huber, and Bernhard Schätz. AutoFOCUS – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik-Forschung und Entwicklung*, 14(3):121–134, 1999. 2.3, 4.4
- [BKH⁺13] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. The BRICS Component Model: A Model-Based Development Paradigm For Complex Robotics Software Systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC ’13, pages 1758–1764, New York, NY, USA, 2013. ACM. 1.1, 4.3, 4.4
- [BKM⁺05] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Orebäck. Towards Component-Based Robotics. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 163–168. IEEE, 2005. 4.4
- [Blu13] Blumenthal, Sebastian and Bruyninckx, Herman. Towards a Domain Specific Language for a Scene Graph based Robotic World Model. In *Fourth International Workshop on Domain-Specific Languages and Models for ROBOTIC Systems*, November 2013. 4.4
- [BO92] Don Batory and Sean O’Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992. 7.5
- [BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik Spektrum*, 30(1):3–18, 2007. 2.4, 5.6
- [Bru01] Herman Bruyninckx. Open Robot Control Software: the OROCOS project. In *2001 ICRA IEEE International Conference on Robotics and Automation (ICRA)*, volume 3, pages 2523–2528. IEEE, 2001. 1.1, 2.3, 4.4
- [BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001. 2.3, 2.4, 4.4, 5.6
- [BS06] Davide Brugali and Paolo Salvaneschi. Stable Aspects In Robot Software Development. *International Journal of Advanced Robotic Systems*, 3, 2006. 1.1, 2.3
- [BST⁺94] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca Model of Software-System Generators. *IEEE Software*, 11(5):89–94, September 1994. 7.5

- [CCF⁺15a] Betty H.C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe. On the Globalization of Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, volume 9400 of *LNCS*. Springer International Publishing, 2015. 4.3
- [CCF⁺15b] Betty H.C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*. Springer, 2015. 4.2.1
- [CCIN08] Daniele Calisi, Andrea Censi, Luca Iocchi, and Daniele Nardi. OpenRDK: a modular framework for robotic software development. In *008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1872–1877. IEEE, 2008. 4.4
- [CDB⁺14] Benoit Combemale, Julien DeAntoni, Benoit Baudry, Robert B. France, Jean-Marc. Jézequél, and Jeff Gray. Globalizing modeling languages. *Computer*, 47(6):68–71, June 2014. 4.3
- [CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the OOPSLA ’03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*, Anaheim, California, USA, 2003. 2.1
- [CKS11] Damien Cassou, Pierrick Koch, and Serge Stinckwich. Using the DiaSpec design language and compiler to develop robotics systems. In *Proceedings of the Second International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2011)*, 2011. 1.1, 4.4, 6.4
- [CoEEB90] IEEE Computer Society. Standards Coordinating Committee, Institute of Electrical, Electronics Engineers, and IEEE Standards Board. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Standards. IEEE, 1990. 2.1
- [CVdBCR15] Tony Clark, Mark Van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual model of the globalization for domain-specific languages. In *Globalizing Domain-Specific Languages*, pages 7–20. Springer, 2015. 4.2.1
- [DD96] Christopher John Date and Hugh Darwen. *A Guide to SQL Standard*. Addison-Wesley Professional, 1996. 2.1, 2.2.2
- [Deu14] Yannick Deuster. *Entwicklung eines Generators für Python Code aus UML/P Klassendiagrammen*. Bachelor’s thesis, RWTH Aachen, 2014. 7.4.2
- [DKS⁺12] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. RobotML, a Domain-Specific Language to Design, Simulate and

BIBLIOGRAPHY

- Deploy Robotic Applications. In Itsuki Noda, Noriaki Ando, Davide Brugali, and James J. Kuffner, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 7628 of *Lecture Notes in Computer Science*, pages 149–160. Springer Berlin Heidelberg, 2012. 1.1, 4.4, 6.4, 8.4
- [DSLT05] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. An Architecture Description Language. In *Architecture Description Languages*, pages 181–195. Springer, 2005. 2.3, 8.4
- [DVdHT01] Eric M. Dashofy, André Van der Hoek, and Richard N. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *Working IEEE/IFIP Conference on Software Architecture*, pages 103–112. IEEE, 2001. 2.3, 4.4
- [DVdHT02] Eric M. Dashofy, Andre Van der Hoek, and Richard N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *24rd International Conference on Software Engineering (ICSE 2002)*, pages 266–276. IEEE, 2002. 4.4
- [EB10] Moritz Eysholdt and Heiko Behrens. Xtext - Implement your Language Faster than the Quick and Dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH ’10, pages 307–309, New York, NY, USA, 2010. ACM. 2.2.4
- [EGR12] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA ’12, New York, NY, USA, 2012. ACM. 2.2.2, 4.4
- [EKM98] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Computer-Aided Verification, (CAV ’98)*, volume 1427 of *LNCS*, pages 516–520. Springer-Verlag, 1998. 9
- [ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based Syntactic Language Extensibility. In *ACM SIGPLAN Notices*, 2011. 2.2.4
- [EvdSV⁺13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D.P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Clemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches. In Martin Erwig, Richard F. Paige, and Eric Van Wyk,

- editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer International Publishing, 2013. 2.1, 2.1, 2.2.4
- [FBC11] Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell. A Domain Specific Language for kinematic models and fast implementations of robot dynamics algorithms. In *Proceedings of the Second International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2011)*, 2011. 1.1, 4.4
- [FG07] Lidia Fuentes and Nadia Gámez. Adding Aspects to xADL 2.0 for Software Product Line Architectures. In *First International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 87–96, 2007. 4.3
- [FG12] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012. 1.1, 2.3, 4.3, 4.4, 6.1, 6.4, 8.4
- [FMN08] Paul Fitzpatrick, Giorgio Metta, and Lorenzo Natale. Towards long-lived robot genes. *Robotics and Autonomous systems*, 56(1):29–45, 2008. 4.4
- [FMS11] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, 2011. 2.3, 4.4
- [FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer Berlin Heidelberg, 2000. 10.3
- [Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010. 1.1, 2.2.4, 4.2.3, 4.2.3, 4.2.3
- [FR07] Robert France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE*, pages 37–54, 2007. (document), 1.1, 2.1, 4.3, 6, 10.3
- [Fre] Freemarker Website. <http://freemarker.org/>. Accessed: 2016-01-22. 2.2, 2.2.3
- [GBWK09] Michael Geisinger, Simon Barner, Martin Wojtczyk, and Alois Knoll. A Software Architecture for Model-Based Programming of Robot Systems. *Advances in Robotics Research*, pages 135–146, 2009. 6.4

BIBLIOGRAPHY

- [GG07] Ismenia Galvao and Arda Goknil. Survey of Traceability Approaches in Model-Driven Engineering. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 313–313, October 2007. 10.2
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995. 2.2, 2.2.2, 4.2.3, 7.1, 7.4.1, 4, 3, 7.4.2, 7.5, 8.1.3
- [GHK⁺15] Timo Greifenberg, Katrin Hoelldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Mueller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, Bernhard Rümpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, Angers, France, 2015. Scitepress. 1.4, 7.4.1
- [Gho10] Debasish Ghosh. *DSLs in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. 2.2.4, 4.2.3
- [GKR⁺06] Hans Grönniger, Holger Krahn, Bernhard Rümpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Technical Report Informatik-Bericht 2006-04, Software Systems Engineering Institute, Braunschweig University of Technology, 2006. 2.2
- [GKR⁺07] Hans Grönniger, Holger Krahn, Bernhard Rümpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering*, 2007. 2.2
- [GKR⁺08] Hans Grönniger, Holger Krahn, Bernhard Rümpe, Martin Schindler, and Steven Völkel. Monticore: a framework for the development of textual domain specific languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008. 2.1, 2.2
- [Gli02] Martin Glinz. Statecharts For Requirements Specification - As Simple As Possible, As Rich As Needed. In *International Conference on Software Engineering (ICSE) 2002*. ACM Press, 2002. 4.1
- [GMW97] David Garlan, Robert T. Monroe, and David Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997. 1.1, 2.3, 8.4

- [GMW00] David Garlan, Robert T. Monroe, and David Wile. ACME: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, 68:47–68, 2000. 2.3, 4.3, 8.4
- [Hö07] Frank Höwing. Effiziente Entwicklung von AUTOSAR-Komponenten mit domänenspezifischen Programmiersprachen. In *Proceedings of Workshop Automotive Software Engineering*, LNI. Springer, 2007. 4.4
- [HBB⁺94] Wolfgang Hesse, Georg Barkow, H von Braun, Hans-Bernd Kittlaus, and Gert Scheschonk. Terminologie der Softwaretechnik. Ein Begriffssystem für die Analyse und Modellierung von Anwendungssystemen. Teil 2: Tätigkeits- und ergebnisbezogene Elemente. *Informatik Spektrum*, 17(2):96–105, 1994. 2.1
- [HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison Wesley, 2001. 2.3
- [HF11] Florian Hödl and Martin Feilkas. AutoFocus 3-A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 317–322. Springer, 2011. 4.4
- [HGS⁺13] Nico Hochgeschwender, Luca Gherardi, Azamat Shakhirmardanov, Gerhard K Kraetzschmar, Davide Brugali, and Herman Bruyninckx. A model-based approach to software deployment in robotics. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3907–3914. IEEE, 2013. 6.4
- [HHRW15] Lars Hermerschmidt, Katrin Hoelldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp) 2015*, volume 1463 of *CEUR Workshop Proceedings*, pages 18 – 23, Ottawa, Canada, September 2015. 2.1
- [HK00] Charles Herring and Simon Kaplan. Component-Based Software Systems for Smart Environments. *IEEE Personal Communications*, 7(5):60–61, 2000. 2.3
- [HKR⁺11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-Oriented Architectural Variability using MontiCore. In *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. ACM, 2011. 2.4, 10.3
- [HLMSN⁺15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Voelkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Proceedings of the 3rd*

BIBLIOGRAPHY

- International Conference on Model-Driven Engineering and Software Development*, Angers, France, 2015. Scitepress. 1.4, 2.2, 2.2, 2.2.1, 2.2.2, 4.2.2, 4.3, 5.2
- [HNR⁺15] Robert Heim, Pedram Mir Seyed Nazari, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Modeling Robot and World Interfaces for Reusable Tasks. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1793–1798, 2015. 9.2.3
- [HO10] Christian Hofer and Klaus Ostermann. Modular Domain-Specific Language Components in Scala. In *ACM SIGPLAN Notices*, 2010. 2.2.4
- [Hoc13] Hochgeschwender, Nico and Schneider, Sven and Voos, Holger, and Kraetzschmar, Gerhard K. Towards a Robot Perception Specification Language. In *Fourth International Workshop on Domain-Specific Languages and Models for ROBotic Systems*, November 2013. 1.1, 4.4
- [Hof02] Jörg Hoffmann. Extending FF to Numerical State Variables. In *15th European Conference on Artificial Intelligence (ECAI 2002)*, pages 571–575, 2002. 9.2.3
- [HPB11] Jon Holt, Simon Perry, and Mike Brownsword. *Model-Based Requirements Engineering (Iet Professional Applications of Computing)*. The Institution of Engineering and Technology, 2011. 2.1
- [HR04a] Gregoire Hamon and John Rushby. An Operational Semantics for Stateflow. In *Fundamental Approaches to Software Engineering: 7th International Conference (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 229–243, Barcelona, Spain, March 2004. Springer-Verlag. 4.4
- [HR04b] David Harel and Bernhard Rumpe. Meaningful Modeling: What’s the Semantics of “Semantics”? *Computer*, 37(10):64–72, 2004. 2.1
- [HR13] Andreas Horst and Bernhard Rumpe. Towards Compositional Domain Specific Languages. In *Proceedings of the 7th Workshop on Multi-Paradigm Modeling (MPM’13)*, pages 1–5, 2013. 2.2
- [HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VI*, pages 13–22, Munich, Germany, February 2010. fortiss GmbH. 2.4
- [HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen, february 2012. 1, 1.3, 2, 2.1, 2.3, 2.4, 2.4, 3.3, 4, 4.1.4, 4.1.4, 4.1.4, 4.1.4, 4.1.4, 4.1.4, 4.1.4, 4.2.2, 5.1, 6.1, 6.4, A.1.1, 32, A.1.2, E

- [HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, Munich, Germany, February 2011. fortiss GmbH. 10.3
- [HRW11] John Hutchinson, Mark Rouncefield, and John Whittle. Model-Driven Engineering Practices in Industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 633–642, May 2011. 2.1
- [HTM] HTML Specification: <http://www.w3.org/html/wg/html5/>. [Online; accessed 2015-01-02]. 2.1
- [Hud96] Paul Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys (CSUR)*, 28(4):196, 1996. 2.2.4, 4.4
- [Hud98] Paul Hudak. Modular Domain Specific Languages and Tools. In *Fifth International Conference on Software Reuse*, pages 134–142, 1998. 2.2.2
- [IK04] Ivica Ivkovic and Kostas Kontogiannis. Tracing Evolution Changes of Software Artifacts through Model Synchronization. In *20th IEEE International Conference on Software Maintenance*, pages 252–261, September 2004. 10.2
- [Jö13] Sven Jörge. *Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013. 7.5
- [JAB⁺06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006. 2.1
- [JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1–2):31–39, 2008. 10.3
- [JB06] Frederic Jouault and Jean Bezivin. KM3: a DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (LNCS 4037)*, pages 171–185, 2006. 2.1
- [JBCG05] Ackbar Joolia, Thais Batista, Geoff Coulson, and Antonio T.A. Gomes. Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In *5th Working IEEE/IFIP Conference on Software Architecture, 2005. WICSA 2005*, pages 131–140. IEEE, 2005. 1.1, 4.3, 4.4

BIBLIOGRAPHY

- [JBF11] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, chapter Model Driven Language Engineering with Kermeta, pages 201–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 2.2.4
- [JCB⁺13] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and Francois Fouquet. Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. *Software & Systems Modeling*, 14(2):905–920, 2013. 2.2.4
- [Jéz07] Jean-Marc Jézéquel. Generative Software Engineering. In Labit, Claude, editor, *Shaping the Future: 10 years of IrisaTech*, pages 51–54. IrisaTech, 2007. 2.1
- [JMD⁺14] Jean-Marc Jézequéel, David Mendez, Thomas Degueule, Benoit Combemale, and Olivier Barais. When Systems Engineering Meets Software Language Engineering. In *Complex Systems Design & Management (CSD&M 2014)*, Paris, France, November 2014. Springer. 1
- [Kü05] Thomas Kühne. What is a Model? In *Language Engineering for Model-Driven Software Development, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl*, pages 200–0, 2005. 2.1
- [KGO⁺01] Rohit Khare, Michael Guntersdorfer, Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. xADL: Enabling Architecture-Centric Tool Integration with XML. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*. IEEE, 2001. 4.3, 6.1, 8.4
- [KKL⁺15] Dierk König, Paul King, Guillaume Laforge, Hamlet D’Arcy, Cédric Champeau, Erik Pragt, and Jon Skeet. *Groovy in Action*. Manning Publications, 2015. 4.2.3
- [KKP⁺09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM’09)*, pages 7–13, 2009. 4.1
- [KLR96] Steven Kelly, Kalle Lyytinen, and Matti Rossi. Metaedit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *Advanced Information Systems Engineering*, 1996. 2.2.4
- [Knu68] Donald F. Knuth. Semantics of Context-Free Languages. *Mathematical systems theory*, 12:127–145, 1968. 2.2.4

- [KR05] Vinay Kulkarni and Sreedhar Reddy. Model-Driven Development of Enterprise Applications. In *UML Modeling Languages and Applications*, pages 118–128. Springer, 2005. 2.1
- [Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, 2010. 1, 2, 2.1, 2.1, 2.2, 2.2, 2.2, 2.2.2, 2.2.4, A.1.3, A.2.1
- [KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006*, pages 150–158, Finland, 2006. University of Jyväskylä. 3.3
- [KRV07] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Proceedings of Models 2007*, pages 286–300, 2007. 2.2
- [KRV08a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Mit Sprachbaukästen zur schnelleren Softwareentwicklung: Domänenspezifische Sprachen modular entwickeln. *Objektspektrum*, 4:42–47, 2008. 2.2
- [KRV08b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proceedings of Tools Europe*, 2008. 1, 2, 2.1, 2.2, 4.3, A.2.1, A.2.2
- [KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. In *International Journal on Software Tools for Technology Transfer (STTT)*, volume 12, pages 353 – 372, 2010. 1.3, 2.2, 2.2.2, 4.3, A.2.1, A.2.2
- [KSB10] Markus Klotzbücher, Peter Soetens, and Herman Bruyninckx. Orocos RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages. In *International Workshop on Dynamic languages for RObotic and Sensors*, 2010. 4.4
- [KSBD11] Markus Klotzbücher, Ruben Smits, Herman Bruyninckx, and Joris De Schutter. Reusable Hybrid Force-Velocity controlled Motion Specifications with executable Domain Specific Languages. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4684–4689. IEEE, September 2011. 1.1
- [Küh06] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006. 2.1
- [KV10] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference*

BIBLIOGRAPHY

- on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010.*, pages 444–463, 2010. 2.2.4
- [KvdSV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*, 2009. 2.2.4
- [Lee10] Edward A Lee. CPS Foundations. In *Proceedings of the 47th Design Automation Conference*, pages 737–742. ACM, 2010. 4.2.4, 5.5.1
- [LeJ] LeJOS - Java for Lego Mindstorms. <http://www.lejos.org>. Accessed: 2015-10-07. 3.1, 3.1, 6.2.2, 9
- [Lis87] Barbara Liskov. Keynote Address - Data Abstraction and Hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum), OOPSLA '87*, pages 17–34, New York, NY, USA, 1987. ACM. 4.1.3
- [LNPR⁺13] Markus Look, Antonio Navarro Perez, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In *Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC)*, Miami, Florida, USA, 2013. 1.4, 2.2, 2.2, 2.2.1, 2.2.2, 4.3, 5.2
- [LPJ10] Marc M. Lankhorst, Henderik A. Proper, and Henk Jonkers. The Anatomy of the ArchiMate Language. *International Journal of Information System Modeling and Design (IJISMD)*, 1(1):1–32, 2010. 8.4
- [LSGB12] Tinne De Laet, Wouter Schaekers, Jonas De Greef, and Herman Bruyninx. Domain Specific Language for Geometric Relations between Rigid Bodies targeted to robotic applications. In *Proceedings of the Third International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2012)*, 2012. 1.1, 4.4
- [LTR05] Perttu Laurinen, Lauri Tuovinen, and Juha Röning. Smart Archive: a Component-based Data Mining Application Framework. In *5th International Conference on Intelligent Systems Design and Applications (ISDA '05)*, pages 20–25. IEEE, 2005. 2.3
- [LW11] Ingo Lütkebohle and Sven Wachsmuth. Requirements and a Case-Study for SLE from Robotics: Event-oriented Incremental Component Construction. *Workshop on Software-Language-Engineering for Cyber-Physical Systems*, 2011. 4.4
- [Má] Más website <http://www.mas-wb.com>. [Online; accessed 2014-10-12]. 2.2.4

- [MAHR10] Henrik Mühe, Andreas Angerer, Alwin Hoffmann, and Wolfgang Reif. On reverse-engineering the KUKA Robot Language. In *First International Workshop on Domain-Specific Languages and Models for ROBotic Systems*, 2010. 1.1, 4.4
- [MAKT11] Fumio Machida, Ermeson Andrade, Dong Seong Kim, and Kishor S Trivedi. Candy: Component-based Availability Modeling Framework for Cloud Service Management Using SysML. In *30th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 209–218. IEEE, 2011. 2.3
- [MC12] Alvaro Miyazawa and Ana Cavalcanti. Refinement-oriented models of Stateflow charts. *Science of Computer Programming*, 77(10):1151–1177, 2012. 4.4
- [McI68] Douglas McIlroy. Mass-Produced Software Components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968. 2.3
- [MCWF02] Hong Mei, Feng Chen, Qianxiang Wang, and Yaodong Feng. ABC/ADL: An ADL Supporting Component Composition. In *Formal Methods and Software Engineering*, pages 38–47. Springer, 2002. 1.1, 4.3
- [MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In *Software Engineering — ESEC’95*, pages 137–153. Springer, 1995. 2.3, 4.3, 8.4
- [MDT07] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49(1):12–31, 2007. 2.3, 3, 4.3, 4.4, 10.3
- [Mer13] Marjan Mernik. An Object-oriented Approach to Language Compositions for Software Language Engineering. *Journal of Systems and Software*, 2013. 2.2.4
- [MFBC12] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoit Combemale. Modeling Modeling Modeling. *Software & Systems Modeling*, 11(3):347–359, 2012. 2.1
- [MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, December 2005. 1.1, 6
- [MLM⁺13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, 2013. 2.3, 4.4, 6

BIBLIOGRAPHY

- [Mon] MontiArc. <http://www.monticore.de/languages/montiarcl/>, [Online; accessed 2015-12-17]. 2.4
- [Mos09] Pieter J. Mosterman. Elements of a Robotics Research Roadmap: A Model-Based Design Perspective, 2009. Accessed: 2014-11-21. 1.1, 2.3
- [MP93] Zohar Manna and Amir Pnueli. Verifying Hybrid Systems. In *Hybrid Systems*, pages 4–35. Springer, 1993. 4.2.4
- [MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM, 2013. 2.4
- [MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *36th International Conference on Software Engineering (ICSE 2014)*, pages 95–105, Hyderabad, India, June 2014. ACM New York. 2.4
- [MRT99] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 44–53. IEEE, 1999. 1.1, 4.3
- [MRT03] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit. In *2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2436–2441. IEEE, 2003. 4.4
- [MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000. 1.1, 2.3, 2.4, 4.3, 4.4
- [Mue13] Ken Mueller. *Literature Research and Analysis on Robotic Architectures and Frameworks*. Bachelor’s thesis, RWTH Aachen, 2013. 1.1, 2.4, TRQ-1
- [Mur02] Jan Murray. Specifying Agents with UML in Robotic Soccer. In *The First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, AAMAS ’02, pages 51–52, New York, NY, USA, 2002. ACM. 4.4
- [NDZR04] Leila Naslavsky, Hadar Ziv Dias, H Ziv, and D Richardson. Extending xADL with Statechart Behavioral Specification. In *Third Workshop on Architecting Dependable Systems (WADS)*, Edinburgh, Scotland, pages 22–26. IET, 2004. 1.1, 2.3, 4.4, 8.4

- [NFBL10] Tim Niemueller, Alexander Ferrein, Daniel Beck, and Gerhard Lakemeyer. *Design Principles of the Component-Based Robot Software Framework Fawkes*, volume 6472 of *Lecture Notes in Computer Science*, pages 300–311. Springer, Darmstadt, Germany, 2010. 1.1, 2.3, 4.4
- [NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In I. Ober, A. S. Gokhale, J. H. Hill, J.-M. Bruel, M. Felderer, D. Lugato, and A. Dabholka, editors, *Proceedings of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing*, volume 1118 of *CEUR*, pages 15–24, Miami, Florida, USA, 2013. CEUR-WS.org. 2.2.2, 2.3, 4.3
- [NTN⁺04] Dag Nyström, Aleksandra Tešanovic, Mikael Nolin, Christer Norström, and Jörgen Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. IET, 2004. 2.3
- [NW12] Arne Nordmann and Sebastian Wrede. A Domain-Specific Language for Rich Motor Skill Architectures. In *Proceedings of the Third International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2012)*, 2012. 1.1, 4.4
- [OAR⁺14] Francisco J. Ortiz, Diego Alonso, Francisca Rosique, Francisco Sánchez-Ledesma, and Juan A. Pastor. A Component-Based Meta-Model and Framework in the Model Driven Toolchain C-Forge. In Davide Brugali, Jan F. Broenink, Torsten Kroeger, and Bruce A. MacDonald, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 8810 of *Lecture Notes in Computer Science*, pages 340–351. Springer International Publishing, 2014. 1.1
- [Oli07] Travis E. Oliphant. Python for Scientific Computing. *Computing in Science Engineering*, 9(3):10–20, May 2007. 4.1.1
- [OMG03] Object Management Group. MDA Guide Version 1.0.1, June 2003. http://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf [Online; accessed 2015-12-17]. 2.1
- [OMG06] Object Management Group. MOF Specification Version 2.0 (2006-01-01), January 2006. <http://www.omg.org/docs/ptc/06-05-04.pdf>. 2.2.4
- [OMG10] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05), May 2010. <http://www.omg.org/spec/UML/2.3/Superstructure/PDF/> [Online; accessed 2015-12-17]. 1.1, 2.1, 2.2.4, 2.3, 4, 3.1, 4.4, 5.6

BIBLIOGRAPHY

- [PM03] Jorge Enrique Pérez-Martínez. Heavyweight extensions to the UML meta-model to describe the C3 architectural style. *ACM SIGSOFT Software Engineering Notes*, 28(3), 2003. 8.4
- [PM06] Roland Petrasch and Oliver Meimberg. *Model Driven Architecture: Eine praxisorientierte Einführung in die MDA*. dpunkt Verlag, 2006. 2.1
- [QGC⁺09] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009. 1.1, 2.3, 3.1, 3.1, 4.4, 6.6, 6.2, 6.2.3, 7.4.2, 7.4.2, 7.5, 9, 9.1.2
- [Rai05] Chris Raistrick. Applying MDA and UML in the Development of a Healthcare System. In *UML Modeling Languages and Applications*, pages 203–218. Springer, 2005. 2.1
- [RBH⁺07] Ralf Reussner, Steffen Becker, Jens Happe, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model, 2007. 4.3
- [Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014. 1.3, 2.4, 3.3, 10.2
- [RMT14a] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. Model-driven Software Development Approaches in Robotics Research. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, 2014. 4.4
- [RMT14b] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. SafeRobots: A Model-Driven Framework for Developing Robotic Systems. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1517–1524. IEEE, 2014. 4.4
- [RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 5(1-2):29–53, July 2011. 2.4, 4.4, 5.1.3, 5.6
- [RRRW14] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. In *1st International Workshop on Model-Driven Robot Software Engineering (MORSE 2014)*, volume 1319 of *CEUR Workshop Proceedings*, pages 66 – 77, York, Great Britain, July 2014. 1.4, 7, 7.1, 7.5, 8.1.1, 9.2.1

- [RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015. 1.4
- [RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Norbert Seyff and Anne Koziolek, editors, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 143–155. Monsenstein und Vannerdat, Münster, 2012. 1.4
- [RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Case Study on Model-Based Development of Robotic Systems using MontiArc with Embedded Automata. In Holger Giese, Michaela Huhn, Jan Philipps, and Bernhard Schätz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme*, pages 30–43, 2013. 1.4, 9.1.1, 9.1.1, 9.1.1
- [RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In Stefan Wagner and Horst Licher, editor, *Software Engineering 2013 Workshopband*, volume 215 of *LNI*, pages 155–170. GI, Kölle Druck+Verlag GmbH, Bonn, 2013. 1.4, 2.2, 2.2.3, *MRQ-8.5*, 4.4, 6.2.1, 7.4.1, 7.4.1, 9, 10.2
- [RRW13c] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Workshops and Tutorials Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany, May 6-10 2013. 1.4, 4.1.1
- [RRW14a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Number 20 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2014. 1.4, 4.1.4, 5, 5.1.3, 5.3, 5.5.1
- [RRW14b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Multi-Platform Generative Development of Component & Connector Systems using Model and Code Libraries. In *1st International Workshop on Model-Driven Engineering for Component-Based Systems (ModComp 2014)*, volume 1281 of *CEUR Workshop Proceedings*, pages 26 – 35, Valencia, Spain, September 2014. 1.4, 6.4
- [RRW15a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Composing Code Generators for C&C ADLs with Application-Specific Behavior

BIBLIOGRAPHY

- Languages (Tool Demonstration). In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 113–116, New York, NY, USA, 2015. ACM. 1.4, 9.2.1
- [RRW15b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *Proceedings of the Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, 2015. 1.4, 4.1.1
- [RRW15c] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Transforming Platform-Independent to Platform-Specific Component and Connector Software Architecture Models. In *2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems (Mod-Comp) 2015*, volume 1463 of *CEUR Workshop Proceedings*, pages 30 – 35, Ottawa, Canada, September 2015. 1.4
- [RSVW10] Bernhard Rumpe, Martin Schindler, Steven Völkel, and Ingo Weisemöller. Generative Software Development. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010)*, volume 2, pages 473–474. ACM, 2010. 2.1
- [RSVW11] Bernhard Rumpe, Martin Schindler, Steven Völkel, and Ingo Weisemöller. Agile Development with Domain Specific Languages. *Modelling Foundations and Applications*, pages 387–388, 2011. 2.2
- [Rub12] Kenneth S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Professional, 2012. 9.1.1, 9.1.2, 9.1.3
- [Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Doktorarbeit, Technische Universität München, 1996. 1.3, 5, 5, 5.1.4, 5.6
- [Rum11] Bernhard Rumpe. *Modellierung mit UML*. Xpert.press. Springer Berlin, 2nd edition, September 2011. 2.2.2, 2.4, 2.4, 4.2.2
- [SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008. 9
- [Sch91] Andreas Schürr. *Operationales Spezifizieren mit Programmisierten Graphersetzungssystemen: Formale Definitionen Anwendungsbeispiele and Werkzeugunterstützung*. Wiesbaden: Deutscher Universitäts-Verlag, 1991. 10.3
- [Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band

11. Shaker Verlag, 2012. 2.1, 2.2, 2.2, 2.2.1, 2.2.3, 2.2.4, 2.4, 2.4, 3.3, 4.1.3, 5, 7.4.1, 7.4.1, 7, 7.4.1, A.2.1, A.2.2, E
- [Sch14] Stefan Schubert. *Entwicklung einer I/O^w Modellierungssprache zur Einbettung in die MontiArcAutomaton-Sprachfamilie*. Master's thesis, RWTH Aachen, 2014. 5, 5.3
- [SCS07] Ulrik Pagh Schultz, David Johan Christensen, and Kasper Stoy. A Domain-Specific Language for Programming Self-Reconfigurable Robots. In *Workshop on Automatic Program Generation for Embedded Systems*, pages 28–36, 2007. 1.1, 4.4
- [Sei03] Ed Seidewitz. What Models Mean. *Software, IEEE*, 20(5):26–32, Sept 2003. 2.1
- [Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *Software, IEEE*, 20(5):19–25, Sept 2003. 1.1, 2.1
- [Sel06] Bran Selic. Model-Driven Development: Its Essence and Opportunities. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, April 2006. 2.1
- [SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996. 2.3
- [SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press Series. ACM Press, 2002. 2.3
- [SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, pages 136–145, Washington, DC, USA, 2006. IEEE Computer Society. 8.4
- [Sie00] Jon Siegel. *CORBA 3 Fundamentals and Programming*, volume 2. John Wiley & Sons Chichester, 2000. 4.4
- [Sim] MathWorks Simulink. <http://www.mathworks.com/products/simulink/>, [Online; accessed 2015-12-17]. 2.1
- [SMN⁺07] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. Model-Driven Development with the jABC. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *Hardware and Software, Verification and Testing*, volume 4383 of *Lecture Notes in Computer Science*, pages 92–108. Springer Berlin Heidelberg, 2007. 7.5

BIBLIOGRAPHY

- [SMTS09] Jonathan Sprinkle, Marjan Mernik, J Tolvanen, and Diomidis Spinellis. Guest editors' introduction: What Kinds of Nails Need a Domain-Specific Hammer? *Software, IEEE*, 26(4):15–18, 2009. 1.1
- [Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *ACM Sigplan Notices*, 21(11):38–45, 1986. 2.3
- [SOK05] Adel Smeda, Mourad Oussalah, and Tahar Khammaci. MADL: Meta Architecture Description Language. In *Third ACIS International Conference on Software Engineering Research, Management and Applications, 2005*, pages 152–159. IEEE, 2005. 4.3
- [Sol05] Riccardo Solmi. *Whole platform*. PhD thesis, University of Bologna, 2005. 2.2.4
- [SRVK10] Jonathan Sprinkle, Bernhard Rümpe, Hans Vangheluwe, and Gabor Karai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 57–76. Springer, 2010. 2.1
- [SSL11] Christian Schlegel, Andreas Steck, and Alex Lotz. Model-Driven Software Development in Robotics : Communication Patterns as Key for a Robotics Component Model. In Daisuke Chugo and Sho Yokota, editors, *Introduction to Modern Robotics*. iConcept Press, 2011. 1.1, 2.3, 4.3, 4.4, 6.1, 6.4, 8.4, 9, 9.1.3, 9.2.3
- [Sta] Mathworks Stateflow. <http://www.mathworks.de/products/stateflow/>, [Online; accessed 2015-12-17]. 4.4
- [Sta73] Herbert Stachowiak. Allgemeine Modelltheorie, 1973. 2.1
- [Sta06] Miroslaw Staron. Adopting Model Driven Software Development in Industry - A Case Study at Two Companies. In *Model Driven Engineering Languages and Systems*, pages 57–72. Springer, 2006. 2.1
- [SVEH05] Thomas Stahl, Markus Völter, Sven Efttinge, and Arno Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt Verlag, 2005. 1.1
- [TBD07] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, pages 44–53, Washington, DC, USA, 2007. IEEE Computer Society. 7.5
- [THR⁺13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rümpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *2013 ICRA IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany, 2013. 1.1, 4.4

- [TM02] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*, volume 2. Springer, 2002. 2.1
- [TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley and Sons, Inc., 1 edition, 2009. 2.3, 4.4
- [TRMS09] Lucy A. Tedd, Jelena Radjenovic, Branko Milosavljevic, and Dusan Surla. Modelling and implementation of catalogue cards using FreeMarker. *Program*, 43(1):62–76, 2009. 2.1, 2.2, 2.3
- [Tro11] Piotr Trojanek. Model-driven engineering approach to design and implementation of robot control system. In *Proceedings of the Second International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2011)*, 2011. 1.1, 4.4
- [TVT⁺13] Federico Tomassetti, Antonio Vetro, Marco Torchiano, Markus Voelter, and Bernd Kolb. A Model-Based Approach to Language Integration. In *5th International Workshop on Modeling in Software Engineering (MiSE 2013)*, 2013. 2.2
- [Tya12] Agam Kumar Tyagi. *MATLAB and SIMULINK for Engineers*. Oxford University Press, 2012. 4.4
- [UNT10] Naoyasu Ubayashi, Jun Nomura, and Tetsuo Tamai. Archface: A Contract Place Where Architectural Design and Code Meet Together. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 1, pages 75–84. ACM, 2010. 4.3
- [Vö11] Steven Völkel. *Kompositionale Entwicklung domänenpezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. 2011. Shaker Verlag, 2011. 2.1, 2.2, 2.2, 2.2, 2.2.1, 2.2.1, 2.2.2, 2.2.4, 4.2.2, 4.2.3, 4.3, C.1
- [Van13] Vanthienen, Dominick and Klotzbuecher, Markus and De Laet, Tinne and De Schutter, Joris and Bruyninckx, Herman. Rapid application development of constrained-based task modelling and execution using Domain Specific Languages. In *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1860–1866, Tokyo, Japan, 2013. 1.1, 4.4
- [VBD⁺13] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. 2.1, 2.2.4

BIBLIOGRAPHY

- [vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000. 1.1, 2.1, 2.2.4, 4.4
- [vdSCL14] Tijs van der Storm, William R. Cook, and Alex Loh. The design and implementation of Object Grammars. *Science of Computer Programming*, 2014. 2.2.4
- [VKB14] Dominick Vanthienen, Markus Klotzbuecher, and Herman Bruyninckx. The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming. *JOSER: Journal of Software Engineering for Robotics*, 5(1):17–35, 2014. 4.4
- [VNE⁺01] Richard Volpe, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das. The CLARAty Architecture for Robotic Autonomy. In *2001 IEEE Aerospace Conference Proceedings*, volume 1, 2001. 4.4
- [VS10] Markus Voelter and Konstantin Solomatov. Language and IDE Modularization, Extension and Composition with MPS. *Software Language Engineering (SLE’10)*, page 16, 2010. 2.2.4
- [VSB⁺13] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. Wiley, 2013. 2.1
- [VVKM00] Rob Van Ommering, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000. 1.1, 2.3, 4.3, 8.4
- [WBGK08] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science*, 2008. 2.2.4
- [WdMBK02] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In *Proceedings of 11th International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science LNCS*, 2002. 2.2.4
- [Wei06] Tim Weilkiens. *Systems Engineering mit SysML/UML*. UML. dpunkt. verlag, 2006. 4.4
- [WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The State of Practice in Model-Driven Engineering. *Software, IEEE*, 31(3):79–85, 2014. 1.1, 2.1, 2.3, 10.3

- [WICE03] Brian C. Williams, Michel D. Ingham, Seung H. Chung, and Paul H. Elliott. Model-Based Programming of Intelligent Embedded Systems and Robotic Space Explorers. In *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, pages 212–237, 2003. 4.4
- [Wil01] David S. Wile. Supporting the DSL Spectrum. *Computing and Information Technology*, 4:263–287, 2001. 2.1, 2.2.4, 2.3, 4.3, 6, 6.4, 8.4, 9.1.3, 10.3
- [Wir96] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996. 2.2
- [WWM⁺07] Thomas Weigert, Frank Weil, Kevin Marth, Paul Baker, Clive Jervis, Paul Dietz, Yexuan Gui, Aswin Van Den Berg, Kim Fleer, David Nelson, et al. Experiences in Deploying Model-Driven Engineering. In *SDL 2007: Design for Dependable Systems*, pages 35–53. Springer, 2007. 2.1
- [YHMP09] Zhibin Yang, Kai Hu, Dianfu Ma, and Lei Pi. Towards a Formal Semantics for the AADL Behavior Annex. In *Design, Automation Test in Europe Conference Exhibition*, pages 1166–1171, April 2009. 4.4
- [ZR11] Steffen Zschaler and Awais Rashid. Towards Modular Code Generators Using Symmetric Language-Aware Aspects. In *Proceedings of the 1st International Workshop on Free Composition*, FREECO ’11, pages 6:1–6:5, New York, NY, USA, 2011. ACM. 7.5

Appendix A

Modeling Language Grammars

This appendix presents the grammars of the modeling languages presented throughout this thesis. First, Section A.1 describes two variants of the MontiArcAutomaton ADL grammar. Afterwards, Section A.2 presents two variants of the AUTOMATA grammar before Section A.3 describes the generator description grammar.

A.1 MontiArcAutomaton ADL Grammars

The MontiArcAutomaton ADL extends from MontiArc. Hence, this appendix first presents the MontiArc grammar before it introduces two variants of the MontiArcAutomaton ADL grammar. The MontiArc grammar and one MontiArcAutomaton ADL grammar are simplified for human comprehension, the other MontiArcAutomaton ADL is the grammar actually used with MontiCore. The former omit technical details to improve clarity.

A.1.1 MontiArc Grammar for Human Comprehension

The MontiArcAutomaton ADL extends MontiArc and inherits productions to describe components, connectors, and ports. We therefore present a simplified variant of the MontiArc grammar for better comprehension. The MontiArc grammar extends from another grammar (`CommonValues`) itself (l. 1) and inherits commonly used productions (such as `Expression`) from it (l. 25). In MontiArc, most productions implement the interface `ArcElement` (l. 3), which allows reusing these in component bodies. This for instance holds for components as well (l. 5) and allows to easily define inner components [HRR12]. A component (ll. 5-8) has a name, an optional instance name (for inner components), a head, and a body. The head is defined by `ArcComponentHead` and consists of an optional list of configuration parameters followed by up to one extension declaration. A component's body (l. 15) consists of a set of `ArcElement` instances. This can be either an `ArcInterface` (ll. 17-18), consisting of a set of ports (l. 20), a subcomponent (ll. 22-27) with a list of instance names (l. 26), or a connector (ll. 28-30). These and more productions, as well as the corresponding well-formedness rules are presented in [HRR12].

```

1 grammar MontiArc extends CommonValues {
2
3   interface ArcElement;
4
5   ArcComponent implements ArcElement =
6     "component" Name (instanceName:Name)?
7     head:ArcComponentHead
8     body:ArcComponentBody;
9
10  ArcComponentHead = ("[" ArcParameter (",," ArcParameter)* "]")?
11    ("extends" superComponent:ReferenceType)?;
12
13  ArcParameter = Type Name;
14
15  ArcComponentBody = "{" ArcElement* "}";
16
17  ArcInterface implements ArcElement =
18    "port" ports:ArcPort (",," ports:ArcPort)* ";" ;
19
20  ArcPort = (in:["in"] | out:["out"]) Type Name?;
21
22  ArcSubComponent implements ArcElement =
23    "component"
24    type:ReferenceType
25    ("(" Expression (",," Expression)* ")" )?
26    (instances:Name (",," instances:Name)* )? ";" ;
27
28  ArcConnector implements ArcElement=
29    "connect" source:QualifiedName "->"
30    targets:QualifiedName (",," targets:QualifiedName)* ";" ;
31 }

```

MCG

Listing A.1: The quintessential elements of the MontiArc grammar [HRR12] for human comprehension.

A.1.2 MontiArcAutomaton ADL Grammar for Human Comprehension

The MontiArcAutomaton ADL grammar for human comprehension concisely describes the new modeling language elements of MontiArcAutomaton. After declaring a package (l. 1), the grammar declaration begins with the keyword `grammar`, followed by the grammar's name, and a declaration that it extends the MontiArc grammar (l. 3). Thus, MontiArcAutomaton ADL inherits all productions from MontiArc (the MontiArc grammar is presented in [HRR12]). Afterwards, MontiArcAutomaton declares an external production for embedding of component behavior modeling languages (l. 5). This production serves as MontiArcAutomaton ADL extension point and is used to integrate component behavior languages grammatically.

The central production of the MontiArcAutomaton ADL grammar is `MAAComponent` (ll. 7-10), which describes a component type definition in MontiArcAutomaton. The `MAAComponent` consists of an optional stereotype, followed by the optional keyword `interface` (cf. Section 4.1.1), the keyword `component`, its name, a component head, and a body. Both, component head and body, are inherited from MontiArc. The component head contains a list of `ArcParameter` instances and the body a list of instances of the `ArcElement` interface. The following productions make use of these.

The production `DefaultParameter` (ll. 12-13) introduces parameters with default values (cf. Section 4.1.1) to MontiArcAutomaton. To this effect, it inherits from the production `ArcParameter` used by MontiArc and requires the assignment of a value. The value is created as an instance of the production `CVEexpression`, which is MontiArc's expression language. Behavior models are integrated via the `ArcComponentBehavior` production (ll. 15-18), which implements the interface `ArcElement` and thus can be used throughout component bodies (cf. Section 4.1.1). It begins with an optional stereotype, followed by the keyword `behavior`, an identifier (`kind`), and an optional name. Afterwards, in curly brackets, it contains a single instance of the external production `BehaviorModel`, which is instantiated with the identifier. Embedding replaces `BehaviorModel` with the embedded production of a behavior language.

Productions for references to run-time environments (cf. Section 4.1.1) and GPL behavior implementations (cf. Section 4.1.1) follow (ll. 20-24). Both implement `ArcElement` as well, begin with a unique keyword, and require a single name.

Component variables (cf. Section 4.1.1) follow (ll. 26-29). MontiArcAutomaton ADL components can contain multiple variable declarations (ll 26-27), each of which begin with the optional keyword `var`, followed by their type (also inherited from MontiArc), and a list of variables. This resembles variable declarations in Java and allows declaring multiple variables of the same type conveniently. The individual variables (l. 29) also begin with an optional stereotype, followed by a name and an optional `CVEexpression` initial assignment.

A.1.3 MontiArcAutomaton ADL Grammar for MontiCore

This complete MontiArcAutomaton ADL grammar also begins with a package declaration, the grammar's name, and extension of MontiArc (ll. 1-3) as introduced above. Afterwards, a MontiCore options follow (ll. 4-8), which declare the main production (`compilationunit`) of this grammar and arguments for the parser and lexer to be employed. Subsequently, the grammar defines an external production for embedding of component behavior productions from other languages (l. 10) and an interface for keywords (l. 12). Currently, this interface is used with the single keywords `interface` only (ll. 14), but defining keywords via the interface allows to easily add new keywords. All subsequent productions begin with a slash to indicate that they are derived [Kra10]. This entails generation of so-called prototype classes for AST nodes, which expect to be inherited by handcrafted AST node classes, and enables adding functionality to the language's AST classes.

```
1 package de.montiarcautomaton.core.languages.adl;
2
3 grammar MontiArcAutomaton extends mc.uml.p.arc.MontiArc {
4
5   external BehaviorModel;
6
7   MAAComponent = Stereotype? ("interface")?
8     "component" Name
9     head:ArcComponentHead
10    body:ArcComponentBody;
11
12  DefaultParameter extends ArcParameter =
13    Type Name "=" value:CVExpression?;
14
15  ArcComponentBehavior implements ArcElement =
16    Stereotype? "behavior" kind:Name Name? "{"
17      BehaviorModel(parameter kind)
18    }";
19
20  GPLBehaviorImplementation implements ArcElement =
21    Stereotype? "implementation" name:QualifiedName ";";
22
23  RTE implements ArcElement =
24    Stereotype? "rte" name:QualifiedName ";";
25
26  VariableDeclaration implements ArcElement =
27    "var"? Type Variable ("," Variable)* ";";
28
29  Variable = Stereotype? Name ("=" CVExpression)?;
30 }
```

MCG

Listing A.2: The MontiArcAutomaton ADL grammar for human comprehension.

In this grammar, the production MAAComponent extends ArcComponent and thus inherits its properties. It also features an optional instance name, as MontiArc uses the same production for inner components (see context condition MU6). The following productions employ so-called syntactic predicates to guide parsing. For instance, the production DefaultParameter (ll. 20-21) utilizes the syntactic predicate (Type Name "=") => ArcParameter that helps the parser to distinguish instances of DefaultParameter (which must be followed by "=") from instances of ArcParameter. Aside from begin derived and employing syntactic predicates, the remaining productions to not differ from the MontiArcAutomaton ADL grammar for human comprehension.

```

1 package de.montiarcautomaton.core.languages.adl;
2
3 grammar MontiArcAutomaton extends mc.umlpa.arc.MontiArc {
4   options {
5     compilationunit MAAComponent
6     parser lookahead=5
7     lexer lookahead=7
8   }
9
10  external BehaviorModel;
11
12  interface Keyword;
13
14  InterfaceKeyword implements Keyword = "interface";
15
16  / MAAComponent extends ArcComponent =
17    Stereotype? Keyword? "component" Name (instanceName:Name)?
18    head:ArcComponentHead body:ArcComponentBody;
19
20  / DefaultParameter extends (Type Name "") => ArcParameter =
21    Type Name "=" value:CVExpression?;
22
23  / ArcComponentBehavior implements
24    (Stereotype? "behavior" kind:Name Name? "{}")
25    => ArcElement =
26    Stereotype? "behavior" kind:Name Name? "{}"
27    BehaviorModel(parameter kind)
28    "{}";
29
30  / GPLBehaviorImplementation implements
31    (Stereotype? "implementation" name:QualifiedName ";")
32    => ArcElement =
33    Stereotype? "implementation" name:QualifiedName ";";
34
35  / RTE implements
36    (Stereotype? "rte" name:QualifiedName ";")
37    => ArcElement =
38    Stereotype? "rte" name:QualifiedName ";";
39
40  / VariableDeclaration implements ArcElement =
41    "var"? Type Variable ("," Variable)* ";";
42
43  / Variable = Stereotype? Name ("=" CVExpression)?;
44 }

```

MCG

Listing A.3: The MontiArcAutomaton ADL grammar for processing by MontiCore.

A.2 Automata Grammars

The AUTOMATA grammar comprises productions to describe finite automata that operate in a context of inputs, outputs, and variables. We provide two versions of the AUTOMATA grammar: one for human comprehension that omits parser directives and elements to influence AST generation and one version as processed by MontiCore. The next sections introduce both grammars and the language configuration file that defines embedding of Java/P expressions.

A.2.1 Automata Grammar for Human Comprehension

The following grammar is prepared for human comprehension. Therefore, it omits parser directives, production derivation information, and syntactic predicates. MontiCore provides such elements to improve parser performance and to influence AST generation [KRV08b, Kra10, KRV10].

The AUTOMATA grammar depicted in Listing A.4 begins with the keyword `grammar` followed by its name. Afterwards it declares to extend the grammar `CommonValues` (presented in [Sch12]), which provides productions common to many grammars, such as `Name` and `Type`. The productions `GuardExpression` (l. 3) and `Valuation` (l. 4) are external non-terminals and act as extension points for productions from other languages (for instance Java/P expressions). Separating both forms of expressions via different non-terminals allows easy integration of other expressions languages into guards, while retaining valuation expressions. A previous version of MontiArcAutomaton, for instance, embedded OCL/P [Sch12] expressions into guards.

The main production of the AUTOMATA grammar is `Automaton` (ll. 6-9) which holds the modeled automaton's stereotype, name, context, and content. An automaton's context (production `AutomatonContext`) consists of an arbitrary number of inputs, outputs, and variables (l. 11) in arbitrary order. Each of these (ll. 14-16) begins with a certain keyword and a type followed by a list of `Channel` non-terminals. Each channel (l. 17) is a name with an optional assignment of external non-terminal `Valuation`.

An automaton's content (production `AutomatonContent`, l. 12) comprises an arbitrary number of state declarations, initial state declarations, and transitions in arbitrary order. Each state declaration (`States`, l. 19) begins with the keyword `state` followed by a list of states. Each `State` is a name with an optional stereotype (l. 20). Initial state declarations (l. 22) begin with the keyword `initial`, followed by a list of initial state names and an optional `Block` that describes the states' initial output. Transitions omit introductory keywords, but are recognized by their unique syntax instead. Each transition (ll. 24-27) begins with the name of its source state, followed by an optional target state name, an optional guard, an optional stimulus `Block`, and an optional action `Block`.

Guards (l. 29) consist of square brackets that delimit their expressions of external non-terminal `GuardExpression`. The `Block` productions (l. 30) used for stimuli and actions comprise assignment lists of non-terminal `Assignment`, optionally enclosed in curly brackets. Assignment lists (l. 32) are comma-separated lists of assignments, which

consist of at least one assignment. Consequently, empty curly brackets {} are prohibited. Each assignment (l. 33) may begin with an optional name to identify the referenced input, output, or variable, followed by either an alternative or a list of values. Alternatives (l. 35) begin with the keyword `alt` followed by alternative value lists, enclosed in curly brackets. Thus, nesting alternatives is not possible. Each value list (ll. 37-38) consists either of a comma-separated list of external `Valuation` non-terminals delimited by square brackets, or of a single `Valuation` element.

Reflecting the distinction between `AutomataContext` and `AutomatonContent` in the grammar facilitates language embedding, as the embedding language can reuse `AutomatonContent` directly (cf. Section 4.2).

A.2.2 Automata Grammar for MontiCore

The AUTOMATA MontiCore grammar contains all elements of the simplified AUTOMATA grammar for human comprehension presented in Section A.2.1. This section therefore only discusses their differences, which begin by stating a package declaration (l. 1) and extending `CommonValues` [Sch12] via its full name (l. 4). Afterwards it specifies options, such as the name of the compilation unit non-terminal and the parser's and lexer's lookahead (ll. 7-9). After declaring the familiar external non-terminals, each subsequent production begins with a dash / to indicate that the AST generator should prepare the AST classes for manual extension. The resulting AST infrastructure thus expects handcrafted subclasses of the AST classes following a naming convention. AUTOMATA employs this mechanism to enrich the generated AST classes with additional fields and methods for easier access and to encapsulate related calculations [KRV08b, KRV10]. Another difference arises from the option `greedy` added to `ValueList` (ll. 42-43), which instructs MontiCore to process comma-separated `Valuation` lists modestly.

A.3 Generator Description Grammar

Models of the generator description modeling language (Section 7.2) represent code generators in terms of their type (as represented by the interface), and their most important properties for composition, configuration, and execution. To this effect, the generator description grammar must provide means to identify a code generator description, reference the interface the represented generator should implement, declare the processable language fragment, context conditions, additionally supported behavior languages, a start template, and a run-time environment.

The code generator description grammar begins with a package declaration followed by a grammar declaration (ll. 1-4). The latter also declares, that this grammar inherits from the `CommonValues` grammar. After the AST options (ll. 6-8) declaring the compilation unit (i.e., the 'top-level' model element containing all other model elements of this language), the grammar defines the production `GeneratorDescription` (ll. 10-13) which consists of the keyword `generator` followed by a name, the keyword `conforms` and a the qualified name of the interface the represented generator should implement.

```

1 grammar Automata extends CommonValues {
2
3   external GuardExpression;
4   external Valuation;
5
6   Automaton = Stereotype? "automaton" Name "{" "
7     AutomatonContext
8     AutomatonContent
9   "}";
10
11  AutomatonContext = ( Input | Output | Variable )*
12  AutomatonContent = ( States | Initial | Transition)*;
13
14  Input = "input" Type Channel ("," Channel)* ";" ;
15  Output = "output" Type Channel ("," Channel)* ";" ;
16  Variable = "variable" Type Field ("," Channel)* ";" ;
17  Channel = Name ("=" ValueExpression)?;
18
19  States = "states" State ( "," State )* ";" ;
20  State = Stereotype? Name;
21
22  Initial = "initial" Name ("," Name)* ("/" Block)? ";" ;
23
24  Transition = source:Name ("->" target:Name)?
25    Guard?
26    stimulus:Block?
27    ("/" action:Block)? ";" ;
28
29  Guard = "[" GuardExpression "]";
30  Block = ( "{" AssignmentList "}" ) | AssignmentList;
31
32  AssignmentList = Assignment ("," Assignment)*;
33  Assignment = (Name "=")? (Alternative | ValueList);
34
35  Alternative = "alt{" ValueList ("," ValueList)* "}";
36
37  ValueList = ( ("[" Valuation ("," Valuation )* "]")
38            | Valuation );
39 }

```

MCG

Listing A.4: A simplified AUTOMATA grammar for human comprehension.

Afterwards, a body consisting of a list of `DescriptionElement` productions, delimited by curly brackets, follows. The interface production `DescriptionElement` (l. 15) allows adding different productions to the body, as long as these implement this interface. Currently, these productions are `Start`, `Language`, `RTE`, `ContextConditions`, and `Behaviors` (ll. 17-34), which allow to define the aforementioned properties.

```

1 package de.montiarcautomaton.core.languages.ioautomaton; MCG
2
3 grammar Automata
4   extends de.monticore.lang.common.CommonValues {
5
6   options {
7     compilationunit Automaton
8     parser lookahead = 3
9     lexer lookahead = 5
10    }
11
12   external GuardExpression;
13   external Valuation;
14
15   / Automaton = Stereotype? "automaton" Name "{}"
16     AutomatonContext AutomatonContent "}";
17
18   / AutomatonContext = (Input | Output | Variable)*
19   / AutomatonContent = (States | Initial | Transition)*;
20
21   / Input = "input" Type Field ("," Field)* ";" ;
22   / Output = "output" Type Field ("," Field)* ";" ;
23   / Variable = "variable" Type Field ("," Field)* ";" ;
24   / Field = Name ("=" ValueExpression)?;
25
26   / States = "states" State ("," State)* ";" ;
27   / State = Stereotype? Name;
28
29   / Initial = "initial" Name ("," Name)* ("/" Block)? ";" ;
30
31   / Transition = source:Name ("->" target:Name)?
32     Guard? stimulus:Block? ("/" action:Block)? ";" ;
33
34   / Guard = "[" GuardExpression "]";
35   / Block = ("{" AssignmentList "}") | AssignmentList;
36
37   / AssignmentList = Assignment ("," Assignment)*;
38   / Assignment = (Name "=")? (Alternative | ValueList);
39
40   / Alternative = "alt{" ValueList ("," ValueList)* "}";
41
42   / ValueList = (Valuation | options {greedy=false;} :(
43     ("[" Valuation ("," Valuation)* "]")) );
44 }

```

Listing A.5: The complete AUTOMATA MontiCore grammar.

The productions Start (ll. 17-18), Language (ll. 20-21), and RTE (ll. 23-25) each consist of distinct keyword followed by a qualified name. Slightly more complex are the productions ContextConditions (ll. 26-30) and Behaviors (ll. 32-34). Both begin with a distinct keyword and span a body containing lists of names delimited by curly brackets. For ContextConditions, the keyword contextconditions may be followed by the keyword in and a qualified name to describe a common base package the context conditions listed afterwards reside in (cf. Section 7.2.1). As this does not prohibit defining, for instance, multiple start templates, the language's context conditions (Section 7.2.3) take care of this.

```
1 package montiarcautomaton.languages.generatordescription;
2
3 grammar GeneratorDescription
4   extends de.monticore.lang.common.CommonValues {
5
6   options {
7     compilationunit GeneratorDescription
8   }
9
10 / GeneratorDescription =
11   "generator" Name "conforms" interface:QualifiedName "{ "
12     DescriptionElement*
13   "}";
14
15 interface DescriptionElement;
16
17 Start implements DescriptionElement =
18   "start" QualifiedName "(" ")"? ";" ;
19
20 Language implements DescriptionElement =
21   "language" QualifiedName ";" ;
22
23 RTE implements DescriptionElement =
24   "rte" QualifiedName ";" ;
25
26 ContextConditions implements DescriptionElement =
27   "contextconditions" ("in" cocoPackage:QualifiedName)? " { "
28     coco:QualifiedName ("," coco:QualifiedName)*
29   "}";
30
31 Behaviors implements DescriptionElement =
32   "behaviors" "{ "
33     QualifiedName ("," QualifiedName)*
34   "}";
35 }
```

MCG

Listing A.6: The generator description grammar for processing by MontiCore.

A.4 Application Configuration Grammar

The application configuration grammar realizes the application configuration language elements as introduced in Section 8.1.1.

```

1 package montiarcautomaton.languages.applicationconfiguration; MCG
2
3 grammar ApplicationConfiguration
4   extends de.monticore.lang.common.CommonValues {
5
6   options { compilationunit ApplicationConfiguration }
7
8   / ApplicationConfiguration =
9     "application" Name "for" architecture:QualifiedName "{"
10    ApplicationConfigurationElement*
11  }";
12
13  interface ApplicationConfigurationElement;
14
15  / Generator implements ApplicationConfigurationElement
16    = name:QualifiedName ";";
17
18  / Binding implements ApplicationConfigurationElement =
19    "bind" subcomponent:QualifiedName "to"
20    componentType:ReferenceType
21    "(" (CVEExpression (," CVEExpression) *)? ")" ";";
22 }
```

Listing A.7: The application configuration grammar for processing by MontiCore.

After the package declaration (l. 1), the grammar begins with keyword **grammar** followed by the name **ApplicationConfiguration** and the declaration to extend the grammar **de.monticore.lang.common.CommonValues** (ll. 3-4). The production **ApplicationConfiguration** defines an application configuration and begins with the keyword **application** followed by the configuration's name, the keyword **for**, and the qualified name of the architecture it references (ll. 8-11). Afterwards, it contains a list of **ApplicationConfigurationElement** productions (l. 10), where **ApplicationConfigurationElement** is an interface (l. 13) implemented by the productions **Generator** (ll. 15-16) and **Binding** (ll. 18-21). The production **Generator** consists of a qualified name followed by a semicolon and represents the name of a participating generator. **Binding** begins with the keyword **bind**, followed by a qualified name representing the path to the bound subcomponent, a **ReferenceType** and a list of **CVEExpression** items in brackets. The production **ReferenceType** is inherited from the super grammar **CommonValues** and consists of qualified names with optional angle brackets containing further reference types. This allows modeling qualified type names with generic type arguments. **CVEExpression** is a compact expression language

similar to the expressions of Java and imported from CommonValues as well. Bindings use CVExpression instead of Java expressions to ensure compatibility with MontiArc 2.5.0, which uses the same production for component configuration arguments.

Appendix B

Survey Materials

This chapter provides the complete surveys and aggregated results of the evaluations presented in Chapter 9.

B.1 NXT Java Coffee Delivery

In the first lab course on MontiArcAutomaton, the MontiArc was introduced as the predecessor of MontiArcAutomaton and AUTOMATA models were part of the MontiArc-Automaton grammar. Hence, the survey differentiates between MontiArc and Monti-ArcAutomaton to identify C&C structure modeling (MontiArc) and behavior modeling (MontiArcAutomaton).

Questionnaire

1. What percentage (0% - 100%) of the time did you spend on...
 - understanding MontiArc: 86%
 - understanding MontiArcAutomaton: 29.00%
 - understanding LeJOS: 10.71%
 - understanding the code generation process: 33.29%
2. What percentage (0% - 100%) of the time did you spend on...
 - building LEGO robots: 24.29%
 - modeling the architecture using MontiArc: 19.29%
 - modeling behavior using MontiArcAutomaton: 36.43%
 - implementing behavior using Java: 20.00%
3. What percentage (0% - 100%) of the time was wasted because you tried something that was conceptually wrong?
 - Building LEGO robots: 40.71%
 - Modeling the architecture using MontiArc: 12.86%
 - Modeling behavior using MontiArcAutomaton: 33.57%
 - Implementing behavior using Java: 12.86%

APPENDIX B SURVEY MATERIALS

4. What percentage (0% - 100%) of the time was wasted because you tried something that failed due to bugs in the code generator?
 - Building LEGO robots: 2.14%
 - Modeling the architecture using MontiArc: 25.71%
 - Modeling behavior using MontiArcAutomaton: 52.14%
 - Implementing behavior using Java: 5.71%
5. How many times did you revise or recreate the...
 - LEGO robots: 1.33
 - architecture using MontiArc: 1.33
 - behavior using MontiArcAutomaton: 1.80
 - implementation using Java: 0.40
6. Rate from 1 (simple) – 10 (almost impossible) the effort to understand and work on artifacts created by your team members.
 - LEGO robots: 2.86
 - Architecture using MontiArc: 4.43
 - Behavior using MontiArcAutomaton: 6.29
 - Implementation using Java: 4.29
7. Rate the amount of documentation 1 (no documentation) – 10 (well documented) of the artifacts of the team.
 - LEGO robots: 2.57
 - Architecture using MontiArc: 4.43
 - Behavior using MontiArcAutomaton: 5.14
 - Implementation using Java: 4.71
8. Rate your confidence in the correctness of the artifacts created by you from 1 (no confidence) – 10 (works perfectly). Assume a perfectly working code generator.
 - LEGO robots: 7.40
 - Architecture using MontiArc: 7.17
 - Behavior using MontiArcAutomaton: 6.29
 - Implementation using Java: 6.50
9. Rate your confidence in the correctness of the artifacts created by your team members from 1 (no confidence) – 10 (works perfectly). Assume a perfectly working code generator.
 - LEGO robots: 8.29

- Architecture using MontiArc: 7.14
 - Behavior using MontiArcAutomaton: 6.14
 - Implementation using Java: 6.57
10. Rate the effort to fix bugs in the artifacts 1 (simple) – 10 (almost impossible). Assume a perfectly working code generator.
- LEGO robots: 4.71
 - Architecture using MontiArc: 6.29
 - Behavior using MontiArcAutomaton: 6.29
 - Implementation using Java: 3.17
11. How often did you run interactive tests with the following?
- LEGO robots: 33.29
 - Architecture using MontiArc: 15.14
 - Behavior using MontiArcAutomaton: 43.57
 - Implementation using Java: 11.57
12. How many non-interactive regression (aka. junit) tests did you implement?
- LEGO robots: 0.00
 - Architecture using MontiArc: 0.00
 - Behavior using MontiArcAutomaton: 0.00
 - Implementation using Java: 0.29
13. What percentage of the components you've developed does use automata? 57.29
14. What percentage of the components do you think could have been developed using automata? 64.57

B.2 Robotino ROS Python Transport Services

In this lab course, we conducted two surveys using questionnaires. The students answered the first survey a few weeks into development and the second survey at the end of the course. The first questionnaire consists of 12 questions and the second consists of 18 questions.

First Questionnaire

1. What percentage (0% - 100%) of the time did you spend on...
 - understanding MontiArcAutomaton C&C modeling elements: 21.11%
 - understanding the automata behavior language: 21.67%

APPENDIX B SURVEY MATERIALS

- understanding Python: 14.44%
 - understanding ROS: 26.67%
 - understanding the code generation process: 15.00%
2. What percentage (0% - 100%) of the time did you spend on...
 - modeling the C&C structure: 35.00%
 - modeling behavior: 14.44%
 - implementing behavior using Python: 45.00%
3. What percentage (0% - 100%) of the time was wasted because you tried something that was *conceptually wrong* (i.e., you tried something not supposed to be possible that way) while...
 - modeling the C&C structure: 4.44%
 - modeling behavior: 3.33%
 - implementing behavior architecture using Python: 33.89%
 - using ROS modules: 11.11%
4. What percentage (0% - 100%) of the time was wasted because you tried something that failed due to bugs in the code generator while...
 - modeling the C&C structure: 8.89%
 - modeling behavior: 15.00%
 - implementing behavior architecture using Python: 27.78%
5. How many times did you revise or recreate the...
 - C&C models: 1.89
 - behavior models: 0.67
 - Python implementations: 1.56
 - ROS nodes: 0.22
6. Rate from 1 (simple) – 10 (almost impossible) the effort to understand and work on artifacts created by your team members.
 - C&C models: 2.22
 - Behavior models: 3.25
 - Python implementations: 3.00
 - ROS nodes: 4.29
7. Rate the amount of documentation 1 (no documentation) – 10 (well documented) of the artifacts of the team.
 - C&C models: 6.22

- Behavior models: 5.11
 - Python implementations: 7.44
 - ROS nodes: 2.8
8. Rate your confidence in the correctness of the artifacts created by you from 1 (no confidence) – 10 (works perfectly).
- C&C models: 8.63
 - Behavior models: 6.50
 - Python implementations: 6.78
 - ROS nodes: 2.86
9. Rate your confidence in the correctness of the... artifacts created by your team members from 1 (no confidence) – 10 (works perfectly).
- C&C models: 7.75
 - Behavior models: 6.11
 - Python implementations: 6.67
 - ROS nodes: 4.50
10. Rate the effort to fix bugs in the... artifacts 1 (simple) – 10 (almost impossible).
- C&C models: 4.56
 - Behavior models: 5.25
 - Python implementations: 2.78
 - ROS nodes: 4.60
11. What percentage of the components you've developed does use automata? 16.50%
12. What percentage of the components do you think could have been developed using automata? 34.75%

Second Questionnaire

1. What percentage (0% - 100%) of the time did you spend on...
 - understanding MontiArcAutomaton C&C modeling elements: 13.80%
 - understanding the automata behavior language: 14.39%
 - understanding Python: 16.40%
 - understanding ROS: 43.66%
 - understanding the code generation process: 11.76%
2. What percentage (0% - 100%) of the time did you spend on...
 -
 -
 -
 -
 -

- modeling the C&C structure: 16.67%
 - modeling behavior: 18.89%
 - implementing behavior using Python: 64.44%
3. What percentage (0% - 100%) of the time was wasted because you tried something that was *conceptually wrong* (i.e., you tried something not supposed to be possible that way) while...
- modeling the C&C structure: 5.00%
 - modeling behavior: 9.44%
 - implementing behavior architecture using Python: 8.33%
 - using ROS modules: 6.11%
4. What percentage (0% - 100%) of the time was wasted because you tried something that failed due to bugs in the code generator while...
- modeling the C&C structure: 14.44%
 - modeling behavior: 27.78%
 - implementing behavior architecture using Python: 10.00%
5. How many times did you revise or recreate the...
- C&C models: 3.00
 - behavior models: 2.86
 - Python implementations: 6.14
 - ROS nodes: 3.00
6. Rate from 1 (simple) – 10 (almost impossible) the effort to understand and work on artifacts created by your team members.
- C&C models: 4.00
 - Behavior models: 4.56
 - Python implementations: 3.11
 - ROS nodes: 6.14
7. Rate the amount of documentation 1 (no documentation) – 10 (well documented) of the artifacts of the team.
- C&C models: 5.56
 - Behavior models: 4.89
 - Python implementations: 7.89
 - ROS nodes: 5.00

8. Rate your confidence in the correctness of the artifacts created by you from 1 (no confidence) – 10 (works perfectly).
 - C&C models: 8.00
 - Behavior models: 6.50
 - Python implementations: 7.22
 - ROS nodes: 7.33
9. Rate your confidence in the correctness of the... artifacts created by your team members from 1 (no confidence) – 10 (works perfectly).
 - C&C models: 7.88
 - Behavior models: 6.11
 - Python implementations: 7.72
 - ROS nodes: 6.00
10. Rate the effort to fix bugs in the... artifacts 1 (simple) – 10 (almost impossible).
 - C&C models: 4.56
 - Behavior models: 5.22
 - Python implementations: 4.78
 - ROS nodes: 7.25
11. What percentage of the components you've developed does use automata? 10.22%
12. What percentage of the components do you think could have been developed using automata? 19.33%
13. Do you think Scrum helped to enable you develop the common code base? (Yes/No/- Don't know): (7/0/1)
14. How many days do you think would it take to reuse your solution with another (similar, e.g., Roomba) robot? 5.00
15. Please order the technologies by the complexity to understand them.
 - C&C models: 2.89
 - Behavior models: 3.78
 - Python implementations: 1.89
 - ROS nodes: 4.44
 - Continuous integration: 4.40
16. How many hours per week did you ca. spend with this lab? 12.44

17. Rate the different parts of the established Scrum methodology 1 (totally useless) – 10 (great benefit). Rate 0 if you do not know what is meant by the proposed artifact / methodology.
 - Product Backlog: 5.89
 - Impediment Backlog: 6.33
 - Sprint Backlog: 5.11
 - Daily Scrum: 8.56
 - Spring Planning Meeting: 9.11
 - Sprint Retrospective: 5.56
 - Definition Of Done: 6.00
18. Rate the concrete implementation of the different parts of the Scrum methodology 1 (bad implementation) – 10 (best possible integration). Rate 0 if you do not understand the question.
 - Product Backlog: 6.50
 - Impediment Backlog: 6.00
 - Sprint Backlog: 4.89
 - Daily Scrum: 8.22
 - Spring Planning Meeting: 8.56
 - Sprint Retrospective: 6.44
 - Definition Of Done: 6.78

B.3 Robotino SmartSoft Java Transport Services

Both surveys were conducted via questionnaires. The first questionnaire, answered a few weeks into the lab course consists of 12 questions. The second questionnaire, answered at the course's end, consists of 18 questions. The additional six questions inquire about the students' participation and their experiences with Scrum during the course.

First Questionnaire

1. What percentage (0% - 100%) of the time did you spend on...
 - understanding MontiArcAutomaton C&C modeling elements: 14.67%
 - understanding AUTOMATA models: 12.71%
 - understanding the SmartSoft platform: 40.08%
 - understanding the code generation process: 16.21%
2. What percentage (0% - 100%) of the time did you spend on...

- modeling the architecture using MontiArcAutomaton C&C modeling elements: 16.91%
 - modeling behavior using AUTOMATA models: 6.05%
 - implementing behavior using Java: 40.50%
 - implementing technological foundations using SmartSoft: 30.18%
3. What percentage (0% - 100%) of the time was wasted because you tried something that was *conceptually wrong* (i.e., you tried something not supposed to be possible that way)?
- Modeling the architecture using MontiArcAutomaton C&C modeling elements: 12.50%
 - Modeling behavior using AUTOMATA models: 8.50%
 - Implementing behavior architecture using Java: 26.00%
 - Using SmartSoft modules: 35.45%
4. What percentage (0% - 100%) of the time was wasted because you tried something that failed due to bugs in the code generator?
- Modeling the architecture using MontiArcAutomaton C&C modeling elements: 12.50%
 - Modeling behavior using AUTOMATA models: 6.00%
 - Implementing behavior architecture using Java: 24.60%
5. How many times did you revise or recreate the...
- architecture using MontiArcAutomaton C&C modeling elements: 3.00
 - behavior using AUTOMATA models: 0.45
 - implementation using Java: 7.55
 - usage of SmartSoft nodes: 4.55
6. Rate from 1 (simple) – 10 (almost impossible) the effort to understand and work on artifacts created by your team members.
- Architecture using MontiArcAutomaton C&C modeling elements: 3.00
 - Behavior using AUTOMATA models: 2.33
 - Implementation using Java: 3.00
 - Usage of SmartSoft nodes: 7.33
7. Rate the amount of documentation 1 (no documentation) to 10 of the artifacts of the team
- MontiArcAutomaton C&C modeling elements: 4.36
 - AUTOMATA models: 4.89

- Java implementations: 7.00
 - SmartSoft nodes: 3.29
8. Rate your confidence in the correctness of the artifacts created by you from 1 (no confidence) – 10 (works perfectly). Assume a perfectly working code generator.
- MontiArcAutomaton C&C modeling elements: 8.78
 - AUTOMATA models: 8.33
 - Java implementations: 7.78
 - SmartSoft nodes: 4.60
9. Rate your confidence in the correctness of the artifacts created by your team members from 1 (no confidence) – 10 (works perfectly). Assume a perfectly working code generator:
- MontiArcAutomaton C&C modeling elements: 9.00
 - AUTOMATA models: 7.83
 - Java implementations: 7.09
 - SmartSoft nodes: 8.50
10. Rate the effort to fix bugs in the artifacts 1 (simple) – 10 (almost impossible). Assume a perfectly working code generator:
- MontiArcAutomaton C&C modeling elements: 3.73
 - AUTOMATA models: 3.50
 - Java implementations: 4.18
 - SmartSoft nodes: 8.50
11. How often did you run interactive tests with the following?
- MontiArcAutomaton C&C modeling elements: 4.50
 - AUTOMATA models: 3.57
 - Java implementations: 10.10
 - SmartSoft nodes: 1.00
12. How many non-interactive regression (aka. junit) tests did you implement?
- MontiArcAutomaton C&C modeling elements: 0.00
 - AUTOMATA models: 0.00
 - Java implementations: 4.33
 - SmartSoft nodes: 0.00
13. What percentage of the components you've developed does use automata? 11.00%
14. What percentage of the components do you think could have been developed using automata? 20.00%

Second Questionnaire

1. What percentage (0% - 100%) of the time did you spend on...
 - understanding C&C models: 5.00%
 - understanding AUTOMATA models: 5.00%
 - understanding SmartSoft: 46.82%
 - understanding the code generation process: 6.36%
2. What percentage (0% - 100%) of the time did you spend on...
 - modeling the architecture using C&C models: 10.08%
 - modeling behavior using AUTOMATA models: 4.75%
 - implementing behavior using Java: 36.50%
 - implementing technological foundations using SmartSoft: 23.25%
3. What percentage (0% - 100%) of the time was wasted because you tried something that was conceptually wrong (i.e., you tried something not supposed to be possible that way)?
 - Modeling the architecture using C&C models: 8.33%
 - Modeling behavior using AUTOMATA models: 7.92%
 - Implementing behavior architecture using Java: 7.08%
 - Using SmartSoft components: 16.75%
4. What percentage (0% - 100%) of the time was wasted because you tried something that failed due to bugs in the code generator?
 - Modeling the architecture using C&C models: 8.13%
 - Modeling behavior using AUTOMATA models: 8.75%
 - Implementing behavior architecture using Java: 9.38%
5. How many times did you revise or recreate the...
 - architecture using C&C models: 11.00
 - behavior using AUTOMATA models: 1.18
 - implementation using Java: 12.20
 - usage of SmartSoft components: 2.80
6. Rate from 1 (simple) – 10 (almost impossible) the effort to understand and work on artifacts created by your team members.
 - C&C models: 3.83
 - AUTOMATA models: 4.67
 - Java implementations: 4.25

APPENDIX B SURVEY MATERIALS

- SmartSoft components: 8.75
7. Rate the amount of documentation 1 (no documentation) to 10 of the artifacts of the team members.
- C&C models: 7.70
 - AUTOMATA models: 4.38
 - Java implementations: 7.73
 - SmartSoft components: 2.14
8. Rate your confidence in the correctness of the artifacts created by you from 1 (no confidence) – 10 (works perfectly).
- C&C models: 8.20
 - AUTOMATA models: 5.20
 - Java implementations: 7.58
 - SmartSoft components: 7.40
9. Rate your confidence in the correctness of the artifacts created by your team members from 1 (no confidence) – 10 (works perfectly).
- C&C models: 7.17
 - AUTOMATA models: 6.33
 - Java implementations: 7.18
 - SmartSoft components: 5.40
10. Rate the effort to fix bugs in the artifacts 1 (simple) – 10 (almost impossible).
- C&C models: 3.67
 - AUTOMATA models: 2.83
 - Java implementations: 5.25
 - SmartSoft components: 8.22
11. What percentage of the components you've developed does use AUTOMATA models? 6.00
12. What percentage of the components do you think could have been developed using AUTOMATA models? 7.91
13. Do you think Scrum helped to enable you develop the common code base individually? (Yes/No/Don't know): (9/0/3)
14. How many days do you think would it take to reuse your solution with another (similar, e.g., Roomba) robot? 18.90
15. Please order the technologies by the complexity to understand them.

- SmartSoft components: 5.90
 - C&C models: 3.60
 - AUTOMATA models: 3.33
 - Java implementations: 1.70
 - Maven: 3.90
 - Continuous Integration (Jenkins / Nexus): 2.30
16. How hours per week did you ca. spend with this lab? 15.92
17. Rate the different parts of the established Scrum methodology 1 (totally useless) – 10 (great benefit). Rate 0 if you do not know what is meant by the proposed artifact / methodology.
- Product Backlog: 5.25
 - Impediment Backlog: 4.08
 - Sprint Backlog: 5.67
 - Daily Scrum: 7.92
 - Sprint Planning Meeting: 8.50
 - Sprint Retrospective: 6.17
 - Definition of Done: 5.50
18. Rate the concrete implementation of the different parts of the Scrum methodology 1 (bad implementation) – 10 (best possible integration). Rate 0 if you do not understand the question:
- Product Backlog: 3.45
 - Impediment Backlog: 2.91
 - Sprint Backlog: 4.09
 - Daily Scrum: 7.27
 - Sprint Planning Meeting: 7.36
 - Sprint Retrospective: 5.55
 - Definition of Done: 4.18

Appendix C

Kinds of Names in MontiArcAutomaton

MontiArcAutomaton integrates at least four modeling languages for structural aspects of software architectures, their data types, description of code generators, and configuration of applications. Most related entities are referenced by name in or between models of these languages. This appendix thus describes the different types of names and illustrates their usage.

Table C.1: Important kinds of names in MontiArcAutomaton.

Name	Example	Definition	Usage
Architecture root component name	robots.BumperBot	MontiArc-Automaton ADL component type definition (cf. Listing 2.3)	In application configuration models (such as Listing 8.3)
Code generator name	ComponentsJava	In generator description models (e.g., Listing 7.1)	Application configuration models reference it to select generators (Listing 8.1)
Component type name	BumpControl	Component type definition (for instance Listing 5.26)	Subcomponent declarations (e.g. Listing 2.3)
Context condition name	robotarmPy.NoComplexTypes	By constituting Java classes [Vö11]	In MontiCore language definition classes and code generators (Listing 7.5)

Table C.2: Important types of name in MontiArcAutomaton (continued).

Name	Example	Definition	Usage
Data type name	commands.TimerCMD	Class diagram model or Java/P model (Figure 5.2)	Definition of port types in component type definitions (Listing 4.2)
Generator kind name	generatorkinds.Behavior	MontiArc-Automaton infrastructure	Conformance declaration in generator description models (Listing 7.5)
GPL behavior implementation name	hazards.AirQualityChecker	GPL behavior implementation artifacts	Platform-specific atomic components (Section 4.1.1)
Non-terminal production name	MontiArcAutomaton.MAAComponent	MontiCore grammar of a DSL (Section A.1.2)	Processable languages of code generators (Listing 7.7)
Run-time environment name	runtimes.javats	RTE package declaration	Generator descriptions (Listing 7.1) and platform-specific atomic components (Listing 4.2)
Subcomponent instance name	controller	Definition of containing component type (also Listing 5.26)	Definition of connectors (Listing 5.26) and specification of bindings (e.g., Listing 8.1)
Starting point name	componentsjava.MainTemplate or componentsjava.Generator.start()	By the generator's artifacts	Declared by generator description models (Listing 7.7)

Appendix D

Diagram and Listing Tags

This appendix describes the tags and stereotypes used to describe models, artifacts, and their constituents through this thesis. Both tags and stereotypes are used with figures and listings. First, Table D.1 describes the used tags. Afterwards, Table D.2 displays the used stereotypes.

Table D.1: Tags used in listings and figures throughout this thesis.

Tag	Description
AD	UML/P activity diagram
App	Application configuration model
AC	Combined application configuration with generator description
Automata	Automata model
BARC	BehaviorARC model
CpD	Component diagram
FM	FreeMarker template
GBC	Groovy behavior configuration model
GD	Generator description model
Java	Java/P model
LNG	MontiCore language configuration file
MA	MontiArc model
MAA	MontiArcAutomaton model
MCG	MontiCore grammar
MCL	MontiCore languages
PC	Pseudo code
Python	Python code
ROS	ROS graph
ROS-MAA	ROS graph connected to MontiArcAutomaton model
ROS msg	ROS msg model

APPENDIX D DIAGRAM AND LISTING TAGS

Tag	Description
«gen»	Generated element
«hc»	Handcrafted element

Table D.2: Explanation of the stereotypes used throughout this thesis.

Appendix E

Curriculum Vitae

Andreas Wortmann, born March 07, 1982 in Joinville, Brazil

Academic Employment

Since 10/2014	RWTH Aachen University: Team leader automotive and robotics software engineering.
Since 03/2011	RWTH Aachen University: Research and teaching assistant.
04/2006-08/2007	RWTH Aachen University: Student research assistant at the embedded systems laboratory.

Education

Since 03/2011	RWTH Aachen University: Ph.D. studies in Software Engineering
04/2007-09/2011	RWTH Aachen University: Business Informatics studies. Diploma in Business Informatics.
10/2003-08/2010	RWTH Aachen University: Computer Science studies. Diploma in Computer Science.
06/2003	Wirtschaftsschulen Steinfurt: German Abitur.

List of Figures

2.1	Example grammar AST nodes.	12
2.2	MontiCore toolchain overview.	13
2.3	Typical MontiCore activities.	14
2.4	ARC symbol table entries.	15
2.5	Language integration effects on parsers.	18
2.6	Language integration effects on ASTs.	19
2.7	MontiArcBumperBot software architecture.	24
2.8	Data types of BumperBot.	25
3.1	Exploration robot platforms.	30
3.2	Platform-independent software architecture of ExplorerBot.	32
3.3	Platform-specific software architecture of NXTEexplorerBot.	34
3.4	MontiArcAutomaton infrastructure with roles.	39
3.5	The three stages of MontiArcAutomaton.	40
3.6	Instantiation and configuration of MontiArcAutomaton begins with integration component behavior languages.	41
3.7	Modeling a platform-independent architecture.	42
3.8	Monolithic and compositional code generators.	43
4.1	A MontiArcAutomaton variant of BumperBot.	46
4.2	An excerpt of the entries of the MontiArc symbol table that represents structural aspects of C&C software architectures.	53
4.3	MontiArcAutomaton symbol table entries.	54
4.4	MontiArcAutomaton activities overview.	65
4.5	Language integration relies on syntax embedding, symbolic adaptation, and well-formedness rule reuse.	68
4.6	An example for language embedding.	70
4.7	The MontiArcAutomaton ADL language family.	74
4.8	Adaptation between names of different languages enables interpreting references in embedded behavior language properly.	75
4.9	An adapter to use port entries of BehaviorARC as field entries for symbolic integration of FSM models.	76
4.10	Integrating behavior languages into MontiArcAutomaton.	77
4.11	Integrating FSM behavior into the MontiArcAutomaton ADL requires provision of a single new class inheriting from MAAADLTool only.	78
4.12	Quintessential classes of MontiArcAutomaton's Groovy behavior configuration infrastructure.	81

LIST OF FIGURES

4.13 Exemplarily behavior language configuration and artifacts in the context of their containing projects.	82
5.1 Automaton RobotController.	87
5.2 RobotController data types.	88
5.3 AUTOMATA symbol table entries.	95
5.4 The AUTOMATA language family.	96
6.1 Binding constituents overview.	113
6.2 The NavigationBot architecture.	114
6.3 Platform-specific variant of the NavigationBot architecture.	117
6.4 A conflict between two bindings.	118
6.5 Constituents of binding platform-independent architectures.	119
6.6 An implementation library and a interface library.	120
6.7 Components of the interface library BumperBotModels.	121
6.8 JavaNXT component types.	123
6.9 PythonROS component types.	126
6.10 Resolving a clash between two bindings for subcomponent motor.	128
7.1 The prime constituents of generators and their composition.	136
7.2 The generator interfaces of MontiArcAutomaton.	140
7.3 Code generators, interfaces, and kinds.	141
7.4 Generator description symbol table.	147
7.5 The activities required to develop a compositional MontiArcAutomaton code generator.	155
7.6 Example component generator constituents.	156
7.7 The generator orchestrator instantiates participating code generators via the implementations of their descriptions.	157
7.8 Code generator composition activities.	158
7.9 Atomic component Selector with embedded behavior model.	159
7.10 A RTE for Java component implementations.	161
7.11 Java classes generated by component generator ComponentsJavaTS for component Selector.	163
7.12 Artifacts produced by behavior generator AutomataJavaTS	165
7.13 ROS graph of ImprovedBumperBot.	168
7.14 RTE for Python component implementations using ROS.	169
7.15 Artifacts produced by the pythonts component generator.	171
8.1 Constituents of a typical MontiArcAutomaton application.	176
8.2 Application configuration language symbol table.	180
8.3 The MontiArcAutomaton language family.	193
8.4 Typical MontiArcAutomaton activities.	194
8.5 MontiArcAutomaton application constituents.	195
9.1 A robotic coffee service with Lego NXT robots.	198

LIST OF FIGURES

9.2	Confidence in coffee service lab course artifacts.	200
9.3	Robotino ROS transport service robot.	201
9.4	Robotino ROS transport service architecture.	202
9.5	Distribution of the students' lab course time.	203
9.6	Complexity of and confidence in created artifacts.	204
9.7	Top-level architecture of the logistics application.	205
9.8	Time consumption in the Robotino SmartSoft logistics lab.	206
9.9	Complexity of Robotino SmartSoft lab course artifacts.	208
9.10	A toast service robot system using the ROBOTARM behavior language.	209
9.11	BumperBot platform-independent architecture.	210
9.12	iserveU top-level architecture.	214

Listings

2.1	MontiCore grammar of the ARC language to define software components with ports and connectors.	11
2.2	FreeMarker template for transformation of ARC (cf. Listing 2.1) models to Java code.	20
2.3	Textual model of the MontiArcBumperBot component depicted in Figure 2.7. The names for the subcomponents of types UltraSonic, Timer, and Navigation are derived from their types' names.	25
2.4	Textual model of the component type Navigation as depicted in Figure 2.7 and Listing 2.3.	26
4.1	The <i>interface component</i> DistanceSensor.	48
4.2	The component type Clock defines two configuration parameters short and long of which the latter has the default value 10.	48
4.3	The composed component DoubleClock declares two instances clock0 and clock1 of component type Clock. The former applies two arguments to the parameters of Clock and the latter uses its default value for the parameter long.	49
4.4	The atomic component StateBasedController contains the two component variables min (l. 8) and max (l. 9) of type Integer.	50
4.5	The component Recorder defines a configuration parameter (l. 1) and a port (l. 3) of name distance, as well as port (l. 4) and variable (l. 6) of name min. Both are prohibited.	55
4.6	The atomic component BehaviorController contains two behavior models (ll. 3-9).	55
4.7	MultiSensor references two component implementations.	56
4.8	The component type MultiSystemDistanceSensor references two run-time environments (ll. 5-6).	56
4.9	The component type HistogramPrinter declares a variable of name History (l. 5) which produces a warning regarding its name.	57
4.10	The behavior implementation of RobotController (l. 2) violates the context condition to start its name with an upper-case letter.	58
4.11	Component ClockWork declares subcomponents of type Clock with too few arguments (l. 4) and too many arguments (l. 5).	59
4.12	The component type RGBSensor provides a generic type parameter T and contains a port data (l. 3) and variable lastReading of type T (l. 5). Assigning initial values to lastReading thus is prohibited.	59

4.13 Component MaxMotor uses a configuration parameter of generic data type.	60
4.14 The component Validator defines two configuration parameters. The first two parameters feature a default value but the third does not.	60
4.15 The interface component Inverter contains a behavior model (ll. 6-8). .	61
4.16 The interface component Clocks is composed as it contains two subcomponents (ll. 2-3), which is prohibited.	61
4.17 The component AtomicController is atomic and contains a behavior model (ll. 3-5), but declares a run-time environment (l. 6).	62
4.18 The composed component SensorArray erroneously declares a variable (l. 5).	62
4.19 Component NXTUltrasonic (ll. 1-7) references an implementation (l. 5) and declares a RTE (l. 6), i.e., it is platform-specific, but the inheriting component Sensor (ll. 9-12) is an interface component.	63
4.20 The component RegulatedMotor (ll. 7-10) extends from ROSMotor (ll. 1-5)m but does not override its implementation reference.	64
4.21 The component type Logger defines two variables with initial values incompatible to their types (l. 2-3).	64
4.22 The component type IMotor defines the configuration parameter max of type int with a default value "10" of type String.	64
4.23 The composed component DoubleAdder declares three subcomponents in two subcomponent declarations (ll. 7-8).	66
4.24 The transformed MultiAdder component features three subcomponent declarations with a single subcomponent instance each.	67
4.25 The component type DoubleClock after applying the default parameter value to its second subcomponent declaration (l. 3).	67
4.26 A MontiCore language configuration file.	71
4.27 The component Filter contains an embedded FSM model comprising two states and four transitions to describe its behavior.	72
4.28 Groovy behavior configuration model for embedding the FSM.	80
 5.1 Textual model of the automaton RobotController depicted in Figure 5.1.	90
5.2 The declaration of the automaton MotorController begins with the keyword automaton followed by its name and curly brackets.	91
5.3 The automaton TimedMotorController defines three inputs, one output, and two variables (ll. 2-7).	91
5.4 Automaton InitializedMotorController initializes variables (ll. 6-7).	92
5.5 The automaton StatebasedMotorController introduces five states to operate on (ll. 9-10).	92
5.6 The automaton TruckMotorController contains multiple states and transitions to control a truck within speed limits.	94
5.7 The automaton BinaryMotorController uses an alternative (l. 9) to enable a transition for alternative values.	95
5.8 The automaton HRI defines multiple states of the same name (ll. 4-6). .	97

5.9	The automaton Logging declares the states File (l. 4) and Cloud as initial multiple times (ll. 5-6).	98
5.10	The automaton Buffer defines an input (l. 2), an output (l. 3), and a variable (l. 4) of the same name.	98
5.11	The automaton ColorSensor provides no inputs.	99
5.12	The automaton BinaryMotor declares no outputs.	99
5.13	The automaton StatelessBuffer defines no states.	99
5.14	The automaton MapBuilder declares no initial states.	100
5.15	The automaton PersonFollower uses a prohibited Java/P conditional expression for an assignment (ll. 9-10).	101
5.16	The automaton's name begins with a lower-case letter (l. 1).	101
5.17	Automaton Scheduler contains an input, an output, and a variable of names starting with capital letters (ll. 2-4).	102
5.18	Automaton ToastService defines the state toasting that starts with a lower-case letter (l. 4).	102
5.19	Automaton LaneFollower references missing input or variable.	103
5.20	One transition of automaton CoffeeService (ll. 9-10) reads from the output msg, the other (ll. 12-13) assigns a value to the input strength.	103
5.21	The automaton WindowController uses the nonexistent state Idle in an initial state declaration (l. 5) and in a transition (l. 7).	104
5.22	The automaton AssemblyController contains transitions with ambiguous stimuli and actions.	105
5.23	The automaton IrrigationController contains a transition with non-Boolean guard (l. 7).	105
5.24	Automaton ElevatorCabinController contains two incompatible assignments (ll. 9,18) and one incompatible valuation (ll. 12-14).	106
5.25	The automaton AlarmController assigns the special literal NoData to variable latest (ll. 8, 11) and compares NoData to it (l. 13).	107
5.26	The component BumpControl contains an embedded AUTOMATA model with four states and four transitions (ll. 15-35).	108
5.27	Groovy behavior configuration model for embedding part of the AUTOMATA language.	110
6.1	The interface component Timer with configuration parameters and ports of platform-independent CD types.	115
6.2	The bind procedure replaces the types of subcomponents with either platform-specific bound types or new, unambiguous types.	129
7.1	The code generator description model ComponentsJava represents a code generator that translates component models into Java artifacts.	142
7.2	The generator description AutomataPython implements the generator interface generators.BehaviorGenerator.	142

7.3	Generator description <code>ClassdiagramPython</code> defines that the represented generator implements the interface <code>DataTypeGenerator</code> (l. 2) and that it is started via the generators. <code>cdpython.MainTemplate</code> template (l. 4).	143
7.4	Generator description <code>IOTablePython</code> explicates that the represented generator can process models derivable from production <code>TableContent</code> of grammar <code>languages.ihtable.IOTable</code> (l. 6).	143
7.5	Generator description <code>RobotArmPython</code> contains two sets of context conditions (ll. 10-18) describing the contained conditions directly (ll. 10-13) and via abbreviation (ll. 15-18).	145
7.6	The code generator represented by description <code>ComponentsPython</code> produces artifacts compatible to the runtimes. <code>pythonuntimed</code> run-time environment (l. 8).	145
7.7	Generator description <code>ArchitecturePython</code> describes that the represented generator can translate models of the behavior languages <code>TableContent</code> and <code>RobotArmProgram</code> (ll. 10-13).	146
7.8	The generator description <code>ComponentsWithAutomata</code> is not well-formed as it declares two <code>start</code> elements (ll. 4-5).	148
7.9	Generator description <code>IOTablesGroovy</code> references the context condition <code>iotgen.NoInnerComponents</code> two times.	148
7.10	The generator description <code>flowchartPython</code> violates GC1 as its name starts with a lower-case letter (l. 1).	149
7.11	The interface referenced by generator description <code>ArcGenerator</code> cannot be resolved as it does not exist or is unavailable.	150
7.12	The generator description <code>StatechartPython</code> references a template as starting point (l. 4) that is unavailable to the generator it represents.	150
7.13	The generator represented by description <code>RecipePython</code> cannot access the referenced language <code>Recipe.Main</code> (l. 4).	151
7.14	The generator description <code>TimedAutomataPython</code> references the missing context condition <code>tap.RequireDeterminism</code> (ll. 4-5).	151
7.15	The generator represented by <code>PlainComponents</code> cannot access the referenced language <code>lng.Automata.Body</code> (l. 6).	152
7.16	Code generator description <code>AutomataLisp</code> declares that the represented generator is a <code>BehaviorGenerator</code> , but omits designation of a run-time environment.	153
7.17	Generator description of the <code>ComponentsJavaTS</code> component generator with abbreviated package names.	164
7.18	Generator description of behavior generator <code>AutomataJavaTS</code> with abbreviated package names.	166
7.19	A composable class diagram generator based on the generator presented in [Sch12].	166
7.20	The data type <code>MotorCMD</code> of Figure 2.8 as a ROS msg.	167

7.21	Description of a composable component generator that produces Python implementations interfacing the ROS middleware.	172
7.22	A composable data types generator for the production of Python classes and ROS msg types.	172
8.1	The application configuration NavigationBotNXTJava references the NavigationBot software architecture and defines three bindings.	177
8.2	The application configuration BumperBot (l. 1) references the software architecture architecture.NavigationBot (l. 2).	177
8.3	Application configuration BoundBumperBot defines three bindings.	178
8.4	Application configuration BoundBumperBotPythonROS selects four code generators for translation of components (l. 8), class diagrams (l. 9), and two behavior languages (ll. 10-11).	179
8.5	The BumperBotNXTJava application configuration binds the subcomponent sensor twice (ll. 4-8).	181
8.6	This application configuration is invalid as it contains multiple code generators responsible for the same language fragment (ll. 4-5).	182
8.7	Application configuration BumperBotROSNative contains two data type code generators.	182
8.8	An application configuration providing insufficient and wrong code generators.	183
8.9	Application configuration expRob begins with a lower-case letter and consequently raises a warning (ll. 1-2).	184
8.10	This application configuration references the nonexistent software architecture arc.XumperBot, which raises the error depicted.	185
8.11	This application configuration does not bind the interface component sensor of the referenced software architecture, which gives rise to the depicted error.	185
8.12	The application configuration DistributedBB binds a subcomponent of non-interface type (l. 4) and a nonexistent subcomponent (l. 7). Therefore, it gives rise to two errors.	186
8.13	Binding the existing interface subcomponent sensor to an nonexistent component type raises the depicted error.	186
8.14	The application configuration EmbeddedJavaBot references an nonexistent code generator (l. 4) and, thus, is erroneous.	187
8.15	Two of the code generators used by application configuration SimulationBot rely on different run-time environments (ll. 1-2).	187
8.16	The application configuration OfficeBot employs a code generator and a binding of incompatible run-time environments.	188
8.17	The component type DistanceSensor is interface and, therefore, cannot be bound (l. 4).	189

8.18	The depicted application configuration binds the subcomponent <code>sensor</code> to the component type <code>NXTMotor</code> . As <code>NXTMotor</code> does not descend from <code>DistanceSensor</code> , this is invalid.	189
8.19	Application configuration <code>BumperBotCPP</code> contains two bindings with missing (l. 4) and redundant (l. 6) arguments.	190
8.20	The composed component <code>ExtendedBumperBot</code> instantiates <code>RGBASensor</code> and defines its generic parameter to be <code>Integer</code>	191
8.21	Application configuration <code>RobotinoJava</code> binds the generic type of an interface component erroneously.	192
9.1	An excerpt of the Java implementation generated for the component type <code>BumpControl</code> with its most important members and methods.	211
9.2	An excerpt of the Python implementation generated for the component type <code>BumpControl</code> with its most important members and methods.	213
A.1	The quintessential elements of the MontiArc grammar [HRR12] for human comprehension.	250
A.2	The MontiArcAutomaton ADL grammar for human comprehension.	252
A.3	The MontiArcAutomaton ADL grammar for processing by MontiCore.	253
A.4	A simplified AUTOMATA grammar for human comprehension.	256
A.5	The complete AUTOMATA MontiCore grammar.	257
A.6	The generator description grammar for processing by MontiCore.	258
A.7	The application configuration grammar for processing by MontiCore.	259

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: “Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.” Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum11], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR⁺06] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR⁺09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG⁺14] we discuss how to improve reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR⁺06]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Many of our contributions build on UML/P, which is described in the two books [Rum11] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP⁺98] and describe UML semantics using the “System Model” [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH⁺98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR⁺06], [KRV10], [Kra10] allows the spe-

cification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK⁺11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK⁺07], guidelines to define DSLs [KKP⁺09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR⁺11] using deltas [HRRS11] and evolution on deltas [HRRS12]. [GHK⁺07] and [GHK⁺08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR⁺09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR⁺07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP⁺09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a].

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called “System Model” by using mathematical theory in [RKB95, BHP⁺98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH⁺97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH⁺98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK⁺08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR⁺11, HRR⁺11] and to Delta-Simulink [HKM⁺13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK⁺13] describes an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK⁺11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts

[GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a] as well as in building management systems [FLP⁺11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b] that perfectly fit Robotic architectural modelling. The LightRocks [THR⁺13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK⁺07, GHK⁺08]. [HKM⁺13] describes a tool for delta modeling for Simulink [HKM⁺13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO₂ emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP⁺11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems [HHK⁺14], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

References

- [BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.
- [BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.
- [BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.
- [BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.
- [BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.
- [BGH⁺97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737. TU Munich, Germany, 1997.
- [BGH⁺98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.
- [BHP⁺98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.
- [BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.
- [BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.
- [BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.
- [CEG⁺14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

- [CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.
- [CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.
- [EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.
- [FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.
- [FLP⁺11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plessner, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.
- [FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plessner, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.
- [GHK⁺07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.
- [GHK⁺08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.
- [GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.
- [GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.
- [GKR⁺06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.
- [GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.
- [GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.
- [GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.
- [GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

- [GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.
- [HHK⁺13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.
- [HHK⁺14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.
- [HKM⁺13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.
- [HKR⁺07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.
- [HKR⁺09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineering in Research and Practice (SERP'09)*, pages 172–176, July 2009.
- [HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.
- [HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.
- [HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.
- [HRR⁺11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.
- [HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.
- [HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, pages 1 – 10. fortiss GmbH, 2011.
- [HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.
- [HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergeräte-software. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.

- [KKP⁺09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.
- [KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.
- [KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.
- [KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Licher and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering Band 14. Shaker Verlag, Aachen, Deutschland, 2012.
- [Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering Band 1. Shaker Verlag, März 2010.
- [KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapman & Hall, 1996.
- [KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.
- [KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.
- [KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.
- [KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.
- [KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 4735, pages 286–300. Springer, 2007.
- [KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe '08)*, LNBP 11, pages 297–315. Springer, 2008.
- [KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.
- [LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

- [MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.
- [MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.
- [MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.
- [MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.
- [MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.
- [MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.
- [MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.
- [NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.
- [PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.
- [PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.
- [PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.
- [PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.
- [RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.
- [RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

- [RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.
- [RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.
- [Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.
- [Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.
- [Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.
- [Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.
- [Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer Berlin, 2te Edition, September 2011.
- [Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer Berlin, 2te Edition, Juni 2012.
- [Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering Band 11. Shaker Verlag, 2012.
- [SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.
- [THR⁺13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.
- [Völ11] Steven Völkel. *Kompositionale Entwicklung domänenpezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. Shaker Verlag, 2011.
- [Wei12] Ingo Weisemöller. *Generierung domänenpezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering Band 12. Shaker Verlag, 2012.
- [ZPK⁺11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCON-TROL, 2011.