University of Rostock

Toward Composing Variable Structure Models and Their Interfaces

A Case of Intensional Coupling Definitions

Dissertation

zur
Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
der Fakultät für Informatik und Elektrotechnik
der Universität Rostock

betreut durch:
Prof. Dr. rer. nat. habil. Adelinde M. Uhrmacher

vorgelegt von:
Dipl.-Inf. Alexander Steiniger, geb. am 25.03.1983 in Bad Muskau
Satow, den 1. März 2018

urn:nbn:de:gbv:28-diss2018-0141-0

Verteidigung: 23. Juli 2018, Rostock

Gutachter:

- Prof. Dr. rer. nat. habil. Adelinde M. Uhrmacher, Institut für Informatik, Universität Rostock
- Prof. Hans L. M. Vangheluwe, Ph.D., Department of Mathematics and Computer Science, University of Antwerp
- Bernard P. Zeigler, Ph.D., Professor Emeritus, Electrical and Computer Engineering Department, University of Arizona

In memory of
Eberhard Walter Steiniger

Abstract

Modeling and simulation are well established tools to study intriguing systems, both real and imaginary. For this, systems of interest need to be represented by formal models capturing the systems' essential behavior, while abstracting from irrelevant aspects.

Many systems of interest are inherently complex, i. e., they consists of numerous homogeneous or heterogeneous components, where each component can be viewed as a system of its own. The behavior of such a complex system emerges from the interaction of its components and can often not be understood by studying the components in isolation. In addition, the structure of complex systems is often variable, i. e., can change over time. Systems with a time-variant structure are, e. g., socio-technical systems or biological systems.

Component-based modeling and simulation takes the structural complexity of systems under study as well as the correctness and consistency of model compositions representing these systems into account. Creating a model of a complex system by pursuing a component-based approach (i. e., by composition) allows the modeler to reduce (i) the complexity of individual model units, and (ii) the development costs by reusing already existing components. Variable structure modeling, on the other hand, deals with the structural variability of systems, by allowing the modeler to explicitly reflect structure changes in the models representing those systems (i. e., system specifications). Variable structure modeling, similar to component-based modeling, enables the modeler to reduce the complexity of system specifications.

Both component-based modeling and variable structure modeling describe and focus on a similar aspect of a model, that is its structure. However, traditional component-based modeling assumes a static model structure, whereas variable structure modeling often does not provide means as sophisticated as those provided by component-based modeling to specify complex models as an assembly of reusable, self-contained, replaceable, retrievable, customizable, and interoperable components, which can be used in different contexts or by third parties.

In this thesis, we investigate a combination of both kinds of modeling approaches and discuss its implications and limitations. The focus is on a structural consistent specification of couplings in modular, hierarchical models with a variable structure. For this, we exploit intensional definitions, as known from logic, and introduce a novel intensional coupling definition, which allows a concise yet expressive specification of complex communication and interaction patterns in static as well as variable structure models, without the need to worry about structural consistency. The intensional coupling definition is frequently translated into a concrete coupling scheme by the respective simulation algorithm, which takes the current state and structure of the model into account and guarantees the correctness of the derived concrete couplings. Furthermore, we emphasize model (component) interfaces based upon which couplings are defined intensionally. As a proof of concept, the introduced intensional coupling mechanism is realized as a part of ML-DEVS, a modular-hierarchical, system-theoretic modeling formalism, which allows variable structure and multi-level modeling. The abstract simulator, i. e., simulation algorithm, of ML-DEVS illuminates how an intensional coupling definition can be translated into a concrete coupling scheme during simulation while ensuring structural consistency.

At the end of this thesis, we briefly discuss how intensional definitions can help modelers to streamline their models further.

CSS-Classification (2012)

- Computing methodologies~Modeling and simulation~Model development and analysis~Modeling methodologies
- Computing methodologies~Modeling and simulation~Simulation theory~Systems theory
- Computing methodologies~Modeling and simulation~Simulation types and techniques~Discrete-event simulation
- Theory of computation~Formal languages and automata theory~Formalism~Algebraic language theory
- Applied computing~Life and medical science~Systems biology
- Applied computing~Law, social and behavioral sciences~Sociology

Keywords

modeling and simulation, modeling methodology, modeling formalism, closure under coupling, system theory, extensional definition, intensional definition, componentbased modeling and simulation, model composition, composability, modular modeling, hierarchical modeling, variable structure modeling, structure modeling, structure simulation, multi-level modeling, variable structure models, model components, composition, interfaces, complex models, smart environments, mitochondrial networks

Zusammenfassung

Modellierung und Simulation sind etablierte Werkzeuge um interessante Systeme zu studieren, sowohl reale als auch imaginäre. Dafür müssen diese Systeme durch formale Modelle repräsentiert werden, welche das grundlegende Verhalten dieser Systeme widerspiegeln, während sie von irrelevanten Aspekten abstrahieren.

Viele Systeme von Interesse besitzen eine inhärente Komplexität, d. h. sie bestehen aus einer Vielzahl homogener oder heterogener Komponenten. Jede dieser Komponenten kann selbst wieder als ein System betrachtet werden. Das Verhalten solcher komplexen Systeme emergiert aus der Interaktion seiner Komponenten und kann oft nicht verstanden und nachvollzogen werden, wenn die Systemkomponenten in Isolation betrachtet werden. Des Weiteren ist die Struktur von komplexen Systemen oft variabel, d. h. sie kann sich mit der Zeit verändern. Beispiele für Systeme mit zeitveränderlicher Struktur sind z. B. sozio-technische Systeme oder biologische Systeme.

Komponentenbasierte Modellierung und Simulation berücksichtigt die strukturelle Komplexität von Systemen sowie die Korrektheit und Konsistenz von Modellen, die diese Systeme repräsentieren. Modelle komplexer Systeme durch einen komponentenbasierten Ansatz zu erstellen (d. h. durch Komposition), erlaubt es, (i) die Komplexität der individuellen Modellteile zu reduzieren und (ii) die Entwicklungskosten durch die Wiederverwendung von schon existierenden Modellkomponenten zu reduzieren. Die Modellierung von variablen Strukturen beschäftigt sich mit der Strukturvariabilität von komplexen Systemen, in dem sie es dem Modellierer erlaubt, im System auftretende Strukturänderungen explizit in den Modellen dieser Systeme zu reflektieren. Ähnlich wie komponentenbasierte Modellierung, erlaubt es die variable Strukturmodellierung dem Modellierer, die Komplexität von Systemmodellen zu reduzieren.

Sowohl die komponentenbasierte Modellierung als auch die variable Strukturmodellierung beschreiben ähnliche oder gleiche Aspekte eines Modells: deren Struktur. Dennoch nimmt die traditionelle komponentenbasierte Modellierung statische Modellstrukturen an, wohingegen variable Strukturmodellierung oft nicht so ausgefeilte Mittel, wie die komponentenbasierte Modellierung, zur Spezifikation von komplexen Modellen als eine Komposition von wiederverwendbaren, in sich geschlossenen, austauschbaren, abrufbaren, konfigurierbaren und interoperablen Komponenten zur Verfügung stellt, welche in verschieden Kontexten oder durch Dritte benutzt werden können.

In dieser Dissertation untersuchen wir die Kombination beider Arten der Modellierung und diskutieren deren Auswirkungen und Limitierungen. Dabei liegt der Fokus auf einer strukturell-konsistenten Spezifikation von Kopplungen in modular-hierarchischen Modellen mit einer variablen Struktur. Dafür nutzen wir intensionale Definitionen, wie sie aus der Logik bekannt sind, und führen eine neuartige intensionale Definition von Modellkopplungen ein, welche eine kompakte, dennoch ausdrucksstarke Spezifikation von Kommunikations- und Interaktionsmustern in Modellen mit sowohl statischer als auch variabler Struktur ermöglicht, ohne dass sich der Modellierer Gedanken über strukturelle Konsistenz machen muss. Die intensionale Kopplungsdefinition wird ständig von dem entsprechenden Simulationsalgorithmus in ein konkretes Kopplungsschema übersetzt, welcher den aktuellen Zustand und die aktuelle Struktur des Modells berücksichtigt und die Korrektheit der erzeugten konkreten Kopplungen garantiert. Des Weiteren heben wir die Bedeutung und Rolle von (Modell-)Schnittstellen hervor, basierend auf welchen intensionale Kopplungen definiert werden können. Als Machbarkeitsstudie haben wir das Konzept von intensionalen Kopplungsdefinitionen als Teil von

ML-DEVS realisiert. ML-DEVS ist ein modular-hierarchischer, system-theoretischer Modellierungsformalismus, welcher variable Strukturen und Mehrebenenmodellierung erlaubt. Der abstrakte Simulator, d.h. der Simulationsalgorithmus, von ML-DEVS, verdeutlicht, wie eine intensionale Kopplungsdefinition in ein konkretes Kopplungsschema während der Simulation und unter Zusicherung von struktureller Konsistenz umgewandelt werden kann.

Am Ende der Dissertation diskutieren wir kurz, wie intensionale Definitionen Modellierern dabei weiterhelfen können, ihre Modelle weiter zu verschlanken.

CSS-Klassifikation (2012)

- Computing methodologies~Modeling and simulation~Model development and analysis~Modeling methodologies
- Computing methodologies~Modeling and simulation~Simulation theory~Systems theory
- Computing methodologies~Modeling and simulation~Simulation types and techniques~Discrete-event simulation
- Theory of computation~Formal languages and automata theory~Formalism~Algebraic language theory
- Applied computing~Life and medical science~Systems biology
- Applied computing~Law, social and behavioral sciences~Sociology

Schlagwörter

Modellierung und Simulation, Modellierungsmethodik, Modellierungsformalismen, Closure under Coupling, Systemtheorie, extensionale Definitionen, intensionale Definitionen, komponentenbasierte Modellierung und Simulation, Modellkomposition, Komponierbarkeit, modulare Modellierung, hierarchische Modellierung, variable Strukturmodellierung, Mehrebenenmodellierung, variable Strukturmodelle, Modellkomponenten, Komposition, Schnittstellen, komplexe Modelle, smarte Umgebungen, Mitochondriennetzwerke

Acknowledgements

First, I want to thank my supervisor Prof. Adelinde “Lin” Uhrmacher who not only gave me the chance to start my doctoral studies but also the opportunity to work with some great and kind people. I am also grateful for Lin’s endless patience and that she never lost hope that the day of the submission will finally come.

Another thanks go to my reviewers Bernard P. Zeigler and Hans L. M. Vangheluwe, who were willing to review my thesis and whose seminal work, after all, had a huge influence on the content of the thesis.

A very special thanks go to all my former colleagues from the modeling and simulation group at the University of Rostock, especially to Stefan Rybacki, Roland Ewald, Stefan Leye, Fiete Haack Johannes Schützel, Tobias Helms, Danhua Peng, Tom Warnke, Jan Himmelspach, and Carsten Maus. They provided my with the best help and support anyone could wish for.

I also want to thank Prof. Thomas Kirste who gave me the change to become a scholarship holder in the research training group “MuSAMA.”

Furthermore, I am grateful to my former colleagues from the research training group. With a special mention to Michael Zaki, Kristina Yordanova, René Leistikow, Redwan Mohammed, Till Wollenberg, David Gassmann, Anke Lehmann, Enrico Seib, René Zilz, and Axel Radloff.

A very special thanks go to my financée, who supported me in her inspiring, understanding, and affectionate manner.

And finally, last but by no means least, I want to thank my parents and grandparents, who supported me throughout my whole life and made this possible in the first place.

Thanks for all your encouragement. If I missed someone, please don’t be mad at me.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

> For, usually and fitly, the presence of an introduction is held to imply that there is something of consequence and importance to be introduced.
>
> Arthur Machen, 1915

This chapter starts with setting up a basic terminology that is used throughout the remainder of the thesis, before it gives a motivation for the major part of the work that was done during my doctoral studies. At the same time, the underlying problems and questions are identified and briefly described. Then, we list the major contributions, especially in terms of publications, that were created as a result of my research. Finally, the chapter gives a short overview of the structure of this thesis.

1.1 Motivation

Typically, systems that are of interest as research subjects for modeling and simulation, e. g., socio-technical systems such as smart environments, biological systems such as living cells, or demographic systems such as populations of geographical regions, are complex, by nature. They comprise a number of distinctive, often heterogeneous entities (*system components*), such as technical devices (smart environments), cell organelles (cells), or individuals (populations), which can range from a few dozens up to millions. Most of these system components can be viewed as systems on their own (*systems of systems*), because they can be further decomposed into smaller components, i. e., these system components are complex themselves (cf. Jamshidi [2008, pp. 1–2][1]). For instance, smart environments contain different kinds of sensors in large numbers, which, in turn, usually consists of batteries, transceivers, and sensing units [Steiniger & Uhrmacher 2010]. The behavior of such complex systems emerges from the interaction of their components, i. e., constituent parts. This emergent behavior of a complex system may not be traceable when only observing the involved components in isolation and it (the emergent behavior) may even surprise the observer [Zeigler & Muzy 2016]. Many modeling approaches focusing on modeling complex systems, which consists of a large number of components, intrinsically support a modular, hierarchical model construction out of smaller model units (*model components*) that interact with each other. This allows the modeler to reflect the organizational structure of the system of interest on the one hand and to reduce the complexity of the individual model components on the other hand. So instead of creating a huge monolithic model, modular, hierarchical modeling allows us to create smaller, less complex model units that can be coupled with each other. Examples of such modeling approaches are DEVS (Discrete Event System Specification) [Zeigler, Praehofer, & Kim 2000], SysML (Systems Modeling Language[2]), Modelica[3], or process algebras [Baeten 2005; Hoare 1985; Milner 1982]. As we will see later, there are differences between the various approaches, i. e., in the way models are defined.

Going one step further, *component-based modeling* emphasizes the notion of self-contained, interoperable, replaceable, reusable, and customizable *building blocks*, which can be composed regardless of their implementations via some sort of composition methodology/formalism (see [Verbraeck 2004][4]). One, if not the most important aspect of component-based modeling is the reuse of building blocks or components, also by third parties [Chen & Szymanski 2002; Röhl 2008; Verbraeck & Valentin 2008]; for instance, by storing components in a publicly accessible repository, from which they can be retrieved by other modelers. Reusing prefabricated, validated, and reviewed model components reduces the costs and effort of developing models [Sarjoughian & Elamvazhuthi 2009; Szabo & Teo 2007; Valentin, Verbraeck, & Sol 2003], significantly. A public accessibility of components also facilitates the repeatability and reproducibility of simulation studies and thus the traceability of their results. However, to be usable in unforeseen contexts and for different purposes, a component[5] has to announce its functionality and possible configuration capabilities (parameters) by a well-defined *interface* [Röhl & Uhrmacher 2008]. Using such interfaces, allows us to keep composition descriptions separate from the components' implementations, i. e., the specifications of the involved components. In addition to making components exchangeable and reusable, assuring the *correctness* of simulation models by the respective, underlying composition methodology,

[1] Jamshidi [2008, p. 2] defines a system of systems as "large-scale integrated systems that are heterogeneous and independently operable on their own, but are networked together for a common goal."

[2] `http://sysml.org`; last accessed Februrary 2018

[3] `https://www.modelica.org`; last accessed February 2018

[4] In contrast to Verbraeck [2004], we use the terms "building block" and "component" interchangeably. Verbraeck defines a component as the implementation of a building block in a software environment.

[5] Note the difference between system components, model components in modular, hierarchical formalisms such as DEVS, and (model) components from the point of view of component-based modeling. At this point, we focus on the latter.

at least at a syntactic level (*syntactic composability*), is another defining characteristic of component-based modeling [Petty & Weisel 2003a; Szabo & Teo 2007].

Many systems of interest also exhibit structural changes on top of behavioral changes [Zeigler & Praehofer 1990], i. e., their composition and the interaction and behavior patterns of their components can change over time [Uhrmacher 2001]. For instance, populations are made up of individuals that can reproduce and ultimately die. Similarly, in healthy cells, new organelles are created regularly (biogenesis), whereas dysfunctional cellular components are degraded or recycled (autophagy). When modeling such systems it is only natural to capture their time-invariant structure, thus allow modelers to model structure changes explicitly, as part of the system specification (i. e., model). Several modeling approaches have been developed, to address this structure variability, either by extending existing formalisms, e. g., variants of DEVS such as presented by Barros [1995a]; Hagendorf, Pawletta, and Deatcu [2009]; Hu, Zeigler, and Mittal [2005]; Pawletta, Lampe, Pawletta, and Drewelow [1996]; Uhrmacher [2001], or by offering variable structures as salient feature from the outset, e. g., the π-calculus [Milner 1999] or the rule-based modeling language ML-Rules [Maus 2012; Maus, Rybacki, & Uhrmacher 2011]. All of these approaches support a kind of composition over time (which we call *temporal composition*), as they provide structure in the temporal dimension, i. e., they determine under which circumstances a model incarnation replaces another. This allows the modeler to explicitly reflect structure changes of the modeled system by structure transitions in the model of the system (system model). Moreover, introducing different incarnations of a model reduces the complexity of each incarnation, similar to the decomposition of a complex model into smaller units. Structure variability of a system may not only be manifested in a variable composition, but "some systems are characterized by the plasticity of their interfaces" [Uhrmacher, Himmelspach, Röhl, & Ewald 2006] to their surroundings (system environment). For instance, molecules have different binding sites that can become active or inactive and thus allowing or prohibiting bindings with other molecules, respectively. Modeling this kind of structure variability requires dynamic and variable model interfaces (e. g., by introducing variable model ports[6]).

Conventional or traditional model composition, as described at the beginning of this section, i. e., the construction of correct, consistent simulation models out of prefabricated, self-contained components based on a composition methodology or formalism, is done at *configuration time* [Petty & Weisel 2003a], before the simulation model is executed (simulation). Whereas variable structures and interfaces are runtime phenomena, i. e., structure changes occur during model execution [Hu et al. 2005]. Although the modeler specifies the circumstances under which structure changes occur beforehand[7], it is not foreseeable if and when they take place during the simulation at configuration time. Moreover, like state transitions, structure transitions are part of the internal behavior of a model, which we want to keep separate from the definition of the component that encapsulates this model, because a component "hides its internal structure" [Verbraeck 2004]. Components shall only be composed via the well-defined interfaces. For this reason, traditional composition usually assumes a static model structure, which is not changing once a simulation model is created and configured (cf. Chen and Szymanski [2002], Szabo and Teo [2007] Röhl [2008], or Lau and Ntalamagkas [2009]).

Since both kinds of composition, traditional and temporal, refer to the specification of model structure; and both are appealing for modeling the systems we are focusing on (i. e., socio-technical, biological, and demographic systems) intuitively and naturally, we explore the possibilities and limitations of combining both kinds of composition, which seems to be at odds with each other, in this thesis.

When we think about components that encapsulate variable structure models whose

[6] A port is a point of communication via which a model can send or receive values.

[7] Structure transitions become part of the model specification in modeling formalism that support a variable model structure or interfaces.

interfaces may also be dynamic on the one hand and variable composition and communication schemes at the other hand, among others, the following questions arise:

- How and to what extent can structure variability be captured or reflected in a composition methodology, while keeping compositional descriptions and the implementations of components separate?
- What are the implications of structure variability on checking and assuring the correctness and consistency of model compositions at configuration time?

In this thesis we give answers to these and other questions. As starting points for our work, i. e., combining both kinds of compositions, we chose:

1. The platform- and modeling-formalism-independent composition framework COMO (Component-based Modeling) and its underlying interface and composition descriptions [Röhl 2008; Röhl & Uhrmacher 2008].
2. Ideas of the modular, hierarchical modeling formalism ML-DEVS (Multi-Level-DEVS) for (parallel) discrete event, variable structure, and multi-level modeling and simulation [Maus 2008; Uhrmacher et al. 2007; Uhrmacher, Himmelspach, & Ewald 2010].

The former approach does not only allow a platform- and modeling-formalism-independent specification of model compositions [Röhl 2006], in a compact set-theoretical notation (which is implemented by using XML), but it also incorporates component interfaces as first-class abstractions, emphasizing their role in component-based modeling (cf. Verbraeck [2004]). However, Röhl [2008, p. 113] assumes a static model structure and time-invariant interfaces (i. e., static ports), similar to other conventional composition methodologies or frameworks such as CODES[8] [Szabo 2010; Szabo & Teo 2007; Teo & Szabo 2008] or CoSMoS[9] [Sarjoughian & Elamvazhuthi 2009]. Röhl notices that all checks for analyzing the (syntactic) correctness of a composition are carried out before the actual model execution; in the case of a variable composition and dynamic interfaces, this would mean to analyze all possible incarnations of the model structure and interfaces.

The modeling formalism ML-DEVS, on the other hand, incorporates variable structures and interfaces as well as multi-level modeling concepts into parallel discrete event simulation in the tradition of P-DEVS (Parallel DEVS) [Chow & Zeigler 1994]. Both structure variability and multi-level modeling are relevant for modeling complex adaptive systems, such as smart environments, cells, or entire populations.

Multi-level modeling concepts such as modeling systems of interest at different levels of organizational or behavioral abstraction interacting with each other (up- and downward causation) (cf. Maus [2012, pp. 9–39]) played also a role in my doctoral studies (e. g., Steiniger and Uhrmacher [2010], Steiniger, Zinn, Gampe, Willekens, and Uhrmacher [2014], or Steiniger and Uhrmacher [2016]) and are thus reflected in some sections of this thesis.

1.2 Recurring Examples

Throughout the thesis, we focus on two different systems of interest, i. e., modeling subjectives, which, at a first glance, appear to be quite different but share similar characteristics.

Smart Environments The first systems of interest are *smart environments*. Smart environments are *socio-technical systems*, i. e., man-made, open systems in which humans and machines (technical devices) interact with each other, "in such a way that efficiency and humanity would not contradict each other" [Ropohl 1999]. The idea of smart environments

[8] COmposable Discrete-Event scalable Simulation
[9] Component-based System Modeler and Simulator

has emerged from research driven toward the fulfillment of Mark Weiser's vision of *ubiquitous computing*[10] [Nixon et al. 2004, p. 249]. Cook and Das [2004, p. 3] define a smart environments as

> one that is able to acquire and apply knowledge about an environment and also to adapt to its inhabitants [the users of smart environments] in order to improve their experience in that environment [the users' surroundings].

With respect to their purpose, Kirste [2006, p. 322] gives a more specific definition: "smart environments are physical spaces that are able to react to activities of users, in a way that assist the users in achieving their objectives in this environment." Both definitions describe rather the behavior of smart environments from a macroscopic point of view than their structure or internal functioning. Detailing the actual composition of smart environments, Poslad [2009, p. 30] writes:

> A smart environment consists of a set of networked devices that have some connection to the physical world. [...] the devices that comprise a smart environment usually execute a single predefined task [...]. Smart environment devices may also be fixed in the physical world at a location [stationary] or mobile [...].

Such devices can be sensors, controllers, or computers that are embedded or operate in the respective physical environment [Poslad 2009, p. 7]. In other words, a smart environment is a "region of the real world that is extensively equipped with sensors, actuators, and computing components" [Nixon et al. 2004, p. 249, attributed to Nixon, Lacey, and Dobson [2000]]. In Steiniger and Uhrmacher [2010], we describe the different, heterogeneous components of smart environments, in more detail. The structure or composition of smart environments is time-variant, i. e., can change over time. For instance, users can enter or leave smart environments and new devices can be embedded into the device ensemble ad-hoc or battery-powered devices can run out of power. Herein, we focus on small-scale smart environments, in particular on *smart meeting rooms* that support their inhabitants in conducting collaborative and interactive meetings (cf. Heider and Kirste [2005], Park, Moon, Hwang, and Yeom [2007], Hein, Burghardt, Giersich, and Kirste [2009], or Steiniger, Krüger, and Uhrmacher [2012]). Figure 1.1 shows such an smart meeting room[11].

Eukaryotic Cells The other systems of interest herein are *eukaryotic cells*, which are biological or natural systems (in contrast to smart environments). Eukaryotic cells are the cells of *eukaryotes*, which are single-celled or multicellular organisms whose cells contain a membrane-bound nucleus [van der Giezen 2011], a special cell organelle[12]. In addition to cell nuclei, eukaryotic cells contain other organelles such as mitochondria [Patel, Shirihai, & Huang 2013]. Mitochondria are of crucial importance, since they produce the energy—in form of ATP (*adenosine triphosphate*)—that is necessary to keep the cells working and sustain life. Therefore, mitochondria are also widely know as the "powerhouses" of cells [van der Giezen 2011]. During the lifetime of an eukaryotic cell, new mitochondria can be produced (*biogenesis*) and existing ones can be degraded (*autophagy*), especially when their functionality is impaired. Mitochondria can also be actively transported along cytoskeletal structures, such as microtubules [R. L. Morris & Hollenbeck 1995]. Mitochondria that are close to

[10] Weiser coined the term "ubiquitous computing" in his seminal article *"The Computer for the 21st Century"* [Weiser 1999], in which he described his vision of a world in which computers support the users in their everyday life unobtrusively and eventually "disappear." The term *pervasive computing* is often used as a synonym for ubiquitous computing. According to Nixon, Wagealla, English, and Terzis [2004], other synonyms are *ambient computing*, *active spaces*, or *context-aware computing*.

[11] Despite almost two decades of research, smart environments are not mass-market products yet, that can be bought and configured ad-hoc. Existing environments are often custom-made or of experimental nature.

[12] Cell organelles are one essential type of building blocks of eukaryotic cells.

Figure 1.1: The Smart Appliance Lab (*SmartLab*) of the University of Rostock, which is equipped with numerous, heterogeneous devices such as projectors, microphones, cameras, passive and active location sensors, temperature sensors, notebooks, and air-conditioning.

each other form networks and interact directly and indirectly [Patel et al. 2013]. More precisely, a mitochondrion can fuse with another one in its vicinity and "exchange both soluble and membrane-bound components," which has an influence of the overall health of the mitochondrial network. Fused mitochondria will fission after a while, resulting in a "frequent cycles of fusion and fission" [Patel et al. 2013]. Indirect interaction between cells takes place, e. g.,, when a mitochondrion is producing reactive oxygen species (ROS) that are inhibiting other mitochondria in the mitochondrial network [Park, Lee, & Choi 2011], where the cell cytoplasm, the basic substance within a cell excluding the nucleus, serves as the interaction medium[13]. We conclude that the structure of eukaryotic cells is extremely dynamic, like the structure of other biological systems.

Toward the end of my doctoral studies, we were also investigating demographic systems in cooperation with the Max Planck Institute for Demographic Research[14] (cf. Steiniger et al. [2014] and Warnke, Klabunde, Steiniger, Willekens, and Uhrmacher [2015]). Of particular interest were the hypotheses underlying the *linked lives model* [Noble et al. 2012], i. e., that the lives of individuals can be linked and that these links have an influence on the individuals' life courses. The purpose of the model is the prediction of the supply of and demand for social care in modern UK based on basic demographic processes, including mortality, fertility, health changes, migration, and the formation and dissolution of partnerships and households. Like in biological systems, the structure inherent to demographic systems (e. g., the population of a geographic region) is dynamic. However, the structure is different from the structure of smart environments or living cells in the way that organizational and communicational relationships between the entities in demographic systems form complex networks rather than hierarchies with exclusive memberships (cf. Steiniger et al. [2014] or Warnke et al. [2015]). For instance, individuals can be members of different organizations, such as associations, unions, or sport clubs, at the same time. Therefore, demographic systems do not play a role in the main body of this thesis; only in the last part, we come back to them, when writing about domain-specific languages and their role in modeling.

[13] For a general introduction about the concept of direct and indirect interaction refer to Odell, Van Dyke Parunak, Fleischer, and Brueckner [2003] and Weyns, Helleboogh, Holvoet, and Schumacher [2009].

[14] `https://www.demogr.mpg.de/en/`; last accessed February 2018

1.3 Contribution

The main contribution of my doctoral studies and this thesis is twofold:

1. The revision of the composition framework COMO and its underlying formalism to cope with variable interfaces and structures.
2. The revision, formal definition, and extension of the modular, hierarchical modeling formalism ML-DEVS for discrete event, multi-level, and variable structure modeling, whereby a particular focus was on the introduction of a novel, intensional coupling mechanism, called multi-couplings[15].

Starting from the composition framework COMO and its underlying formalism for describing interfaces and compositions in a modeling-formalism-independent manner (see Himmelspach, Röhl, and Uhrmacher [2010]; Röhl [2006, 2008]; Röhl and Morgenstern [2007]; Röhl and Uhrmacher [2006, 2008]), which assumes static model structures, we extended the underlying formalism and adapted the framework to reflect variable interfaces and interfaces. This extension included:

- The usage of supersets of ports and components, in interface, component, and composition descriptions. These supersets contain all potential ports and components that can become available during model execution, respectively.
- The introduction of an intensional coupling definition, called *loose connections*, which is frequently translated into a concrete coupling scheme during simulation, whenever structure changes occur.
- The further decoupling of interfaces and components descriptions, so that different components can "implement" the same interface.
- The introduction of criteria for analyzing the syntactic correctness of a model composition (and thus the correctness of the derived model) with variable interfaces and structures, beyond the initial model state and configuration (which is known when deriving and configuring the executable simulation model).
- A corresponding adaptation of the underlying composition and analysis methodology.
- A proof-of-concept implementation that makes use of (i) the modeling formalism ML-DEVS as a target for the transformation of compositional descriptions and component implementations into executable simulation models; (ii) XML as a machine-interpretable and exchangeable representation of the set-theoretically defined formalism; and (iii) the modeling and simulation framework JAMES II [Himmelspach & Uhrmacher 2007, 2009] as a simulation engine on top of which COMO resides[16].

The following publication was a result of our work on COMO:

Steiniger, A. and Uhrmacher, A. M. (2013). "Composing Variable Structure Models: A Revision of COMO." In *Proceedings of the 3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications* (SIMULTECH 2013). pp. 286–93.

The second, more important aspect of my work refers to the revision and extension of the modeling formalism ML-DEVS, which served as means for specifying almost all of the models that we have developed in the course of my doctoral studies and research. This revision is

[15] Please note that we keep the term "multi-couplings" for historic reasons. However, the multi-couplings of the first iteration of ML-DEVS are different from the coupling concept developed during my doctoral studies.

[16] COMO is not limited to JAMES II for executing models.

motivated by the characteristics of the systems of interest[17] and substantial, which we see when comparing the original definition of the formalism, as introduced by Uhrmacher et al. [2007], with the iteration presented herein. More specifically, the work on ML-DEVS includes the following aspects:

- The revision of the entire formal specification and notation of ML-DEVS and its underlying concepts.
- The introduction of publicly accessible states that are used for *upward causation* and propagating information between neighboring levels (*global information*); and that are separate from the regular ports.
- The introduction and elevation of model interfaces and their runtime instances as first-class concepts in the model specification, where the public state of a model is also part of its interface.
- The introduction of a powerful, expressive, and intensional coupling mechanism, which is based upon the interface definitions.
- The implementation of a reference algorithm for the translation of the novel coupling definition into concrete and consistent coupling schemes during simulation (correctness by construction), while adhering to the intensional definitions and taking the current model state into account.
- A respective adaptation and extension of the abstract simulator of ML-DEVS.
- The provision of a formal proof of the closure under coupling of ML-DEVS (see the supplementary material to Steiniger and Uhrmacher [2016]).
- The demonstration of a system morphism between ML-DEVS and general I\O system, as defined by Zeigler et al. [2000, pp. 108–16], showing that a arbitrary Micro-DEVS model can be translated in a behaviorally equivalent I\O system that ML-DEVS is a system specification formalism.
- A proof-of-concept implementation of the model specification in the high-level programming language Java and the abstract simulator of ML-DEVS in the open-source, Java-based modeling and simulation framework JAMES II.

The proof of the closure under coupling for ML-DEVS also provides a blueprint for showing that ML-DEVS is behaviorally equivalent to static structure models, e. g., specified in P-DEVS (Parallel Discrete Event System Specification) [Chow & Zeigler 1994; Chow, Zeigler, & Kim 1994]. It also indicates that an arbitrary, hierarchical ML-DEVS model can be flattened. The results of this work were presented and elaborated in the following two publications, with a special focus on the latter of both.

Steiniger, A., Krüger, F., and Uhrmacher, A. M. (2012). "Modeling Agents and their Environment in Multi-Level-DEVS." In *Proceedings of the 2012 Winter Simulation Conference* (WSC'12). Article No. 233.

Steiniger, A. and Uhrmacher, A. M. (2016). "Intensional Couplings in Variable Structure Models: An Exploration Based on Multilevel-DEVS." In *ACM Transactions on Modeling and Computer Simulation* (TOMACS), 26(2). pp. 9-1–9-27.

We also explored the suitability of ML-DEVS for creating continuous time, multi-level models in the domain of computational demography in:

Steiniger, A., Zinn S., Gampe J., Willekens F., and Uhrmacher A. M. (2014). The Role of Languages for Modeling and Simulating Continuous-Time Multi-Level

[17] Chapter 9 elaborates on this subject.

Models in Demography (invited paper). In *Proceedings of the 2014 Winter Simulation Conference* (WSC '14), pp. 2978–89

In addition to the aforementioned work that is presented in this thesis, we also cooperated with colleagues from the chair of Mobile Multimedia Information Systems (MMIS) of the Institute of Computer Science at the University of Rostock[18] on simulation-based testing and the evaluation of context-aware applications, such as probabilistic activity recognition or proactive user assistance in smart environments[19]. Similar to approaches such as Bylund and Espinoza [2001, 2002], Vijayaraghavan and Barton [2001], Barton and Vijayaraghavan [2002], Sanmugalingam and Coulouris [2002], Huebscher and McCann [2004], Nishikawa et al. [2006], M. Martin and Nurmi [2006], Park et al. [2007], McGlinn, O'Neill, Gibney, O'Sullivan, and Lewis [2010], Helal et al. [2011] or Campuzano, Garcia-Valverde, Garcia-Sola, and Botia [2011], we used modeling and simulation as a means to test applications[20]. Instead of "plugging" the application under test into a real-life environment inhabiting human test subjects, we plugged the application into a simulation or we used simulation to systematically create test data, if the feedback of the modeled system to the application was not of interest. For this, we used the modeling formalisms ML-DEVS [Krüger, Steiniger, Bader, & Kirste 2012] and PepiDEVS[21] [Nyolt, Steiniger, Bader, & Kirste 2013, 2015] to model smart environments (i. e., the context). Again, JAMES II served as simulation environment with which the application under test was interacting. The results of this work were presented and discussed in the following publications, to which I contributed:

Krüger, F., Steiniger, A., Bader, S., and Kirste, T. (2012). "Evaluating the robustness of activity recognition using computational causal behavior models." In *Proceedings of the 2012 ACM Conference on Ubiquituous Computing* (UbiComp 2012). pp. 1066–74.

Nyolt, M., Steiniger, A., Bader, S., and Kirste, T. (2013). "Describing and Evaluating Assistance using APDL." In Proceedings of the 2013 International SSMCS Workshop. pp. 38–49

Nyolt, M., Steiniger, A., Bader, S., and Kirste, T. (2015). "Describing and Evaluating Assistance using APDL." In *Smart Modeling and Simulation for Complex Systems: Practice and Theory.* pp. 59–81

1.4 Structure and Notations

The remainder of the thesis is divided into three main parts.

The first part introduces the basic terminology, which is supposed to serve as a basis for a common understanding, and all relevant concepts and previous work, with a particular focus on extensional and intensional definitions. Part I surveys related work from discrete event, component based, and variable structure modeling and simulation, where the focus is rather on modeling than simulation.

The second part of the thesis describes the underlying problems and research questions and it contains the main contribution of my work: the introduction and implementation of an intensional coupling mechanism easing the definition of couplings, particularly in variable structure models, when using a modeling approach that is based on the reactive systems metaphor.

[18] `https://www.mmis.informatik.uni-rostock.de/en/`; last accessed February 2018

[19] First ideas of using modeling and simulation for testing were already discussed in Steiniger and Uhrmacher [2010]

[20] The aforementioned approaches are sometimes called *context simulation* or *context simulators*, since they "simulate" the context information that an context-aware application is reacting and adapting to.

[21] Parallel external process interface DEVS; a DEVS variant that allows integrating external processes into simulation [Himmelspach 2007]

Finally, the third and last part of the thesis draws a conclusion and outlines potential topics, we came across my doctoral studies, for future work.

Relevant terms are written in quotes or in *italic* at the first occurrence in a chapter. We also use italics to emphasize other interesting terms and words. Direct quotes are put into quotation marks or they are indented and separated from the surrounding text, particularly when we are using longer quotes. Remarks on formal notations, as used throughout the thesis, can be found in Appendix A.

Part I

Basics and Background

2 Basic Terminology

> Much of the discussion about socialism and individualism is entirely pointless, because of failure to agree on terminology.
>
> Theodore Roosevelt, Jr., 1915

Although different attempts were made to establish a common terminology in the realm of modeling and simulation (M&S), such as the "IEEE Standard Glossary of Modeling and Simulation Terminology" [1989][1], the *DoD Modeling and Simulation Glossary* [Department of Defense 1998], or the *ACM SIGSIM Modeling and Simulation Glossary*[2], no such uniform and universal terminology exists—unfortunately. Even fundamental terms, such as *simulation* or *simulator*, are used differently in the literature. Moreover, different terms are used to refer to the same or similar concepts (e. g., *symbiotic simulation* and *hardware-in-the-loop simulation*[3]), or terms are used whose meanings remain vague due to the absence of clear-cut definitions. For this reason, the following chapter gives a brief overview and definitions of the most central terms and concepts used throughout the thesis. Other important terms and concepts will be defined or characterized when they first occur in the text.

[1] This standard is already withdrawn.

[2] `http://www.acm-sigsim-mskr.org/glossary.htm`; last accessed February 2018

[3] Both terms refer to a simulation paradigm in which the simulation system (simulation) interacts with a physical system (hardware), i. e., the physical system is embedded into the simulation.

2.1 Systems

The first and one of the most fundamental terms is *system*. Systems in general or phenomena occurring in certain systems in particular are the subjects of interest in M&S. Law and Kelton [2000, p. 3, attributed to Schmidt and Taylor [1970]] define a system as

> a collection of entities, e.g., people or machines, that act and interact together toward the accomplishment of some logical end,

where "an entity is an object of interest in the system" [Banks, Carson II, Nelson, & Nicol 2000, p. 10] and "relationships among those entities exists" [Miller 1978, p. 16][4]. At this point, we already assume that systems consist of some sort of interacting entities, parts, objects, elements, constituents, or components (called *system components* in the following). Each system component can be viewed as a system of its own, i. e., is a subsystem of the composed system. We call such a composed system also *complex system*[5], whose complexity is defined not only by the number of their homogeneous or heterogeneous components but also by the dynamic interaction between them [Maus 2012, p. 9]. Gallagher and Appenzeller [1999] describe a complex system as "one whose properties are not fully explained by an understanding of its component parts [components]." In complex systems, "we observe group or macroscopic behavior emerging from individual actions and interactions" [Hoekstra, Kroc, & Sloot 2010, p. 3].

When talking about systems and their dynamics (i. e., behavior), the systems' states and their evolution over time are of particular interest. The *state of a system* (system state) is defined as a "collection of variables necessary to describe a system at a particular time" [Law & Kelton 2000, p. 3]. These *state variables* can be related to the composition of the system (e. g., the number of certain entities) or to the internal functioning of an individual entity (e. g., a characteristic property). Generally and most often, systems change their states over time, e. g.,. as a result to extrinsic or intrinsic factors. A system whose state (state variables) does not remain constant over time is called *dynamic system*, whereas *static systems* are those whose states are assumed to remain constant in time (cf. Karnopp, Margolis, and Rosenberg [2012, p. 3]). Depending on how and when state variables of a system change over time, Law and Kelton [2000, p. 3] and Banks et al. [2000, p. 12] categorize a system to be either *discrete* or *continuous*. A discrete system is one for which the state variables change instantaneously at separated points in time [Law & Kelton 2000, p. 3], where the overall set of such points in time is discrete[6] [Banks et al. 2000, p. 12]. In contrast, "a continuous system is one for which the state variables change continuously with respect to time" [Law & Kelton 2000, p. 3]. Even though, in practice, most systems are neither entirely discrete nor entirely continuous, often one type of change predominates so that we can categorize the system either as being discrete or continuous [Law & Kelton 2000, p. 3]. Note that this categorization does not make any assumptions about the types of the state variables.

A term closely related to the notion of a system is *system environment*, as "a system is often affected by changes [events] occurring outside the system [i. e., in the system environment]" [Banks et al. 2000, p. 9]. "Each system has its own environment and is in fact a subsystem of some broader system" [Shannon 1975, p. 37]. Based on the considerations of Gaines [1979], Cellier [1991, p. 2] concludes:

> The largest possible system of all is the universe. Whenever we decide to cut out a piece of the universe such that we clearly say what is inside that piece (belongs

[4] These relationships exist not only in *living systems*, on which Miller [1978] is focusing, but in systems in general, including non-living systems (e. g., purely technical systems).

[5] In fact, many systems of interest can be considered as complex. Hence, when we talking about systems in the thesis, we usually refer to complex systems.

[6] Here, a discrete set is either finite or countable infinite.

> to that piece), and what is outside that piece (does not belong to that piece), we define a new 'system.'

The question of what belongs to a system or is already part of its environment (i. e., how to define the boundary between both), depends on the purpose of studying a certain system of interest [Banks et al. 2000, p. 9].

2.2 Models

This brings us to the second fundamental term: *model*. Banks et al. [2000, p. 13] define a model as "a representation of a system for the purpose of studying." Instead of representation, often the term "abstraction" is used to describe a model (e. g., by Smith and Smith [1977]). Cellier [1991, p. 5, attributed to Minsky [1965]] gives the following, more specific but still rather general definition of a model:

> A model (M) for a system (S) and an experiment (E) is anything to which E can be applied in order to answer questions about S.

However, this definition introduces another term, i. e., *experiment*, that yet has to be defined, which is done below. In both definitions, a model refers to a certain system that the model represents and has a purpose, which is answering questions about the system of study. Banks et al. [2000, p. 3] give the following, more concrete characterization of a model, to which we will stick:

> The behavior of a system as it evolves over time is studied by developing a simulation model. This model usually takes the form of a set of assumptions [hypotheses] concerning the operation of the system. These assumptions are expressed in mathematical, logical, and symbolic relationships between the entities, or objects of interest, of the system.

Furthermore, Karnopp et al. [2012, p. 5] write that

> It is important, then, to realize that *no system can be modeled exactly* and that any competent system designer needs to have a procedure for constructing a variety of system models of varying complexity so as to find the simplest model capable of answering the questions about the system under study.

Leye [2013, pp. 3–4] concludes that a model has the following three properties:

- Reference: the model represents a system of interest.
- Abstraction: the model reflects a subset of the system's features.
- Purpose: the model shall answer one or more questions about the system of interest.

Another interesting implication is that a model itself is qualified to be called a system [Cellier 1991, p. 5]. Thus, the boundaries between system and model are blurred. Moreover, the above definitions do not imply the nature of a model. Models can, for instance, be:

- mental/conceptual (they exist only in the mind of the modeler),
- physical/iconic (they are tangible),
- verbal/textual (they are described informally by natural language),
- mathematical/formal (they are specified mathematically or in a formalism),
- computational/executable/programmed (they can be directly executed on a computer in terms of a program that is usually defined in a high-level programming language).

Note that every *computational model* specified as a computer program can be considered as a *formal model* at the same time. However not every formal model may be directly executable on a computer without further ado. Moreover, the term "conceptual model" is used differently in the literature. For instance, Banks [1998, pp. 15–7] implies that a conceptual model is already somehow formalized. Herein, we adhere to the definition of Nance [1994], according to which a conceptual model exists in the mind of the modeler. Further types and categories of model are distinguished in M&S, such as discussed by Leemis and Park [2006, p. 2][7] and depicted in Figure 2.1.

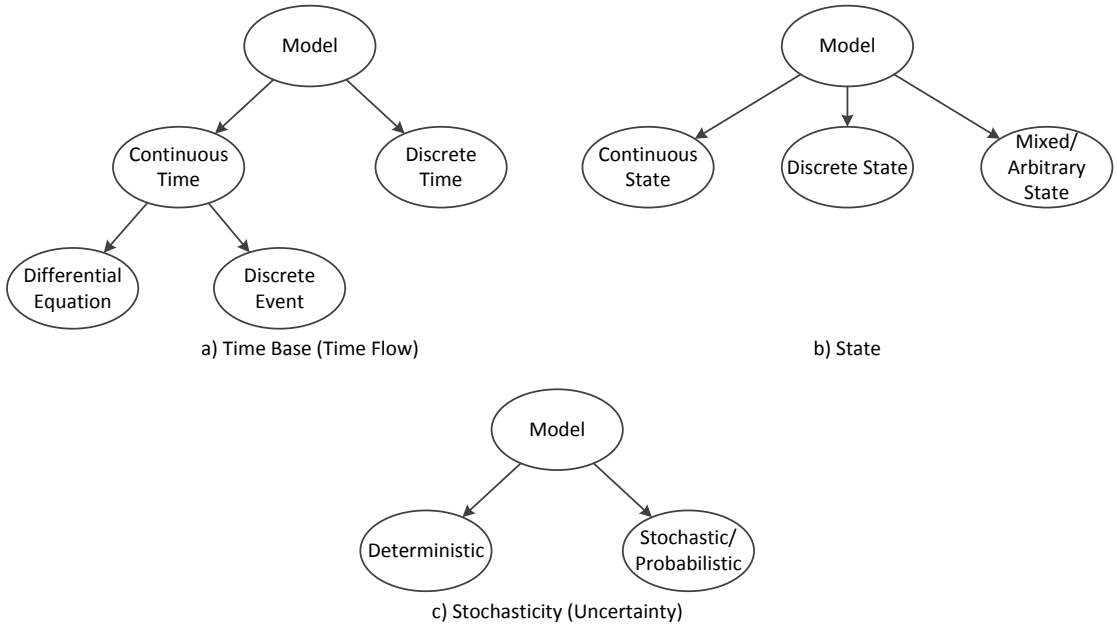

Figure 2.1: Different categories as given by Zeigler [1976, p. 22] according to which models can be categorized, i. e., the time base, the nature of the state, and the incorporation of random variables.

Like systems, the models we are focusing on have a state that can change, reflecting the dynamics of the modeled systems (*dynamic systems*). Moreover, models have often, but not necessarily, inputs and outputs.

The definition of a model given by Cellier makes use of the term "experiment." Cellier [1991, p. 4] defines an experiment as follows:

> An experiment is the process of extracting data from a system by exerting it through its inputs.

This rather general definition does neither imply the nature of an experiment nor does it make a connection to the term "model."

2.3 Simulation

This leads us to the next fundamental term "simulation," for which different definitions can be found in the literature. We adhere to a concise definition given by Korn and Wait [1978, as cited by Cellier [1991, p. 6]]:

> A simulation is an experiment performed on a model.

[7] Leemis and Park [2006, p. 2] distinguish between deterministic and stochastic, dynamic and static, and continuous and discrete models.

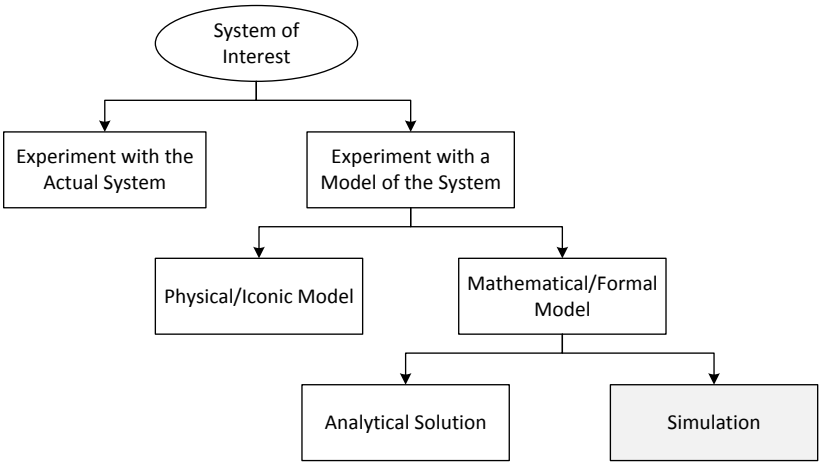

Figure 2.2: Different ways to study a system of interest (adapted from [Law & Kelton 2000, p. 4]).

This definition finally relates a simulation to an experiment and a model (of a system). In other words, in a simulation we experiment with a model instead of the system of interest, where the model surrogates or mimics the actual system in the simulation. Still, the above definition does not imply that a simulation is conducted or executed on a computer. However, similar to Cellier [1991], our focus is on *computer simulation* (also called *computational simulation* or *mathematical simulation*), hence experiments that are performed on a computer[8]. Therefore, we concentrate on conceptual formal models that can be translated into computational models; and we occasionally refer to computer simulation simply as the execution of such models on a computer. "The computational paradigm [simulation] plays a fundamental role in situations where analytical descriptions of the observed phenomena are not tractable and/or out of reach of direct experimentation" [Hoekstra et al. 2010, p. 2]. So in general, simulation is used when the model is too complex, excluding the possibility of an analytical solution [Law & Kelton 2000, p. 5]. More information on when simulation is appropriate and when it is not can be found in Banks et al. [2000, pp. 1–7]. Figure 2.2 shows the different ways to study a system of interest and illuminates the relationship between the terms defined so far.

2.4 Modeling

Eventually, *modeling* or *system modeling* (as systems are our subjects to model) can be considered as the process of building, developing, constructing, or creating models, while having certain questions about a system of interest in mind that shall be answered by the models to create. However, creating models is anything but simple. Shannon [1975, p. 19] notes that the process of creating a model of a system "can best be described as an intuitive art." In contrast, Savory and Mackulak [1994] argue that modeling is a science that can be taught and that the need of a certain amount of intuition to create models "does not constitute an art." Banks et al. [2000, p. 15] write that the process of creating models "is probably as much art as science." Regardless of whether we consider modeling as science or art, it is the most difficult part of a simulation study [Rybacki, Haack, Wolf, & Uhrmacher 2014] and the question of how to create models remains. In general, modeling requires "an ability to analyze a problem, abstract from its essential features, select and modify basic assumptions that characterize the system, and then enrich and elaborate the model until a useful approximation results" [Shannon 1975, p. 20]. So modeling can be considered as an

[8] Experiments that are performed on a computer are also called *in-silico experiments*, where the computer serves as laboratory, or rather as *dry lab*.

evolutionary, iterative process, in which we often start with a simple model and increase its complexity gradually. However, the complexity does not need to "exceed that [complexity] required to accomplish the purposes for which the model is intended" [Banks et al. 2000, p. 15]. W. T. Morris [1967] propose seven general guidelines for creating appropriate models.

2.5 Validity

But when is a model appropriate? Or, more importantly, when not? In other words, we want to know whether a given model really represents (the behavior of) a certain system of interest. This question is closely related to the term "model validity." Balci [1997] describes validity simply as *behavioral accuracy*. Based on Zeigler et al. [2000, pp. 30–1], we define the validity of a model as follows:

> The validity is the degree to which the behavior of a model agrees with the observed behavior of its system counterpart (i. e., the system represented by the model) with respect to a certain question in mind and an experiment.

As Cellier [1991, pp. 5–7] points out, statements on the validity of a model can only be made in context of a certain experiment. A model of a system that is valid for a certain experiment, may not be valid for another experiment. However, validity is only one dimension of the appropriateness of a model. A model may be valid but not easy to understand by anyone else than the modeler, i. e., a third party. For a discussion about criteria for a good model refer to [Shannon 1975, pp. 21–2].

2.6 Simulators

When we have created a model (valid or not), a *simulator* is responsible for executing the model. Zeigler et al. [2000, p. 30] give the following definition of a simulator:

> A simulator is any computation system (such as single processor, a processor network, the human mind, or more abstractly an algorithm) capable of executing a model to generate its behavior.

The *behavior of a model* is its outer manifestation (input/output behavior) or, in simple terms, "the behavior is what the model does" [Zeigler 1976, p. 5]. As our focus is on computer simulation, by simulator we refer to a *simulation algorithm*, which eventually will be implemented as some sort of computer program. According to Zeigler et al. [2000, p. 30] the separation of simulator and model provides several benefits, such as the same model can be executed by different simulators. Moreover, this kind of *separation of concerns* allows the modeler to concentrate on building models, whereas simulation experts can focus on developing efficient simulation algorithms, i. e., simulators. In the context of computer simulation, the term "simulator" sometimes refers not only to a simulation algorithm but to an entire *simulation system*, including auxiliary and additional functionality that is not part of the actual simulation algorithm. Here we distinguish between simulation algorithm and simulation system[9].

2.7 Modeling Formalisms

Another term that plays an important role in this thesis is *modeling formalism*. A modeling formalism allows us to translate conceptual or informally given models into formal models.

[9] We use the term "simulation system" in a rather general sense. In particular, a simulation system can be a simple library, an environment, or a comprehensive framework. A discussion on different kinds of simulation systems is given by Himmelspach [2012].

Sarjoughian [2006, attributed to Sarjoughian and Zeigler [2000]] characterize a modeling formalism as follows:

> A modeling formalism can be defined to consists of two parts: model specifcation and execution algorithm. The former is a mathematical theory describing the kinds of structures and behavior that can be described with it. The latter specifies an algorithm that can correctly execute any model that is described in accordance with the model specification.

The execution algorithm corresponds to the aforementioned simulation algorithm. It defines the *operational semantics* of the model [Vangheluwe, de Lara, & Mosterman 2002]. Hence, a modeling formalism reflects the separation between model and simulator promoted by Zeigler et al. The same modeling formalism can have several implementations on computers based on the choice of a programming language or platform. For instance, a mathematical model can be designed and implemented using object-oriented model concepts and programming languages.

A "model" is not tied to a certain modeling formalism. Instead, the "same model" can be given or specified in different modeling formalisms. "The particular formalism and level of abstraction [of the model] depends on the background and goals of the modeller as much as on the system modelled" [Vangheluwe et al. 2002].

We conclude this section with some final words on the nature of systems, our modeling subjects. Some authors, such as Zeigler [1976] or Banks et al. [2000], indicate that a system we want to model is something that is real and exists (*real system*). However, this point of view seems to be rather restrictive. In addition to real systems that can be natural or artificial (man-made), also hypothetical/imaginary systems, i. e., systems that do not exist (yet), can be of interest for M&S, especially when using simulation to test the design of a future system (*simulation-based testing*).

3 Extensional and Intensional Definitions

> The Beginning of wisdom is the definition of terms.
>
> Socrates (470–339 B.C.)

When talking and reason about words, concepts, or terms, a common and clear understanding of their *meaning*[1] is essential. Such a common and clear understanding can be established by *definitions*, which can be understood and accepted by all parties. For that, definitions need to alleviate *vagueness* and *ambiguity* that can impair the conveyance of meaning (cf. Hurley [2006, pp. 72–6]). So in other words, the goal of a definition is to convey the meaning of a certain word, concept, or term to others comprehensibly, concisely, and unambiguously.

Sometimes, more than one definition for a certain word, concept, or term can exist, such as stated in Chapter 2. These definitions may assign more or less different meanings to the same term. In this case, it is important to agree on one definition, which then serves as the foundation for further elaboration.

In this chapter, we briefly define the term "definition" itself and outline two different approaches to create definitions (i. e., to define words, concepts, or terms):

1. *Extensional definitions* and
2. *Intensional definitions*.

Both, but particularly the latter, play an important role for the remainder of this thesis and the contribution described herein. We discuss this role at the end of this chapter, where we apply both approaches in a more formal, set-theoretical context.

[1] Hurley [2006, p. 72] distinguishes between *cognitive meaning* and *emotive meaning*. The former corresponds to terminology that conveys information, whereas the latter refers to terminology that expresses or evokes feelings. Herein we usually refer to cognitive meaning rather than emotive meaning.

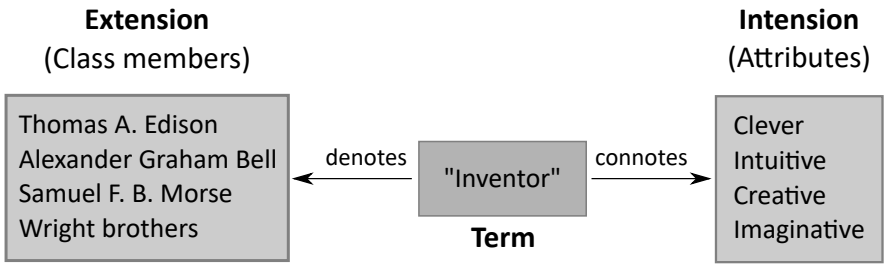

Figure 3.1: Excerpts of the extension and intension of the term "inventor." The figure is adapted from Hurley [2006, p. 83].

3.1 Terms and Definitions

In this thesis, we often use the word "term" to refer to a "thing" or concept that is of interest and about to be defined. Hurley [2006, p. 82] defines a term as follows:

Definition 3.1.1 (Term)
A **term** is any word or arrangement of words that may serve as the subject of a statement. Terms consist of proper names, common names, and descriptive phrases. [...] Words that are not terms include verbs, nonsubstantive adjectives, adverbs, prepositions, conjunctions, and all nonsyntactic arrangements of words.

According to Hurley [2006, p. 83], the (cognitive) meaning of a term can be divided into: *intensional meaning* and *extensional meaning*. Hurley describes both as follows:

> The intensional meaning consists of the qualities or attributes that the term connotes, and the extensional meaning consists of the members of the class that the term denotes.

This brings us to the *intension* and *extension* of a term, which are of interest for the remainder of this chapter. "The intensional meaning of a term is otherwise known as the intension, or connotation, and the extensional meaning is known as the extension, or denotation" [Hurley 2006, p. 83]. More precisely, Copi, Cohen, and McMahon [2014, p. 91] give the following two definitions:

Definition 3.1.2 (Extension)
The collection (or class or group) of all the objects to which a term may correctly be applied.

Definition 3.1.3 (Intension)
The attributes shared by all and only the objects in the class that a given term denotes; the connotation of the term.

Figure 3.1 elucidates the difference between the intension and extension of a term by giving an example. Both intension and extension are used when talking about *definitions*.

A definition can be a statement as referred to in Definition 3.1.1, in which a term is being defined, i. e., the subject of the definition. At the beginning of this chapter, we briefly and informally characterize the purpose of a definition, which is to convey the meaning of a certain word, concept, or term. Based on the characterization of Hurley [2006, p. 87], we derive the following "definition of a definition:"

Definition 3.1.4 (Definition)
A **definition** is a group of words that assigns a meaning to a term, i. e., some other word or a group of words.

Furthermore, Hurley writes that "every definition consists of two parts: the *definiendum* and the *definiens*," where "the definiendum is the word or group of words [i. e., the term] that is supposed to be defined, and the definiens is the word or group of words that does the defining." Like Hurley, we clarify the relationship between both, definiendum and definiens, by an example and Figure 3.2:

Example 3.1.1 (Definition)
In the following definition

> *A mitochondrion is a membrane-bound and energy-producing organelle of eukaryotic cells.*

the word "mitochondrion" is the definiendum and everything that follows the verb "is" is the definiens, i. e., "a membrane-bound and energy-producing organelle of eukaryotic cells." However, it also becomes apparent that readers of the above definitions need an understanding for the words that are part of the definiens, such as "eukaryotic." In other words, it is hard to convey the meaning of the word mitochondrion to someone who does not know membranes, cell organelles, or eukaryotes.

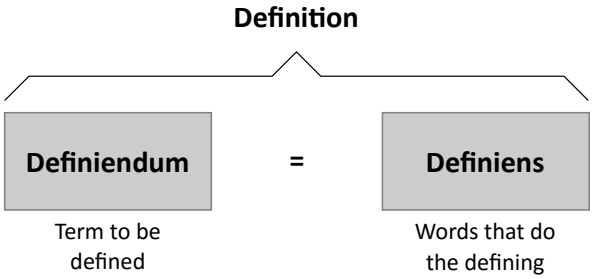

Figure 3.2: The composition of a definition comprising the definiendum and the definiens. The figure is adapted from Hurley [2006, p. 87].

One fundamental requirement for a proper definition is: a definition must not be *circular*, i. e., the definiendum itself must not appear in the definiens, otherwise "the definition can explain the meaning of a term being defined only to those who already understand it" Copi et al. [2014, p. 100].

Example 3.1.2 (Circular Definition)
The following definition is circular, according to Copi and Cohen:

> *A compulsive gambler is a person who gambles compulsively.*

To understand this definition, we already need to know the meaning of compulsively gambling, which, ultimately, is the term that is supposed to be defined.

It is also interesting to note that Hurley [2006, p. 87] concludes that "the definiens is not itself the meaning of the definiendum; rather, it is the group of words that symbolizes (or that is supposed to symbolize) the same meaning as the definiendum." This subtle difference between definiens and the actual meaning of the definiendum implies that the attempt to symbolize the meaning can be insufficient or incorrect.

In logic, different kinds of definitions and techniques to create definitions are distinguished. Here, the following two, which are based on the extension and intension of a term to be defined, are of particular interest:

1. Extensional definitions (or denotative definitions), and
2. Intensional definitions (or connotative definitions).

The focus is on the latter, as the title of this thesis implies. The next two sections characterize both extensional definitions and intensional definitions in more detail.

3.2 Extensional Definitions

As the name indicates, *extensional definitions* (or *denotative definitions*) "employ techniques that identify the extension of the term being defined" [Copi et al. 2014, p. 93]. Hurley [2006, p. 94] defines an extensional definition as follows:

Definition 3.2.1 (Extensional Definition)
An **extensional definition** is one that assigns a meaning to a term (i. e., the definiendum) by indicating the members of the class that the definiendum denotes [which is the extension of the term being defined].

Although extensional definitions are a very effective technique, they have a serious limitation: "it is usually impossible to enumerate all the objects in a class" [Copi et al. 2014, p. 93]. Based on how to identify a term's extension and indicating its members, we can distinguish between different kinds of extensional definitions. [Hurley 2006, p. 94] names three approaches to indicate the members of the class the definiendum denotes (i. e., the extension): (i) pointing to the members (in the literal sense), (ii) naming or listing the members individually, and (iii) naming the members in groups. This distinction results in three kinds of extensional definitions:

1. ostensive or demonstrative definitions,
2. enumerative definitions,
3. definitions by subclasses,

which we briefly discuss in the following. At the end of this section, we describe a further kind of extensional definitions, i. e., recursive definitions, which is not mentioned by Hurley [2006].

3.2.1 Ostensive definitions

The first kind of extensional definitions are *ostensive definitions* (also *demonstrative definitions*). Based on Copi et al. [2014, p. 94], we define an ostensive definition as follows:

Definition 3.2.2 (Ostensive Definition)
An **ostensive definition** is a kind of extensional definition in which the objects denoted by the term (i. e., the definiendum) being defined are referred to by means of pointing [the gesture], or with some other gesture.

In a nutshell, an ostensive definition is one that conveys the meaning of a term by pointing at examples. As such, ostensive definitions are "probably the most primitive form of definition [since] all one need know to understand such a definition is the meaning of pointing" [Hurley 2006, p. 94]. Moreover, Hurley states that "such definitions may be either partial or complete, depending on whether all or only some of the members of the class denoted by the definiendum are pointed to."

Example 3.2.1 (Ostensive Definition)
We can give an ostensive definition of a projector (an optical device for projecting images or videos onto a surface) to someone by pointing to projectors in a smart meeting room (as described in Section 1.2).

The above example indicates an obvious limitation of ostensive definitions, which is "that the required objects [need to] be available for being pointed at" [Hurley 2006, p. 94]. If there are no projectors around, we cannot point at them and thus give an ostensive definition of a projector. However, rather than pointing to physical objects that serve as examples for a certain term to be defined, we can also think about providing depictions of examples of this term to which we refer in our definition, in a corresponding media (e. g., a publication). So instead of pointing at projectors, we can provide pictures of different projectors, e. g., in a book. But even this adaption does not overcome another fundamental limitation of ostensive definitions, which is not considered by Hurley [2006] but becomes apparent in Aphorism 28 of Wittgenstein [1958].

> Now one can ostensively define a proper name, the name of a colour, the name of a material, a numeral, the name of a point of the compass and so on. The definition of the number two, "That is called 'two' "—pointing to two nuts–is perfectly exact.—But how can two be defined like that? The person one gives the definition to doesn't know what one wants to call "two"; he will suppose that "two" is the name given to *this* group of nuts!—He *may* suppose this; but perhaps he does not. He might make the opposite mistake; when I want to assign a name to this group of nuts, he might understand it as a numeral. And he might equally well take the name of a person, of which I give an ostensive definition, as that of a colour, of a race, or even of a point of the compass. That is to say: an ostensive definition can be variously interpreted in *every* case.

Simply put, there is an intrinsic ambiguity in ostensive definitions[2]. The person for whom we want to define a term ostensively needs a sufficient understanding of the information being given.

3.2.2 Enumerative Definitions

Another kind of extensional definitions are *enumerative definitions*. Based on Hurley [2006, p. 95], we define an enumerative definition as follows:

Definition 3.2.3 (Enumerative Definition)
An **enumerative definition** assigns a meaning to a term (i. e., the definiendum) by naming the members of the class the term denotes.

An enumerative definition can be either partial or complete (exhaustive), just like an ostensive definition. Furthermore, Hurley states that "complete enumerative definitions are usually more satisfying than partial ones because they identify the definiendum with greater assurance."

Example 3.2.2 (Enumerative Definition)
An enumerative definition of the decimal digits can be given as follows:

The symbols '0', '1', '2', '3', '4', '5', '6', '7', '8', and '9' are ***decimal digits****.*

[2] Copi et al. [2014, p. 94] mention *quasi-ostensive definitions* that resolve this ambiguity by using descriptive phrases in conjunction with gestures.

In contrast to Hurley [2006], others describe an enumerative as an exhaustive listing of all the members of the class the term denotes (cf. Wilson [1998, p. 13]), which refers to a complete enumerative definition in Hurley's terminology. It is obvious that "relatively few classes, however, can be completely enumerated" [Hurley 2006, p. 95], especially if the number of members of the class the term denotes is infinite. And even if the number of members is finite, the class still may have too many members making an enumerative definition impractical.

This fundamental limitation brings us to a third kind of extensional definitions: a definition by subclasses.

3.2.3 Definitions by Subclasses

Definitions by subclasses are quite similar to enumerative definitions. Hurley [2006, p. 95] defines a definition by subclass as follows:

Definition 3.2.4 (Definition by Subclass)
A **definition by subclass** is one that assigns a meaning to a term (i. e., the definiendum) by naming subclasses of the class denoted by the term.

Like the other kinds of extensional definitions, a definition by subclass can be either partial or complete, "depending on whether the subclasses named, when taken together, include all the members of the class or only some of them."

Example 3.2.3 (Definition by Subclass)
According to Hurley [2006, p. 95], the following definition of cetaceans:

> *cetacean means either a whale, a dolphin, or a porpoise,*

is a complete definition by subclass.

Still, definitions by subclasses suffer from the same limitation like enumerative definitions: "because relatively few terms denote classes that admit of a conveniently small number of subclasses, complete definitions by subclass are often difficult, if not impossible, to provide" [Hurley 2006, p. 95].

3.2.4 Recursive Definitions

Wilson [1998, pp. 13–4] distinguishes a further, rather unusual kind of extensional definitions "that is encountered in logic, mathematics, and other formal studies:" *recursive definitions* (or *inductive definitions*). Wilson gives the following definition of a recursive definition:

Definition 3.2.5 (Recursive Definition)
A **recursive definition** proceeds in three stages:

1. the base clause characterizes a subclass of the extension of the definiendum;
2. the induction or recursion clause gives a rule for determining the remaining objects in the extension of the definiendum by relating any such object to an object the definiendum already applies to;
3. the closure clause states that the definiendum applies to no other objects.

So a recursive definition defines the member of a class by means of other members in this class. The following example from Wilson [1998, p. 14] illustrates the three parts of a recursive definition.

Example 3.2.4 (Recursive Definition)
Ancestor can be defined recursively as follows:

1. *A person's parent are the person's ancestors;*
2. *A parent of a person's ancestor is a person's ancestor;*
3. *Nothing else is a person's ancestor.*

Recursive definitions play a special role, since they are quite different from the other kinds of extensional definitions[3] and do not share their most prominent limitation. In contrast to other extensional definitions, recursive definitions allow denoting the members of large or infinite extensions. However, recursive definitions, unlike other extensional definitions, do not indicate the members of the extension of the definiendum explicitly; they only provide the means to determine whether an object belongs to the extension. As such, recursive definitions share similarities with intensional definitions, which are described in the next section. This makes a delimitation between them difficult.

3.3 Intensional Definitions

Section 3.2 indicates that all kinds of extensional definitions, except of recursive definitions, suffer from serious limitations (cf. Hurley [2006, p. 96]). "Extensions can suggest intensions, but they cannot determine them." These limitations and deficiencies of extensional definitions bring us to *intensional* (or *connotative*) *definitions*, which play a crucial role in the remainder of this thesis. Hurley defines an intensional definition as follows

Definition 3.3.1 (Intensional Definition)
An **intensional definition** is one that assigns a meaning to a word (i. e., the definiendum) by indicating the qualities or attributes that the word connotes (i. e., its intension).

Furthermore, Hurley distinguishes between (at least) four kinds of intensional definitions:

1. Synonymous definitions,
2. Etymological definitions,
3. Operational definitions,
4. Definitions by genus and difference;

all of which differ in the way, how the intension is being indicated.

In the following, all four kinds of intensional definitions are briefly described, where the first two kinds of definitions (synonymous definitions and etymological definitions) are just mentioned for the sake of completeness.

3.3.1 Synonymous Definitions

Hurley [2006, p. 96] defines a *synonymous definition* as follows:

Definition 3.3.2 (Synonymous Definition)
A **synonymous definition** is one in which the definiens is a single word that connotes the same attributes as the definiendum.

[3] In fact and based on the given example one could argue that recursive definitions are intensional definitions rather than extensional definitions.

"In other words, the definiens is a synonym of the word being defined." If such a synonym exists, a synonymous definition is a "highly concise way of assigning a meaning" to a term. However, to convey the meaning of a term by giving a synonym of that term, the meaning of the actual synonym needs to be clear, i. e., the synonym must already been understood [Copi et al. 2014, p. 96].

Example 3.3.1 (Synonymous Definition)
[Hurley 2006, p. 96] gives the following example of a synonymous definition:

> *Physician means doctor.*

Hurley [2006, p. 97] writes that "many words, however, have subtle shades of meaning that are not connoted by any other single word." Hence, not all terms can be defined by giving suitable synonyms conveying the exact meaning of the definiendum. Moreover, Copi et al. [2014, p. 97] state that "synonyms are virtually useless [...] when the aim is to construct a precising or a theoretical definition."

3.3.2 Etymological Definitions

The second kind of intensional definitions are *etymological definitions*. Hurley [2006, p. 97] defines an etymological definition as follows:

Definition 3.3.3 (Etymological Definition)
An **etymological definition** assigns a meaning to a word [i. e., the definiendum] by disclosing the word's ancestry in both its own language and other languages.

According to Hurley, etymological definitions "have special importance for at least two reasons:"

1. "[...] the etymological definition of a word often conveys the word's root meaning or seminal meaning from which all other associated meanings are derived."
2. "[...] if one is familiar with the etymology of one English word, one often has access to the meaning of an entire constellation of related words."

Example 3.3.2 (Etymological Definition)
"The word 'principle' derives from the Latin word 'principium', which means beginning or source. Accordingly, the 'principles of physics' are those fundamental laws that provide the 'source' of the science of physics" [Hurley 2006, p. 97].

However, the above example makes apparent that it is hard to fit etymological definitions into the existing taxonomy of extensional and intensional definitions. In fact, other authors, such as Wilson [1998] or Copi et al. [2014], do not list etymological definitions as a kind of intensional definition.

3.3.3 Operational Definitions

A more relevant kind of intensional definitions are *operational definitions*. Hurley [2006, p. 97] defines:

Definition 3.3.4 (Operational Definition)
An **operational definition** assigns a meaning to a word (i. e., the definiendum) by specifying certain experimental procedures that determine whether or not the word applies to a certain thing.

Hurley gives the following example to illuminate operational definitions:

Example 3.3.3 (Operation Definition)
A subject has "brain activity" if and only if an electroencephalograph shows oscillations when attached to the subject's head.

The above example "prescribes an operation to be performed," i. e., electroencephalography. Without "such an operation, a definition cannot be an operational definitions" [Hurley 2006, p. 98]. Furthermore, Hurley writes: "operational definitions were invented for the purpose of tying down relatively abstract concepts to the solid ground of empirical reality." One limitation of operational definitions is that they "usually convey only part of the intensional meaning of a term." For instance, "'brain activity' means more than oscillations on an electroencephalograph."

3.3.4 Definitions by Genus and Difference

Definitions by genus and difference are an important type of intensional definitions, which are going back to the ancient Greek philosopher Aristotle (384–322 B.C.E.). As the name indicates, such definitions consist of two aspects: (i) *genus* and (ii) *difference*.

The genus (also kind or category) describes a general class or group to which an object belongs, whereas the (specific) difference answers the question how an object differs from other members of the object's genus. In other words, the difference "is the attribute or attributes that distinguish the various [members] within a genus" [Hurley 2006, p. 98]. More formally, Hurley [2006, p. 98] defines a definition by genus and difference as follows:

Definition 3.3.5 (Definition by Genus and Difference)
A **definition by genus and difference** assigns a meaning to a term by identifying a genus term and one or more difference words that, when combined, convey the meaning of the term being defined. It [a definition by genus and difference] consists of combining a term denoting a genus with a word or group of words connoting a specific difference.

"Definitions by genus and difference are also called analytical definitions" [Copi et al. 2014, p. 98]. The following example illuminates the different aspect of a definition by genus and difference.

Example 3.3.4 (Definition by Genus and Difference)
The following definition of a mitochondrion:

> *A mitochondrion is a membrane-bound, eukaryotic cell organelle that produces energy (in the form of adenosine triphosphate or ATP), which is required for the cellular respiration and other processes within the cells,*

is a definition by genus and difference. Cell organelle is the genus of the mitochondrion, whereas the difference is the combination of the following attributes:

1. *membrane-bound: which distinguishes mitochondria from cell organelles such as chromosomes and ribosomes, which have no membrane.*
2. *eukaryotic: which distinguishes mitochondria from cell organelles that only exist in prokaryotic cells, such as mesosomes, or that can exist in prokaryotic and eukaryotic cells, such as ribosomes.*
3. *energy producing: which distinguishes mitochondria from other membrane-bound, eukaryotic cell organelles such as the reticulum or nuclei, which are not producing energy.*

According to Hurley, a definition by genus and difference is "more generally applicable and achieves more adequate results than any of the other kinds of intensional definition." However, it may be challenging to provide a specific difference so that only objects of the class the definiendum denotes are covered by the definition.

3.4 Summary and Discussion

To conclude, a definition is supposed to convey the meaning of a term, unambiguously and concisely. Each definition consists of two parts:

- The definiendum that is the term[4] that is to be defined.
- The definiens that is the word or group of words that does the defining.

The definiens can include other terms that may, in turn, require further definitions to grasp the original definition. Depending on what the definiens looks like, we distinguish between two main techniques of defining a term:

1. Extensional definitions,
2. Intensional definitions.

An extensional definition indicates the members of the extension of the definiendum, whereas an intensional definition describes the attributes shared by all and only the objects in the class the definiendum denotes [Copi et al. 2014, p. 91], which is the intension of the definiendum. Both definitional techniques can be further divided into subtypes of extensional and intensional definitions, as shown in Table 3.1.

Table 3.1: Techniques for defining terms, i. e., creating definitions.

Extensional Definition	Intensional Definition
Ostensive definition	Synonymous definition
Enumerative definition	Etymological definition
Definition by subclass	Operational definition
Recursive definition	Definition by genus and difference

As Section 3.2 and Section 3.3 state, each of these subtypes (kinds) of definitional techniques has limitations; some techniques have more than others. The main shortcoming of extensional definitions, apart from recursive definitions, is their impracticality if the extension of the term to be defined is too big. However, if the number of members of the term's extension is quite small, extensional definitions are convenient, concise, and intuitive. It becomes immediately apparent, which objects belong to the class a term denotes. Although more practical for terms with bigger or infinite extensions, intensional definitions, on the other hand, sometimes require an understanding of the terms that are used to describe the intension of the definiendum. Moreover, we first need to verify whether an object possess the attributes used in an intensional definition before we can decide whether or not this object belongs to the class the definiendum denotes. Recalling the example given in Figure 3.1, we first need to determine whether a certain person is clever, intuitive, creative, and imaginative before we can tell that this person is an inventor. This can prove to be rather difficult. Intensional definitions can also be subject to vagueness and ambiguity, if not carefully specified. A more elaborate comparison of the different kinds of definitional techniques is given by Hurley [2006, pp. 94–100].

[4] We often use the words "term" and "definiendum" interchangeably.

In modeling and simulation, definitions play a crucial role, particularly when specifying the behavior and structure of models, such as the composition or coupling scheme. In this thesis, the focus is on set-theoretical, algebraic model definitions, i. e., models that are defined by means of tuples, sets, and relations on these sets (see Section 4.2). A typical definition of a set often looks like:

$$A = \{a, b, c\},$$

where A is the definiendum (the set to be defined) and everything right of the equal sign is the definiens. Adapting the terminology introduced in this chapter to the definition of sets, the extension of a set corresponds to all the members of this particular set, whereas the intension of the set refers to attributes that all and only the members of this set share. These attributes are often specified by using logical predicates, where a logical predicate is a Boolean-valued function $P\colon X \to \{\top, \bot\}$.

Example 3.4.1 (Definition of Sets)
Assume we want to define the set of single-digit, natural numbers (including the zero), denoted by $\mathbb{N}^{<10}$. An extensional definition of this set would look as follows:

$$\mathbb{N}^{<10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

An intensional definition of the same set can look as follows:

$$\mathbb{N}^{<10} = \{x \in \mathbb{N} \mid x < 10\},$$

where $\mathbb{N}$ is the set of natural numbers. Interestingly, the intensional definition makes use of the set of natural numbers, which is only defined verbally. However, we can define the set of natural numbers $\mathbb{N}$ recursively as follows:

1. $0 \in \mathbb{N}$
2. $\forall n \in \mathbb{N}:\ n + 1 \in \mathbb{N}$
3. *Nothing else is in* $\mathbb{N}$.

Again, the meaning of $n + 1$ needs to be clear to understand the above definition.

For such mathematical definitions, ostensive definitions, synonymous definitions, etymological definitions, and operational definitions are not suited (cf. Hurley [2006, p. 100]), so we focus on the remaining kinds of definitional techniques, particularly:

- Extensional definitions
  - Enumerative definitions
  - Recursive definitions
- Intensional definitions
  - Definition by genus and difference

In Example 3.4.1, the extensional definition

$$\mathbb{N}^{<10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

is an exhaustive enumerative definition. All members of the extension are explicitly listed (i. e., enumerated) and can be immediately perceived just by looking at the definition. The intensional definition

$$\mathbb{N}^{<10} = \{x \in \mathbb{N} \mid x < 10\},$$

is a definition by genus and difference. The set $\mathbb{N}$ is the genus, whereas $x < 10$ is the difference that distinguishes members that belong to the set to be defined (single-digit, natural numbers) from other natural numbers with more than one digit. Such an intensional definition is also referred to as *set-builder notation*.

4 Discrete Event Simulation

> Discrete-event simulation is alive and kicking!
>
> Sally Brailsford

This thesis mainly focuses on *discrete event simulation* rather than *continuous* or *hybrid simulation*. Discrete event modeling provides an appropriate and sufficient abstraction for the systems we have been investigating during my doctoral studies, ranging from smart environments [Krüger et al. 2012; Nyolt et al. 2013, 2015; Steiniger et al. 2012] over mitochondrial networks in eukaryotic cells [Steiniger & Uhrmacher 2013, 2016] to populations and societies [Steiniger et al. 2014; Warnke et al. 2015].

Furthermore, in this thesis, we introduce[1] a modeling formalism that is based on a parallel variant of the *Discrete Event System Specification* (DEVS) formalism for discrete event simulation: Parallel DEVS (P-DEVS), which is rooted in *systems theory*. As discussed by Zeigler et al. [2000, pp. 391–409] and indicated by Vangheluwe [2000], DEVS is a universal formalism for modeling *discrete event systems*, meaning that all possible discrete event systems can be expressed in DEVS. Moreover, the concept of *state quantization* allows approximating *continuous systems* by discrete event systems [Kofman & Junco 2001][2].

This chapter starts with a brief overview of discrete event systems and discrete event simulation before it details DEVS and its variants, particularly P-DEVS. This formalism serves as a foundation for Chapter 9 and introduces the idea of defining a modeling formalism based on *structured sets*, i. e., at the level of *structured systems*.

Parts of this chapter are based on:

Steiniger, A., Krüger, F., and Uhrmacher, A. M. (2012). "Modeling Agents and their Environment in Multi-Level-DEVS." In *Proceedings of the 2012 Winter Simulation Conference* (WSC'12). Article No. 233.

Steiniger, A. and Uhrmacher, A. M. (2016). "Intensional Couplings in Variable Structure Models: An Exploration Based on Multilevel-DEVS." In *ACM Transactions on Modeling and Computer Simulation* (TOMACS), 26(2). pp. 9-1–9-27.

[1] see Chapter 9

[2] The "quantization of the state variables [is] a method to obtain a discrete event approximation of a continuous system [. . .]," which is done by "using a piecewise constant function" [Kofman & Junco 2001]. Zeigler et al. [2000, pp. 419–21] provide details on quantization.

4.1 Basics

Abstraction is an important means of modeling and simulating systems of interest, either real or imagined ones (cf. Chapter 2). Often it is sufficient to abstract a system in the way that it changes its state or state variables only at certain instants of time, i. e., when certain *events* occur. "An event is an abstraction used in the simulation to model some instantaneous action in the physical [or imagined] system" [Fujimoto 2000, p. 32]. This is the underlying idea of *discrete event simulation*.

More formally, Banks et al. [2000, p. 14] describe discrete event simulation as "the modeling of systems in which state variables [or states] change only at a discrete set of points in time." In addition, in a finite time interval, only a finite (countable) number of events (such as state changes) can occur Zeigler [1976, p. 22]. We call models of such systems *discrete event models*[3]. Simply put, discrete event simulation deals with the execution of discrete event models.

Although events happen at discrete points in time, the time base of discrete event models is continuous (real numbers), as discussed by Cellier [1991, p. 14][4]. The state (or state variables) of discrete event models can be arbitrary, i. e., of discrete or continuous nature (cf. Figure 2.1). Regardless of the nature of the state, the state is changing instantaneously. Thus, the state trajectory of a discrete event model consists of *piecewise constant segments* [Zeigler et al. 2000, p. 103], such as depicted in Figure 4.1.

When executing discrete event models, their event-drivenness has an impact on the flow of simulation time, i. e., the abstraction of the physical time advancing in the system that is being modeled [Fujimoto 2000, p. 27]. "In an event-driven simulation, simulation time does not advance from one time step to the next but, rather, advanced from the time stamp of one event to the next" [Fujimoto 2000, p. 33]. Between two consecutive events, no wall-clock[5] time elapses, however the simulation time changes if the events have different time stamps. When processing events and updating the model state consequently, wall-clock time elapses, whereas the simulation time remains unchanged.

According to Fujimoto [2000, p. 34], a sequential discrete event simulation typically makes use of the following three data structures:

1. *State variables* that describe the current state of the modeled system.
2. An *event list* (or event queue) that contains the events that can occur some time in the future. The list is constantly being updated during a simulation, reflecting changes.
3. A *global clock* that indicates the current simulation time.

The processing of discrete event models as described above, i. e., "jumping" from one event to another during simulation based on an event list, is also known as *event scheduling*. According to Zeigler et al. [2000, p. 159], there are two further approaches (also called word views) to discrete event simulation: *activity scanning* and *process interaction*, with "the last being a combination of the first two." Banks [1998, p. 9] describes activity scanning as follows:

> Activity scanning is similar to rule-based programming. (If a specific condition is met, a rule is *fired*, meaning that an action is taken.) [...] Scanning takes place at fixed time increments at which a determination is made concerning whether or not an event occur at that time [simulation time]. If an event occurs, the system state is updated.

[3] Note that depending on a certain question at hand, it may be appropriate to approximate a continuous system (a system that changes it state continuously) by a discrete event system.

[4] In contrast to Zeigler [1976, p. 22], Cellier [1991, p. 14] writes that the time axis of discrete event models does not necessarily have to be continuous but usually is. Here, we assume that the time base of a discrete event model is continuous.

[5] Wall-clock time corresponds to the time that passes during the execution of a simulation [Fujimoto 2000, p. 27], as perceived by the user running the simulation (execution time).

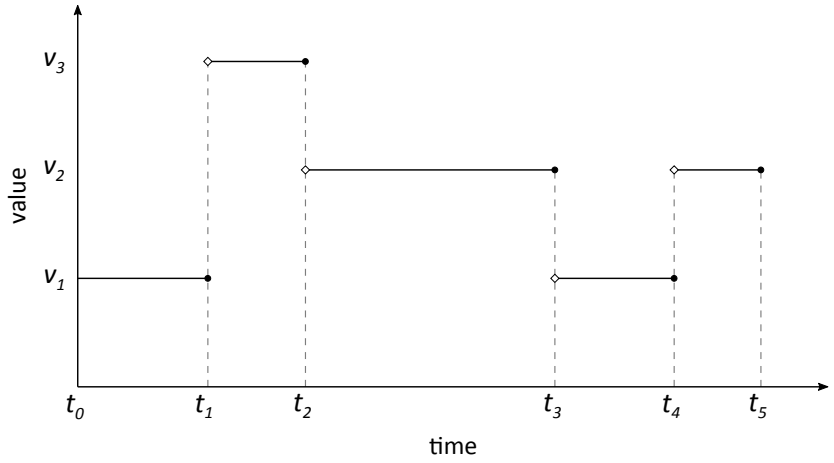

Figure 4.1: An exemplary trajectory consisting of piecewise constant segments.

In the process interaction world view, each entity of the system is represented by a process and their life cycle. These processes can interact with each other during simulation. In addition to the three classic world views, i. e., events, activities, and processes, there is a fourth approach for discrete event simulation: the *three phase approach*, which was first proposed by Tocher [1963]. In the three phase approach, two different types of events or system activities are distinguished (i) time-dependent *bound events* (*B events*) and (ii) state-dependent *conditional events* (*C events*) [Roberts & Pegden 2017]. The occurrence of a bound event is predictable and thus can be scheduled, whereas the occurrence of a conditional event depends on the fulfillment of certain conditions (e. g., the availability of certain resources) [Lin & Lee 1993, p. 383]. As its name indicates, the execution of a three phase simulation consists of three phases:

1. the *A phase* (*time scan*),
2. the *B phase* (*B calls*), and
3. the *C phase* (*C calls*).

In the first phase, the simulation clock advances to the time of the next scheduled bound event. Then, in the second phase, all bound events that are scheduled for the current time are executed. Finally, all conditional events are evaluated and those whose conditions are satisfied are also executed. We repeat these three phases constantly until a certain simulation end criterion is fulfilled, e. g., there are no further bound events. The three phase approach is closely related to activity scanning [Roberts & Pegden 2017]. In this thesis, we pursue the event scheduling world view.

4.2 Discrete Event System Specification and its Variants

The *Discrete Event System Specification* (DEVS) is an established and well-studied modular, hierarchical system specification formalism, i. e., modeling formalism, for discrete event simulation. As such, "DEVS allows to represent all systems, whose input/output behavior can be described by a sequence of events under the condition that the state [of the system] undergoes a finite number of changes within any finite interval of time [such a system is called discrete event system]" [Cellier & Kofman 2006, p. 524]. Bernard P. Zeigler introduced the basic DEVS formalism already in the mid-seventies [Zeigler 1976]. Later, inspired by Wymore [1967, pp. 194–292], Zeigler [1984] extended the basic formalism by a concept for a

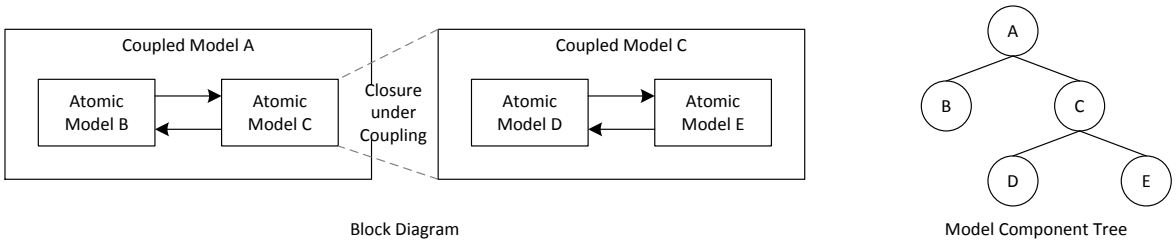

Figure 4.2: A block diagram (left-hand side) of a simple hierarchical model in which we make use of closure under coupling leading to the model component tree depicted on the right-hand side.

modular *coupling* of basic DEVS models[6] and outlined the *closure under coupling*[7] of the formalism. A formalism is called *closed under coupling*, if a coupled system or a network of systems specified in a certain formalism can also be specified as a basic (or atomic) system in the same formalism [Zeigler et al. 2000, p. 149]. Thus, DEVS allows a hierarchical model construction, where coupled models can become components of other coupled models (see Figure 4.2).

DEVS is rooted in *system theory* [Castro, Kofman, & Wainer 2008] and follows the *reactive systems metaphor*, i.e., DEVS views the modeled system as a *reactive system* that (i) changes its state over time as a result of certain inputs (external stimuli) or the flow of time itself and (ii) can create outputs. Vangheluwe [2001] describes the formalism as an extension of *finite automata* (also *finite state machines*)—with a Moore-like output[8]. In contrast to finite state machines (FSMs), DEVS allows, in principle, an infinite number of states and in- and outputs, and thus is not limited to describing systems with a finite number of states [Cellier & Kofman 2006, p. 524]. However, Hwang and Zeigler [2009] define a subclass of DEVS, called Finite and Deterministic DEVS (FD-DEVS), in which infinite state behavior is abstracted by a finite reachability graph[9]. Furthermore, classic FSMs have no notion of time and describe discrete-time systems[10]. Whereas DEVS describes continuous-time systems and has an explicit notion of elapsed (simulation) time.

As a modeling formalism in the tradition of Sarjoughian [2006], DEVS emphasizes a strict separation between model definition (syntax) and execution semantics. The latter of which is specified by an *abstract simulator* and determines how simulation trajectories are computed based upon model definitions. Similar to automata theory, in DEVS and its variants, models are defined set-theoretically or algebraically (cf. Cellier and Kofman [2006, p. 525]), i. e., by defining characteristic sets (inputs, outputs, and states) and functions (or relations) on this sets that determine the behavior of the modeled system in accordance with the execution semantics.

Since its introduction, DEVS serves as basis for a plethora of variants and extensions. These often focus on addressing and capturing certain characteristics or specifics of the systems to be modeled (i. e., to be specified in the respective formalism). Furthermore, there exist also variants and extensions of DEVS that were designed for special purposes, such as the

[6] Zeigler already presented first ideas about networks of system specifications such as DEVS in Zeigler [1976, pp. 241–7]. However, the concept of connecting models therein is non-modular and the term "couplings" was first used by Zeigler in his 1984 book.

[7] Zeigler et al. [2000, pp. 151–2] give a detailed construction of the closure under coupling of DEVS.

[8] The output of a *Moore machine*, a special finite automaton named after Edward F. Moore, depends only on its state but not on its input [Moore 1956]. However, as the state transition function of a Moore machine is defined on its inputs, the inputs can be implicitly encoded in the states.

[9] In more detail, the behavior of FD-DEVS networks can be abstracted by an isomorphic finite-vertex reachability graph that makes no restrictions on the occurrence of external events.

[10] There exist extensions of FSMs that allow describing discrete event systems with a continuous timescale, such as described by Alur and Dill [1994] or Cassandras and Lafortune [2008].

integration of external processes in the simulation (*in-the-loop simulation*) or a model-driven synthesis of executable programs. Uhrmacher et al. [2010, p. 140] identify three types of variants that prevail: (i) variants that introduce continuous aspects to extend the discrete nature[11] of DEVS, (ii) variants that focus on real-time applications and the integration of external processes, and (iii) variants that overcome the static (model) structure of DEVS. Another type of DEVS variants, that should be added to the above classification, are those that formally introduce nondeterminism to DEVS[12]. Table 4.1 gives an overview of some more or less prominent variants and extensions of DEVS and their particularities (the list is not exhaustive). Another brief overview of the vast universe of DEVS variants can be found in Van Tendeloo and Vangheluwe [2017].

4.2.1 Parallel DEVS

A quite prominent and important variant of DEVS is the *Parallel Discrete Event System Specification* (P-DEVS). Since P-DEVS serves as foundation for the formalism presented later in the thesis (see Chapter 9), we focus on P-DEVS in the following. Chow and Zeigler [1994] introduce P-DEVS as a parallel variant of DEVS, to ease the modeling of concurrent systems[13]. In fact, classic DEVS[14] and P-DEVS specify the same class of systems [Zeigler et al. 2000, pp. 392–3]. Similar to DEVS, P-DEVS distinguishes between atomic models (basic models) and coupled models (networks). Atomic models describe the "behaviour of a discrete-event system as a sequence of deterministic transitions between sequential states as well as how it [the described system] reacts to external input (events) and how it [the system] generates output (events)" Vangheluwe [2001]. Coupled models, on the contrary, describe the modeled system as a network of components that can influence each other according to a specified coupling scheme, by exchanging events. In other words, coupled models define structure [Van Tendeloo & Vangheluwe 2018]. The components of a network can be atomic models or, due to the formalisms closure under coupling[15], coupled models. Following Chow and Zeigler [1994], we define an atomic P-DEVS model as follows:

Definition 4.2.1 (Atomic P-DEVS Model)
An **atomic P-DEVS model** (or basic P-DEVS model) is defined as the structure:

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$$

where

- X is an arbitrary set of *inputs* (external events);
- Y is an arbitrary set of *outputs* (output events);
- S is an arbitrary set of *sequential states*;
- $\delta_{int}\colon\ S \to S$ is the *internal (state) transition function*;

[11] Here, "discrete" does not refer to the timescale, but to the fact that in a given time interval only a finite number of state changes can happen at discrete points on a continuous timescale.

[12] The classic DEVS formalism as well as many of its variants specify deterministic systems. However, Zeigler [1976, pp. 131–4] describes, how probabilistic models (i. e., nondeterministic models) can be represented by an equivalent deterministic model under certain assumptions and by exploiting *pseudo-random number generators*.

[13] In P-DEVS, parallelism refers to modeling and execution aspects rather than a property of the modeled system. Already in the original DEVS formalism components (of a coupled model) are assumed to be independent and concurrent to the rest [Syriani & Vangheluwe 2010]. Moreover, P-DEVS models can also be executed sequentially [Himmelspach & Uhrmacher 2006].

[14] Zeigler et al. [2000] provide a comprehensive definition of the "classic" DEVS formalism.

[15] Zeigler et al. [2000, pp. 152–3] proof the closure under coupling of P-DEVS.

Table 4.1: List of some variants and extensions of DEVS.

Formalism (Variant)	**Feature(s) and Notes**	**Reference(s)**
Extended DEVS (E-DEVS)	Parallelism, parallel execution	Wang [1992]; Wang and Zeigler [1993]
Parallel DEVS (P-DEVS)	Parallelism, parallel exeuction	Chow [1996]; Chow and Zeigler [1994]; Chow et al. [1994]
Fuzzy-DEVS	Nondeterminism, uncertainty	Kwon, Park, Jung, and Kim [1996]
Stochastic DEVS (STDEVS)	Nondeterminism, uncertainty	Castro et al. [2008]; Castro, Kofman, and Wainer [2010]
iDEVS	Nondeterminism, uncertainty	Bisgambiglia, de Gentili, and Santucci [2009]
Variable DEVS (V-DEVS)	Variable structures	Barros, Mendes, and Zeigler [1994]
Dynamic Structure DEVS (DSDEVS)	Variable structures	Barros [1995a, 1995b, 1996]
DSDE	Parallel variant of DSDEVS	Barros [1997, 1998]
Dynamic DEVS (dynDEVS)	Variable structures (reflection)	Uhrmacher [2001]
Parallel Dynamic DEVS (PdynDEVS)[*]	Parallel variant of dynDEVS	Himmelspach and Röhl [2009]; Uhrmacher et al. [2010]
ρ-DEVS	Variable structures, parallelism, variable ports, intensional couplings	Uhrmacher et al. [2006]
Self-reproducible DEVS (SR DEVS)	Variable structures, instantiation	"Self-reproducible DEVS formalism" [2005]
Multi-Level DEVS (ML-DEVS)	Multi-level modeling, variable structures, variable ports, intensional couplings	Uhrmacher et al. [2007]
Multi-Resolution DEVS (MR-DEVS)	Multi-level modeling, variable structures	Gao, Li, Wang, and Chen [2012]; Li, Li, Hu, and Chai [2011]
Real-Time DEVS (RT-DEVS)	Real-time execution of models	Hong, Song, Kim, and Park [1997]
Parallel External Process Interface DEVS (PepiDEVS)	Integration of external processes, parallelism	Himmelspach [2007]
Parallel Dynamic External Process Interface DEVS (PdynEpiDEVS)	Combination of PdynDEVS and PepiDEVS	Himmelspach [2007]
Cell-DEVS	Spatial modeling	Wainer [1998, 1999]; Wainer and Giambiasi [1998]
Parameterized DEVS	Output-to-parameter integration of different perspectives of healthcare systems	Djitog, Aliyu, and Traoré [2017]
Vectorial DEVS (VECDEVS)	large-scale modeling, visual modeling	Bergero and Kofman [2014]
Generalized DEVS (GDEVS)	Hybrid systems modeling	Giambiasi and Carmona [2006]; Giambiasi, Escude, and Ghosh [2001]
Discrete Event and Differential Equation Specified Systems (DEV&DESS)	Hybrid systems modeling	Praehofer [1992]; Zeigler et al. [2000]
ΦDEVS	Equation-based modeling, constraint-based modeling	Honig and Seck [2012]
Synthesizable DEVS (SynDEVS)	Model synthesis, model transformation, VHDL	Molter [2012]; Molter, Seffrin, and Huss [2009]
Conceptual Modeling Language for DEVS (CML-DEVS)	Conceptual modeling, model transformation, formalism interoperability	Hollmann, Cristiá, and Frydman [2015]
ZDEVS	Integration of formal methods, Z specification language	Traore [2006]

[a] also called DynPDEVS in Uhrmacher et al. [2010]

- $\delta_{ext}\colon\ Q \times X^b \to S$ is the *external (state) transition function*, where
  - $Q = \{(s, e) \mid s \in S, 0 \leqslant e < ta(s)\}$ is the set of *total states*[a],
  - X^b is a set of bags over the elements in X;
- $\delta_{con}\colon\ S \times X \to S$ is the *confluent (state) transition function*, where X^b is defined as above;
- $\lambda\colon\ S \to Y^b$ is the *output function*, where
  - Y^b is a set of bags over the elements in Y;
- $ta\colon\ S_p \to \mathbb{R}_0^+ \cup \{\infty\}$ is the *time advance function*.

[a] Note that Chow and Zeigler [1994] as well as Zeigler et al. [2000, p. 143] write $0 < e < ta(s)$ instead of $0 \leqslant e < ta(s)$. This, however, seems to be a typographical error, as inputs can be received immediately after (transitory) state transitions.

So an atomic P-DEVS model is defined by the sets X, Y, and S and the functions δ_{ext}, δ_{int}, δ_{int}, λ, and ta. All elements of S, i. e., sequential states, together constitute the *state space*[16] of the atomic model M. The sequentiality of the states implies that the next state of the model depends solely on its current state and potential inputs (cf. Khoussainov and Nerode [2001, p. 47]). Furthermore, we often assume that a certain oder is defined on the set S, i. e., that S is defined as follows:

$$S = \{s_0, s_1, \ldots, s_n\},$$

where s_0 is the *initial state* (or *start state*) of the model M. We need to know the initial state for the actual execution of a model. For this reason, the above definition of atomic P-DEVS (or atomic models of other DEVS variants) is often generalized such that the initial state becomes a part of the defining tuple of the model itself and the set S is defined as an ordinary set, such as in Barros [1997]. Accordingly, we would define an atomic P-DEVS model M also as follows:

$$M = \langle X, Y, S, s_{init}, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$$

with $s_{init} \in S$ being the initial state and X, Y, S, δ_{int}, δ_{ext}, δ_{con}, λ, and ta as in Definition 4.2.1. Going one step further, Van Tendeloo and Vangheluwe [2018] argue that an initial state alone is not enough for a proper model initialization, in addition we also need an initial elapsed time e_{init} leading to an initial total state q_{init} with:

$$q_{init} = (s_{init}, e_{init})\,.$$

The functions δ_{int}, δ_{ext}, δ_{con}, λ, and ta define the actual behavior (dynamics) of the atomic model M. The first three functions refer to the three different state transition functions, which determine how the state of the model evolves during simulation. The time advance function ta assigns a time[17] (*lifespan*) to each state in S. The model remains in its current state $s \in S$ for the time (interval) determined by ta, i. e., $ta(s)$, if the model does not receive an input bag[18] meanwhile (*external event*). When this time has passed and the model has

[16] Russel and Norvig [2010, p. 67] define the state space as "the set of all states reachable from the initial state by any sequence of actions [state transitions]." Herein, we relax the reachability criterion and define the state space as a set of states that at least includes the set of all states a state-based model may reach in principle, where the actual reachability is of no further interest.

[17] Since the timebase of DEVS is continuous, the time that is associated with each state by the ta-function is a real number or ∞. Please note that in Definition 4.2.1 we define $\mathbb{R}_0^+$ as the set of positive real numbers including 0, i. e., $\mathbb{R}_0^+ = \{x \in \mathbb{R} \mid x \geqslant 0\}$ with $\mathbb{R}$ being the set of all real numbers.

[18] As in P-DEVS, in contrast to DEVS, several components of a coupled model can perform internal state transitions in parallel, influenced components can receive more than one input at the same time. Moreover,

not received an input bag in the meantime (*internal event*), an *internal state transition* takes place and the internal transition function δ_{int} determines the new state $s' \in S$ of the atomic model based on its current state s, i. e., $s' = \delta_{int}(s)$. If an input bag $x^b \in X^b$ has been received before the lifespan of the current state s has expired, an *external state transition* takes place and the external transition function δ_{ext} determines the new state s' based on the current state s; the time elapsed since the last state transition, denoted by e (with $0 \leqslant e < ta(s)$); and the input bag x^b; i. e., $s' = \delta_{ext}((s, e), x)$. If both internal event and external event collide, so if $e = ta(s)$, a *confluent state transition* takes place and the confluent transition function δ_{con} is invoked determining the new state of the atomic model. This function gives modelers the control to resolve such collisions and leaves the decision how, to the atomic models[19] [Zeigler et al. 2000, pp. 143–4]. Zeigler et al. define the confluent transition function by default as follows:

$$\delta_{con}(s, x^b) = \delta_{ext}((\delta_{int}(s), 0), x^b).$$

Immediately before an internal or confluent state transition takes place and the current state of the model is changed, the output function λ is invoked and creates an output bag [20] based on the current, yet unchanged state. This particularity of the operational semantics of DEVS variants (i. e., outputs are created before the state is updates) makes it often necessary to define additional, transitory state transitions whenever a model shall immediately create an output as a result of an input.

Coupled P-DEVS models, on the other hand, allow us to couple atomic P-DEVS models[21] and thus to describe a "system as a network of coupled components" [Vangheluwe 2001]. Hence, a coupled model is sometimes called *network*, in the literature. Based on Chow and Zeigler [1994], we formally define a coupled P-DEVS model as follows:

Definition 4.2.2 (Coupled P-DEVS Model)
A **coupled P-DEVS model** (or P-DEVS network) is defined as the structure:

$$N = \langle X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\} \rangle$$

where

- X is an arbitrary set of *inputs*,
- Y is an arbitrary set of *outputs*,
- D is a set of *component references*;

where for each $d \in D$,

a P-DEVS component can receive the same input from different components. Thus, atomic P-DEVS models receive input bags rather than single inputs. Still, a bag may contain only one element or no element at all (*empty bag*). Zeigler et al. [2000, p. 90] define a bag informally as "a set with possible multiple occurrences of its elements." A formal definition of sets of bags (*bag sets*) can be found in Section A.1.4.

[19] In classic DEVS, collisions are resolved by a *serialization* of the model behavior via the tie-breaking function *Select* at the level of coupled models (global decision) [Zeigler et al. 2000, p. 143].

[20] In contrast to the usage of input bags, the usage of output bags is not clearly motivated in the literature. Moreover, output bags are not consistently used in the formal theory on P-DEVS. For instance, Chow and Zeigler [1994] assume, in their proof of the closure under coupling of P-DEVS, that the λ-function returns single outputs rather than bags, although λ is formally defined in such a way that it returns bags. We adhere to the original definition of P-DEVS here and use output bags.

[21] Due to its closure under coupling, coupled models can be represented by atomic models in P-DEVS. Thus, components of a coupled P-DEVS model can in turn be coupled P-DEVS models.

– M_d is an atomic P-DEVS model (*component*) with

$$M_d = \langle X_d, Y_d, S_d, \delta_{int\,d}, \delta_{ext\,d}, \delta_{con\,d}, \lambda_d, ta_d \rangle,$$

– I_d is a set of *influencers* of component d with $I_d \subseteq ((D \cup \{N\}) \backslash \{d\})$ and N being the reference of the network itself[a];

and for each $i \in I_d$ with $d \in (D \cup \{N\})$,

– $Z_{i,d}$ is a function, the *i-to-d output translation*, with

$$Z_{i,d} = \begin{cases} X \to X_d & \text{if } i = N \text{ (\textit{external input coupling})} \\ Y_i \to Y & \text{if } d = N \text{ (\textit{external output coupling})} \\ Y_i \to X_d & \text{otherwise (\textit{internal coupling})} \end{cases}.$$

The sets X_i and Y_i refer to X and Y of the component whose reference is i, respectively. In the following, we call a component reference also *identifier* or, simply, name.

[a] Here we assume that N is the definition of the coupled model as well as the reference to the coupled model.

Like in atomic P-DEVS models, X and Y denote the set of inputs and the set of outputs of the coupled P-DEVS model, respectively. Each component (submodel) of the coupled model has a unique reference $d \in D$ by which the component can be identified and referenced. The set $\{M_d\}$ contains the actual definitions of the components, where each definition is associated with a component reference. By definition, a component is an atomic P-DEVS model as in Definition 4.2.1. The communication between components is explicitly defined by a *coupling scheme* that comprises two sets: a set of influencer sets $\{I_d\}$ and a set of translation functions $\{Z_{i,d}\}$ [Zeigler et al. 2000, p. 128]. For each component with the reference $d \in D$, the set I_d contains the references of all the other components (including the coupled model and excluding component d) that can influence[22] component d, i. e., that can send events[23] to d. If I_d is empty, component d is not influenced by the coupled model or its components. Finally, for each pair of influenced component d (*influencee*) and influencing component i (*influencer*) with $i \in I_d$, the function $Z_{i,d}$ translates events coming from component i (output events) into events of component d (input events). Not only the components of a coupled model can send events to each other, but also the coupled model can forward input events to its components (*external input couplings*) and the components can send output events to the coupled model (*external output couplings*). Couplings between components are called *internal couplings*. Figure 4.3 illustrates such a classic coupling scheme as used in, e. g., P-DEVS.

Above, we defined the operational semantics of P-DEVS informally. A formal definition of the semantics of P-DEVS is given by its abstract simulator, i. e., a simulation algorithm usually specified in pseudocode. The abstract simulator of P-DEVS can be found in Chow et al. [1994] and Zeigler et al. [2000, p. 284–7] or in the Appendix C.

4.2.2 Structured Systems and Structured Paralled DEVS

So far, we defined P-DEVS without making any assumptions about the nature of the states, inputs, and outputs of the models, i. e., the elements of the sets X, Y, and S. However, the

[22] Chow and Zeigler [1994] define the coupling scheme of P-DEVS differently. Instead of specifying a set of influencers for each component of the coupled model (including the coupled model), a set of influencees is specified. So for each component d, the set I_d contains the identifiers of all components (including the coupled model and excluding component d) that can be influenced by the component d. The output translation functions are defined accordingly. However, both approaches are equivalent.

[23] Sometimes the term “message” is used instead.

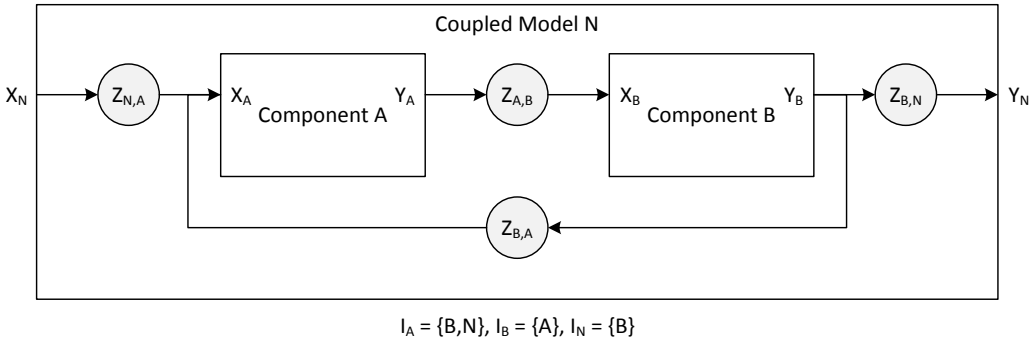

Figure 4.3: Example of a simple coupled model (adapted from Zeigler et al. [2000, p. 129]). The sets I_A, I_B, and I_N determine the influencers for the components "A" and "B" and the coupled model "N", respectively.

states, inputs, and outputs of the systems that we want to model are usually not opaque or abstract, flat entities, but structured according to certain variables—state, input, and output variables. Zeigler et al. [2000, p. 123] call such systems *multivariable* or *structured systems*. In addition to the classic specification of P-DEVS, as given above, there exists also a specification of P-DEVS at the *level of structured systems*, assuming the existence of, at least, input and output variables[24] (see, e. g., Zeigler et al. [2000, pp. 77–88]). In this, from a modeling point of view, more sophisticated specification, the sets of inputs and outputs of atomic and coupled models are refined by using input and output variables, which are called *ports*[25]. Via these in- and output ports components communicate with their surroundings. A first concept of ports was introduced to classic DEVS by Livny [1983, pp. B-1–B-2]. Zeigler et al. [2000, p. 90] define the sets X and Y of an atomic P-DEVS models using ports as follows:

$$X = \{(p, v) \mid p \in \mathit{InPorts}, v \in X_p\}$$
$$Y = \{(p, v) \mid p \in \mathit{OutPorts}, v \in Y_p\},$$

where *InPorts* and *OutPorts* denote the set of input ports and the set of output ports, respectively, i. e., their names. For each input port $p \in \mathit{InPorts}$, X_p denotes the range of values that can be assigned to p, i. e., that can be sent to this port. Accordingly, for each output port $p \in \mathit{OutPorts}$, Y_p denotes the range of values that can be sent or produced by the port. An input or output of the atomic model is then an ordered pair

$$(p, v)$$

comprising a port p and a value v at this port. The rest of the specification of the atomic P-DEVS model at structured system level is equivalent to Definition 4.2.1.

Remark. Note that also other approaches to structure the sets X and Y exist, such as *multivariable sets* as proposed by Zeigler et al. [2000, pp. 123–25]. Depending on the corresponding approach, the meaning of a single element of X and Y may vary. In the case of multivariable sets, one in- or output corresponds to a value assignment for each in- or output port.

At the level of coupled P-DEVS models, using ports lead to *port-to-port couplings*. Instead of specifying sets of influencers and translation functions, the modeler simply connects ports of

[24] A possible structuring of states is not (and does not need to be) explicitly captured in the definition of the formalism. However, when specifying concrete models, we can, e. g., use an n-fold Cartesian product $S_1 \times S_2 \times \ldots \times S_n$ to define the set S, where n refers to the number of state variables and S_i with $i \in [1, n]$ refers to the value range of the corresponding variable.

[25] Also in other modeling formalisms and approaches, such as UML, SysML, or Modelica, we find ports serving as central points for interaction between model entities.

different components with each other, which then exchange events during simulation. Since no translations have to be specified when using port-to-port couplings (values send by the source are received unchanged by the target), they facilitate a more compact and natural specification of coupling schemes. Port-to-port couplings are "almost exclusively used in modeling practice" [Zeigler et al. 2000, p. 129] and a special case of the general coupling scheme [Zeigler et al. 2000, p. 130] as used in Definition 4.2.2. In fact, port-to-port couplings can be translated back into a classic coupling scheme [Praehofer 1992, p. 43]. Following Zeigler et al. [2000, pp. 84–86], a coupled P-DEVS model at the structured system level (i. e., that makes use of ports and port-to-port couplings) can be defined as follows:

Definition 4.2.3 (Structured Coupled P-DEVS Model)
A **coupled P-DEVS model at the level of structured systems** is defined as the structure

$$N = \langle X, Y, D, \{M_d\}, EIC, EOC, IC \rangle,$$

where

- $X = \{(p, v) \mid p \in InPorts, v \in X_p\}$ is a *structured set of inputs* with
  - $InPorts$ being a set of *input ports* (i. e., their names),
  - X_p being an arbitrary set of values (*value range*) that can be assigned to the input port p;
- $Y = \{(p, v) \mid p \in OutPorts, v \in Y_p\}$ is a *structured set of outputs* with
  - $OutPorts$ being a set of *output ports* (i. e., their names),
  - Y_p being an arbitrary set of values (*value range*) that can be assigned to the output port p;
- D is a set of *component references*;

where for each $d \in D$

- M_d is an structured atomic P-DEVS model (*component*) with

  $$M_d = \langle X_d, Y_d, S_d, \delta_{int\,d}, \delta_{ext\,d}, \delta_{con\,d}, \lambda_d, ta_d \rangle,$$

  where
  - $X_d = \{(p, v) \mid p \in InPorts_d, v \in X_p\}$,
  - $Y_d = \{(p, v) \mid p \in OutPorts_d, v \in Y_p\}$;

and where

- $EIC \subseteq \big\{((N, ip_N), (d, ip_d)) \mid ip_N \in InPorts, d \in D, ip_d \in InPorts_d\big\}$ is a set of *external input couplings* connecting external input ports of the coupled model N to input ports of components;
- $EOC \subseteq \big\{((d, op_d), (N, op_N)) \mid d \in D, op_d \in OutPorts_d, op_N \in OutPorts\big\}$ is a set of *external output couplings* connecting output ports of components with external output ports of the coupled model N;
- $IC \subseteq \big\{((a, op_a), (b, ip_b)) \mid a, b \in D, op_a \in OutPorts_a, ip_b \in InPorts_b\big\}$ is a set of *internal couplings* connecting output ports of components to input ports of other components.

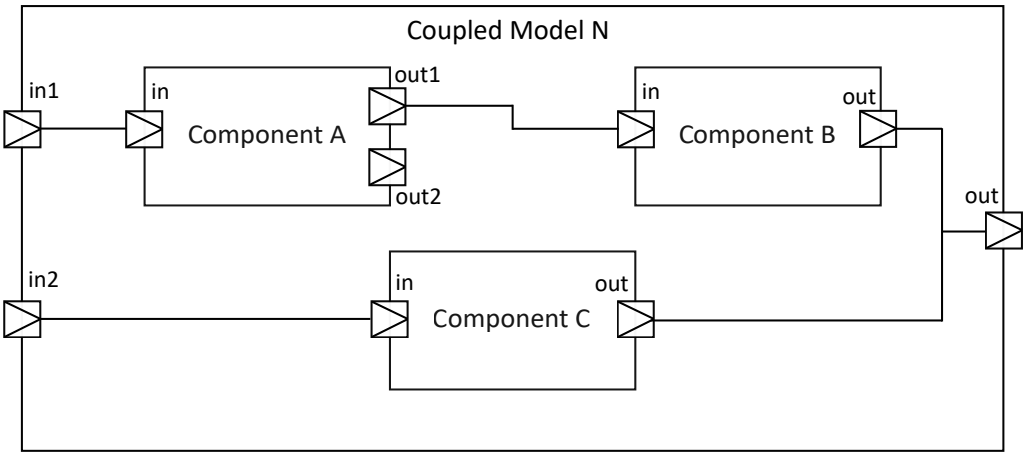

Figure 4.4: Example of a simple coupled model, in which the couplings are defined via port-to-port couplings. The sets EIC, EOC, and IC refer to external input couplings, external output couplings, and internal couplings, respectively.

Like in Definition 4.2.2, N refers to the actual definition of the coupled model as well as to its identifier that is used for defining the couplings.

The structured coupled P-DEVS model is subject to the following constraint:

$$\forall\,((a, op_a)(b, ip_b)) \in IC\colon\ a \neq b,$$

i. e., no direct feedback loops are allowed (no output port of a component shall be coupled to an input port of the same component). Furthermore, the value range of a *source port* (from-port) has to be a subset of the value range of the coupled *target port* (to-port), i. e.,

$$\begin{aligned}\forall((N, ip_N), (d, ip_d)) \in EIC\colon range_{ip_N} \subseteq range_{ip_d},\\ \forall((d, op_d), (N, op_N)) \in EOC\colon range_{op_d} \subseteq range_{op_N},\\ \forall((a, op_a), (b, ip_b)) \in IC\colon range_{op_a} \subseteq range_{ip_b}.\end{aligned}$$

So unlike classic coupling schemes, port-to-port couplings make restrictions on the value ranges of the coupled port. These restrictions are formulated as constraints that we have to adhere to in order to specify consistent models. Figure 4.4 shows an exemplary coupling scheme made from port-to-port couplings.

4.3 Summary

This chapter gives a brief introduction of discrete event simulation and its characteristics. Afterward, the chapter introduces the Discrete Event System Specification (DEVS) as one, well-established example for a modeling formalism that allows us to create models for conducting discrete event simulation. Furthermore, some important variants and extensions of DEVS are listed. The focus is on Parallel DEVS (P-DEVS), a parallel variant of DEVS. The chapter shows how models are specified in P-DEVS. Finally, a refinement of the definition of P-DEVS is given by assuming that the inputs, outputs, and states of a P-DEVS model are structured according to certain, interesting variables.
The modeling formalism presented in Chapter 9, i. e., Multi-Level DEVS (ML-DEVS), is based on P-DEVS and also assumes structured sets of inputs, outputs, and states.

5 Component-Based Modeling and Simulation

> Perfect reusable components are not obtained at the first shot.
>
> Bertrand Meyer

As noted in the introduction, many systems of interest—ranging from smart environments over living cells to populations—consist of smaller, distinct parts: *system components*. Those system components can be homogeneous or heterogeneous, and each component can be considered as a system itself (cf. Chapter 2). The behavior of an overall system emerges from the interaction of its components. Also, the behavior can often not be understood by investigating the components in isolation. Like Aristotle said

> The whole is greater than the sum of its parts.

This chapter introduces the basic ideas of component-based modeling and simulation and outlines the differences between similar approaches concerning with the creation of models (of complex systems) by using smaller "units," either top-down or bottom-up.

5.1 Evolution and Basics

"In modeling and simulation, there is a growing interest for developing larger and more complex models [of complex systems] through model composition" [Szabo & Teo 2007], preferably by composing predefined *building blocks*[1] [Röhl 2008, p. 1]. The "composition of models is considered essential in developing heterogeneous complex systems and in particular simulation models capable of expressing a system's structure and behavior" [Sarjoughian 2006]. This leads us to the idea of *component-based modeling and simulation*. Intuitively, we can think of component-based modeling and simulation as a paradigm or method for (i) building complex models out of *components* (parts) by composing and connecting them and (ii) executing such composed models; where the focus is often on model creation (*component-based modeling*) rather than execution (*component-based simulation*). We call the parts of which a *composed model* consists of *model components* to distinguish them from the system components, i. e., the constituent parts of the system that is mimicked by the model. Depending on the required level of abstraction for a modeling problem at hand, a model component can represent an individual system component or aggregate several system components, or, in turn, a system component can be represented by several, interacting model components. However, often the model composition reflects the actual organizational structure of the system of interest.

Although we find a lot of literature on the subject, there is a lack of a clear-cut definition of the notion of component-based modeling and simulation. In fact, a lot of similar or equal concepts exists that are closely related to the idea of creating models by assembling some sort of building blocks or smaller units. Some of these related concepts are:

- Modular modeling, hierarchical modeling (cf. Zeigler [1984] or Röhl [2008]);
- Object-oriented modeling and simulation (cf. Zeigler [1990]);
- Composability, interoperability (cf. Petty and Weisel [2003a], "The Levels of Conceptual Interoperability Model" [2003], Szabo and Teo [2007], P. K. Davis and Tolk [2007], or Tolk and Miller [2011]);
- Compositionality (especially with respect to modeling languages, cf. de Roever, Langmaack, and Pnueli [1998], Henzinger, Jobstmann, and Wolf [2009], or "MontiCore: A Framework for Compositional Development of Domain Specific Languages" [2010]);
- Parallel, distributed modeling (cf. Fujimoto [2000], Verbraeck [2004] or Tolk [2013]),
- Multi-formalism modeling (cf. Sarjoughian [2006]),
- Multi-paradigm model (cf. Vangheluwe et al. [2002]); etc.

Since there exists, to our best knowledge, no comprehensive taxonomy relating all these concepts and the differences between them are often rather subtle, it is not alway easy to keep them apart. In the remainder of this chapter, we discuss some of the above concepts in more detail, especially those that are of importance for the thesis. However, it is not subject of the thesis to introduce a well-defined taxonomy or ontology that relates the different composition methods and concepts.

Despite the absence of a clear-cut definition, we often find certain, recurring characteristics that are associated with component-based model design and are used as a motivation for this paradigm:

- Reduced complexity of individual components (cf. Valentin and Verbraeck [2002]);
- Reduced time and costs of the development of models, by reusing components (cf. Szabo and Teo [2007]);

[1] In contrast to Verbraeck [2004], we use the terms "building blocks" and "components" interchangeably.

- Reusability of predefined, of-the-shelf, and customizable components (cf. Chen and Szymanski [2002]);
- Reduced effort of performing simulation studies (cf. P. K. Davis and Anderson [2004]; Valentin et al. [2003]);
- Increased credibility of the composed models (cf. P. K. Davis and Anderson [2004]).

The first bullet point resembles the idea of *divide and conquer*, an algorithm design paradigm, in which a complex problem is broken down into simpler subproblems (top-down). Similarly, a component-based model design allows representing a complex system by a number of smaller, less complex components (instead of a large monolithic model). In addition, a component-based approach allows the modeler to adhere to the general design principle *separation of concerns* [Dijkstra 1982, pp. 60–66], in which different concerns are represented by different components. For this reason, many modeling formalisms support a composition of models from smaller units, in one way or another.

5.1.1 Component-Based Software Engineering

It is often noted that component-based modeling and simulation is related to or inspired by *component-based software engineering* (CBSE), *component-based development* (CBD), and *component-oriented programming* (COP) [Dalle 2007; Hu et al. 2005; Röhl 2006; Sarjoughian & Huang 2005; Verbraeck 2004]. According to Jifeng, Li, and Liu [2005], component-based software development (or component-based software engineering) can be viewed as using "reusable components that interact with each other and fit into system architectures," where "the idea to exploit and reuse components to build and maintain software systems goes back to 'structured programming'[2] in the 70s." Heineman and Councill [2001, p. xviii] write that CBSE mostly focuses on three aspects:

1. Developing software from preproduced parts (components),
2. The ability to reuse those components in other applications,
3. Easily maintaining and customizing those components to produce new functions and features.

As a result, reusing predefined components allows programmers to reduce the costs of developing and maintaining software (*reuse-based software engineering*). Furthermore, CBSE "should provide both a methodology and process for developing components that work continuously, with the ability to return to previous stable state when encountering an error without corrupting any components," hence "CBSE is concerned less with building parts [components] than providing users with constantly reliable parts that maintain continuously functioning software" Heineman and Councill [2001, p. xix]. Component-oriented programming, on the other hand, "focuses on the design and implementation of components—in particular, on the concepts of encapsulation, polymoprhism, late binding and safety" [Szyperski 2002, p. 549]. As such, COP is closely related to *object-oriented programming*[3]. Components or software components are the central ingredients of CBSE, CBD, and COP. Szyperski [2002, p. 548] defines as software component as follows:

> A [software] component is a unit of composition with contractually specified interfaces and explicit context dependencies only. Context dependencies are specified

[2] Structured programing introduces *subroutines* that allow reusing certain functionalities by calling the subroutines.

[3] According to Szyperski [2002, p. 549, p. 561], both component-oriented programming and object-oriented programming (OOP) focus on the design and implementation of entities, i. e., components and objects. Both COP and OOP build on the concepts of encapsulation and polymorphism. In addition, COP focuses on late binding and safety, whereas OOP focuses on implementation inheritance.

> by stating the required *interfaces* and the acceptable execution platform(s). A [software] component can be deployed independently and is subject to composition by third parties. For the purposes of independent deployment, a [software] component needs to be an executable unit. To distinguish between the deployable unit and the instances it supports, a [software] component is defined to have no observable state. Technically, a [software] component is a set of atomic components, each of which is a module plus resources. A [software] component targets a particular component platform. The composition of components follows one or more composition schemes that are mandated by that component platform.

So software components are subject of composition, which Szyperski [2002, p. 550] defines as:

> Assembly of parts (components) into a whole (a composite) without modifying the parts. Parts have compositional properties if the semantics of the composite can be derived from those of the components.

According to Szyperski, one essential aspect of a composition is that its parts are composed without being modified. In addition, the assembly or composition of two or more software components yields a new component behavior [Weinreich & Sametinger 2001, p. 42].

"As simulationists, we can learn from component-based theory from the software engineering field to prepare our models for distribution, and parts of our models for reuse," however "it is not easy to create models in a componentized way" [Verbraeck 2004]. According to Verbraeck, object orientation and component-based development appear to be natural approaches when it comes to partitioning of software and thus models. This leads us to modular-hierarchical and object-oriented modeling.

5.1.2 Modular-Hierarchical and Object-Oriented Modeling

In contrast to monolithic modeling formalisms such as finite state machines [Hopcroft, Motwani, & Ullman 2001] or classic cellular automata [von Neumann 1966], many formalisms that focus on modeling complex, multi-component systems intrinsically support a modular, hierarchical model construction [Steiniger & Uhrmacher 2013]. Modular modeling describes models of complex systems as being composed of smaller, self-contained, and interacting parts or units: *modules*, where the emphasis is on the self-containment of the individual parts. In other words, modular modeling deals with the modularization of models. A modular model specification is necessary for a flexible model assembly and disassembly [Zeigler 1984, p. 132]. But what are modules and what are their characteristics? From a more general perspective (software engineer), Szyperski [2002, p. 559] defines a module as follows:

> A closed static unit that encapsulates embedded abstractions. Such abstractions include types, variables, functions, procedures, or classes. As a module is a closed unit, its encapsulated domain is fixed and can be fully analyzed.

Zeigler [1984, p. 132] gives the following, more specific description of a module:

> We shall understand a module to mean a program text that can function as a self-contained autonomous unit in the following sense: Interaction of such a module with other modules can occur only through predeclared input and output ports. Except for such interface variables, all other variables referenced in the module receive declarations local to it. This module may contain memory (saved) variables which retain their values between invocations.

Furthermore, Zeigler notes that "a module is well characterized as an I/O system" that is defined by: (i) an input and output interface via which the systems interacts with other systems and (ii) a state representing the memory of the system. Thus, modules are not only

characterized by their self-containment but also by their interfaces, which define the boundary of a module to its environment. Pursuing this argumentation, modules can be considered as models themselves.
Via their interfaces, modules can be coupled (connected) with each other forming composite modules (often called *coupled models*). These composite modules may in turn become part[4] of other composite modules at a higher level [Zeigler 1984, p. 133], leading to *hierarchical models*. More formally, Zeigler describes a hierarchical model construction as a finite recursion of couplings of modules at the same level and gives the following inductive definition of a hierarchical model in Zeigler [1990, p. 29]:

1. an atomic model is a hierarchical model[5]
2. a coupled model whose components are hierarchical models is a hierarchical model
3. nothing else is a hierarchical model,

where an atomic model is a module that cannot be decomposed into further submodules. "The starting point for hierarchical model construction is the closure of the systems formalism [modeling formalism] under coupling" [Zeigler 1984, p. 149]. This means that coupled models in a respective formalism can be also expressed as atomic models in this formalism. Hence, closure under coupling provides the formal foundation for hierarchical modeling.

Prominent examples of modeling formalisms that facilitate a modular, hierarchical model design are DEVS [Zeigler & Sarjoughian 1999] and many of its variants (see Section 4.2) as well as Modelica [Elmqvist 1978; Elmqvist, Mattsson, & Otter 2001]. Modelica is an object-oriented modeling language with a special focus on modeling physical systems consisting of mechanical, hydraulic, or electrical components. In the language, models are defined by parameterizable classes with typed ports via which models can be connected to form more complex models and the substitution of submodels is based on inheritance (cf. Mattsson and Elmqvist [1998], Otter and Elmqvist [2000], or Otter, Erik Mattsson, and Elmqvist [2007]).

Based on the above definitions, we can conclude that every modular model can also be considered as a hierarchical model, even if all components of this hierarchical model are atomic. However, not every hierarchical model may be a modular model according to the definition of modules given by Zeigler. A special case of hierarchical models are *non-modular, hierarchical models*. Similar as modular, hierarchical models, these models consist of smaller parts of some sort that interact with each other. Furthermore, a coupled, non-modular model can be a part of another coupled, non-modular model (closure under coupling). But these parts do not or only partially interact with each other via clearly defined interfaces. Instead, parts can directly change the state of other parts, because non-modular models do not encapsulate their states, in contrast to modular models. Nevertheless, Zeigler [1984, pp. 137–41] and Zeigler et al. [2000, pp. 161–2] present a procedure how non-modular, hierarchical models can be translated into a modular form, by "identifying the dependencies between components and converting them into input and output interfaces and modular couplings."

One example of a modeling formalism that allows specifying non-modular, hierarchical models is multiDEVS (multicomponent DEVS) as described by Zeigler et al. [2000, pp. 155–7] or Shiginah [2006, pp. 25–7]. The state transition of each "component" in multiDEVS is defined on the states of all components of the superordinate coupled model and can change the state of the current component as well as all the other components. However, each component of multiDEVS is a well-defined model that can be taken from the respective coupled model and executed individually.

[4] Zeigler [1984, pp. 132–57] refers to modules also as components. Here we try to distinguish between both, even though both concepts are closely related.

[5] Zeigler et al. [2000, p. 93] describe a hierarchical model as coupled models that consists of atomic and coupled models.

Zeigler [1984, p. 133] notes that "we should be careful to distinguish the characterization of a module in [a] system theoretic formalism from its realization in programming form." In other words: there can be a difference between a model specified in a formalism and the model that is eventually executed on a computer adhering to the model specification given in the formalism (*executable model*). This applies also for other modeling formalisms, especially those that are based on some sort of mathematical notation or formal calculus. In fact, such modeling formalisms "can have several implementations (i.e., software realizations) based on the choice of programming languages—e.g., a mathematical model can be designed and implemented using object-oriented modeling concepts and a programming language" Sarjoughian [2006]. Which brings us to *object-oriented modeling*.

"Object orientation and component-based development immediately spring to mind as possible technologies to use when it comes to partitioning of software or models [dividing the model into smaller parts]" [Verbraeck 2004]. "Object-oriented modeling is coined by the encapselating [sic] of data and procedures in objects, which results in a modular design of models" and interact with each other via messages [Uhrmacher & Zeigler 1996]. Herein, we understand object-oriented modeling as a special case of modular modeling. Models that are implemented by using an object-oriented programming language are directly executable, whereas not all modular models are executable on a computer without further ado.

Modelica, as mentioned earlier in this section, is an example of an object-oriented modeling formalism. In contrast, DEVS itself is not object-oriented, because the models are specified set-theoretically. However, "DEVS [as well as other modeling formalisms] is most naturally implemented in computational form in an object-oriented framework" Zeigler et al. [2000, p. 93]. For this reason, a number of object-oriented implementations of DEVS and its variants exist, such as the modeling environment DEVSJAVA [Zeigler et al. 2000, pp. 93–5] or the DEVS plugins for the simulation framework JAMES II[6] [Himmelspach & Uhrmacher 2007]. The existence of different implementations of a modeling formalism such as DEVS can yield the problem that models specified using different implementations are not interoperable with each other (cf. Garredu, Vittori, Santucci, and Bisgambiglia [2013]). For instance models created in the environment DEVS-Scheme [Zeigler 1990] cannot be coupled with models created in DEVSJAVA.

5.1.3 Component-based Modeling

According to Verbraeck [2004] "object orientation does not guarantee reuse, nor does it guarantee that objects easily communicate with each other," but "component-based development (CBD) [or component-based software engineering (CBSE)] has addressed this issue for the software-engineering world by aiming at reusable building blocks with clearly defined interfaces." *Component-based modeling* (and simulation) adopts these ideas from CBSE and transfers them to the model design, while emphasizing a separation between composition and implementation. As such, component-based modeling goes one step further than modular-hierarchical modeling, in which composition and implementation are inseparable. Verbraeck distinguishes between building blocks and components. Verbraeck [2004] defines a *building block* as follows:

> A building block is a self-contained, interoperable, reusable and replaceable unit, encapsulating its internal structure and providing useful services or functionality to its environment through precisely defined interfaces. A building block may be customized in order to match the specific requirements of the environment in which it is 'plugged' or used.

As a building block hides its internal structure, customizing a building block should be done without changing the internal structure. This can be done by defining the internal structure

[6] `http://jamesii.org`; last accessed February 2018

in a way that it depends on certain parameters, which can be set from the outside. Creating building blocks in a customizable way has influence on the development of the building blocks. Furthermore, Verbreack writes:

> A building block is independent of its implementation and can therefore only [be] described by its conceptual model and therefore by its function and interface with other building blocks. This means that independent of the implementation, a system consisting of a mixed implementation of different building blocks. For instance a combination of a simulation and a real-time application building block should function without problems.

According to Verbraeck [2004] building blocks have, among others, the following properties:

- different levels of granularity (abstraction),
- fulfill a clear function,
- have a well-specified (or well-defined) interface,
- are nearly independent of other building blocks, but rarely stand alone,
- can be used in unpredictable combinations,
- may change their state.

Verbraeck [2004] explicitly distinguishes between building blocks and components, where "a component is the implementation of a building block in a software environment." "The interface (functionality) of the building block and the component are therefore different representations of the same thing." Similar to other works such as Himmelspach et al. [2010], we will not make such a distinction and refer to components rather than building blocks.

A component-based modeling approach provides a methodology how to specify such components, their interfaces, and how to configure and compose them. As Kasputis and Ng [2000] write:

> We are discovering that unless models [components] are designed to work together, they don't (at least not easily and cost effectively). Without a robust, theoretically grounded framework for design, we are consigned to repeat this problem for the foreseeable future.

Another, very important aspect of component-based modeling is the exchange and reuse of predefined components by third parties [Röhl 2006; Verbraeck 2004]. Therefore, components can be stored in and retrieved from component repositories (cf. Szabo and Teo [2007]).

Although, a component is supposed to be independent of its implementation and a composition can consist of mixed implementations (in terms of the implementation techniques) [Verbraeck 2004], we find component-based modeling approaches that more or less restrict or dictate the underlying implementation technique, such as Varga [2001], Chen and Szymanski [2002] or Buss and Blais [2007], putting less emphasize on a clear separation of composition and implementation.

5.2 Composability and Interoperability

The term *composability* is closely related to component-based modeling and simulation. Petty and Weisel [2003a] define composability as follows:

> Composability is the capability to select and assemble simulation components in various combinations into valid simulation systems to satisfy specific user requirements. The defining characteristic of composability is the ability to combine and recombine components into different simulation systems for different purposes.

Likewise, Szabo and Teo [2007] define composability as

> the capability to select and assemble off-the-shelf model components in various combinations to satisfy user requirements.

So composability does not simply correspond to the process of putting components together but also to the validity (correctness) of and requirements to the outcome of a composition. Furthermore, Petty and Weisel distinguish between *syntactic composability* and *semantic composability*[7]. Syntactic composability answers the question whether "components can be connected" (in principle), whereas semantic composability deals with the question "whether the models that make up the composed simulation system can be meaningfully composed, i. e., if their combined computation is semantically valid" [Petty & Weisel 2003a]. "To be syntactically composable, [. . .] components have to be compatible with respect to data passing mechanisms and timing assumptions" [Szabo & Teo 2007]. For this reason, the notion of the *compatibility* of components, more specifically their interfaces, being connected, as we can find in, e. g., P. C. Davis, Fishwick, Overstreet, and Pedgen [2000], Jifeng et al. [2005], Röhl and Uhrmacher [2008], and Modelica Association [2012], corresponds to syntactic composability. However, although syntactic composability is "by far easier to achieve than semantic composability, syntactic composability still poses a number of problems such as establishing a common component model by which all components involved in the syntactic composability must abide, and, derived from the component model, the way in which syntactic checking is done" [Szabo & Teo 2007].

Assuring semantic composability, on the other hand, solely based on component and composition descriptions, when configuring an assembly of components, is an even more challenging task, as semantic composability refers to the validity of the composition [Petty & Weisel 2003a]. Traditional *model validation*, however, requires experiments to be conducted and the "comparison" of the behavior of the model with the behavior of the actual system with respect to a certain modeling problem (see Section 2.5). Therefore, addressing semantic composability requires the use of additional "tools" such as ontologies. These challenges may be the reason why there is—to our best knowledge—only few works that tackles semantic composability explicitly, such as Petty and Weisel [2003b], Szabo and Teo [2009], or Peng, Ewald, and Uhrmacher [2014]. In fact, we could ask ourselves why it is necessary to think about semantic composability, since the validation is a crucial step in simulation studies, which we may not want to or cannot skip.

Given a certain composition methodology allowing us to configure and assemble of-the-shelf components, as characterized above, we can assume that at least syntactic composability can be assured by using the respective methodology.

Another concept related to composability, we come across when dealing with component-based modeling and simulation, is *interoperability*. Petty and Weisel [2003a] define interoperability as follows:

> For simulations, interoperability is the ability of different simulations [simulation systems], connected in a distributed simulation system, to meaningfully collaborate to simulate a common scenario and virtual world. Their collaboration is normally based on run-time exchange of simulation data or services, typically using an interoperability protocol, such as DIS [Distributed Interactive Simulation], ALSP [Aggregate Level Simulation Protocol], or HLA [High-Level Architecture].

So interoperability is at a higher level than composability, since it involves the interaction of simulation systems rather than the interaction of components, which are executed on the simulation systems. Figure 5.1 illustrates the relationship between both.

[7] Petty and Weisel [2003a] also use the terms "engineering composability" and "modeling composability" instead of "syntactic composability" and "semantic composability," respectively.

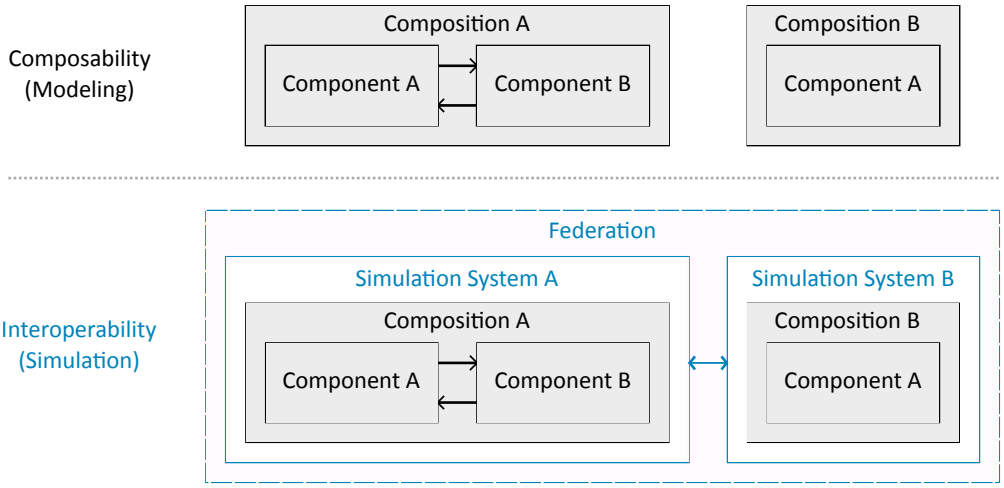

Figure 5.1: Relationship between composability and interoperability.

Similarly as done for composability, Petty and Weisel [2003a] distinguish two types of interoperability: *technical interoperability* and *substantive interoperability* (or *meaningful interoperability*). The former covers the correct use of the *interoperability protocol*[8], whereas the latter assures that the information exchanged between the interoperating simulations is mutually consistent with the simulation models' semantics. "Essentially, interoperability is the ability to exchange data or services at run-time, whereas composability is the ability to assemble components prior to run-time", where "interoperability is necessary but not sufficient to provide composability" [Petty & Weisel 2003a]. Herein, we focus on composability rather than interoperability.

5.3 Component-based Simulation

As the Introduction of this thesis emphasizes, we create model for the purpose of simulation. Component-based simulation may refer to one or more of the following aspects:

- Component-based design of simulation systems, i. e., the simulation systems consist of different interacting software components, which sometimes can be seen as software agents (cf. *agent simulation* as described by Yilmaz and Ören [2007]).
- Component-based design of simulation algorithms, i. e., the simulation algorithms consist of components that may be exchanged during simulation (cf. adaptive simulation algorithms as discussed by Helms, Ewald, Rybacki, and Uhrmacher [2013]). A simulation algorithm itself is a part (component) of a simulation system.
- The execution of component-based models (model compositions).

Herein, we focus on the latter, i. e., the execution of component-based models, when talking about component-based simulation.

By definition, the internal structure of a component (building block) is hidden and independent of its implementation [Verbraeck 2004]. In the most simple case, all components of a composition and their interaction can be specified in the same, sufficiently expressive modeling formalism, such as DEVS or SysML[9]. If so, executing the model composition is straightforward, as we only need a simulation system that supports the corresponding modeling formalism. For the actual execution the composed model may be partitioned into

[8] The means by which the simulation systems interoperate.

[9] At some occasions, we will call such a general purpose and expressive modeling formalism also *super formalism*.

smaller parts[10] and distributed among different computation nodes by the simulation system (parallel and distributed simulation).

However, often systems under consideration have parts (components) with dynamics that are intrinsically different [Sarjoughian 2006]. Even though we may, in principle, be able to describe all components in the same modeling formalism, that does not necessarily mean that all components should be described in this formalism. Sometimes one formalism may be more suitable than another to specify a certain component or type of components (no silver bullet exists[11]). Thus, we should use different modeling formalisms to model composed systems [Sarjoughian 2006], especially if they consist of heterogeneous components. However, the "decomposition and composition of models are challenging when models are heterogeneous in terms of their formal specification [as they] have different structural and behavioral specifications" [Sarjoughian 2006]. Assembling components whose internal structure is defined in different modeling formalisms is closely related to the ideas of *multi-formalism modeling* and *multi-paradigm modeling*. Therefore, we can adopt strategies from both domains for the execution of composed, multi-formalism models. Vangheluwe [2000] identifies three basic approaches for executing multi-formalism models, which we can also apply for the execution of model compositions:

- **Meta-Formalism:** The formalisms used to implement the different submodels (*source formalisms*) are subsumed (combined) to a single *meta-formalism*. Thus, all submodels are specified in the same formalism, i. e., the meta-formalism. This formalism is then used to execute the coupled multi-formalism model on a simulation system that supports the meta-formalism. However, "meaningful meta-formalisms which truly add expressiveness as well as reduce complexity are rare" Vangheluwe [2000]. In addition to merge the model specifications of the source formalisms, a meta-formalism also needs a well-defined execution semantics to facilitate simulation. For this reason, the number of possible source formalisms that can or should be subsumed to a meta-formalism is limited. The *Discrete Event and Differential Equation System Specification* (DEV&DESS), introduced by Praehofer [1992], is an example of such a meta-formalism that combines discrete event simulation with continuous simulation[12].
- **Transformation:** To execute a coupled multi-formalism model on a certain simulation system, the components, which are specified in different formalisms, are *transformed* into a suitable common formalism (*target formalism*). To do so, transformations from the source formalisms to the target formalism have to exist. In addition, the target formalism needs to support the coupling of model components. Vangheluwe [2000] shows that DEVS can serve as such a target formalism ("common denominator") for multi-formalism, hybrid systems modeling. Furthermore, Vangheluwe suggest exploiting the possible closure under coupling of the source and target formalisms to come up with flat models before and after the transformation into the target formalism. Figure 5.2 illustrates the procedure suggested by Vangheluwe, which leads to a single, flat model in the target formalism. This model can eventually be executed on a simulation system that supports the target formalism. We have a less restrictive view on the transformation of model compositions, as we do not demand that the outcome of a transformation has to be a flat model.
- **Co-Simulation:** Each submodel of the coupled multi-formalism model is executed by a source-formalism-specific simulator (simulation algorithm). These simulators can run

[10] The individual parts can be greater than the components of which the model is composed of.

[11] as postulated Brooks Jr. [1987]

[12] DEV&DESS combines the formalism DEVS for discrete event simulation with the formalism DESS (*Differential Equation Specified Systems*) [Zeigler 1976, pp. 229–31] for continuous simulation [Praehofer 1992; Zeigler et al. 2000]. Thus, DEV&DESS can be used for hybrid systems modeling and simulation (multi-paradigm modeling). First ideas to combine DEVS and DESS were already introduced in Praehofer [1991].

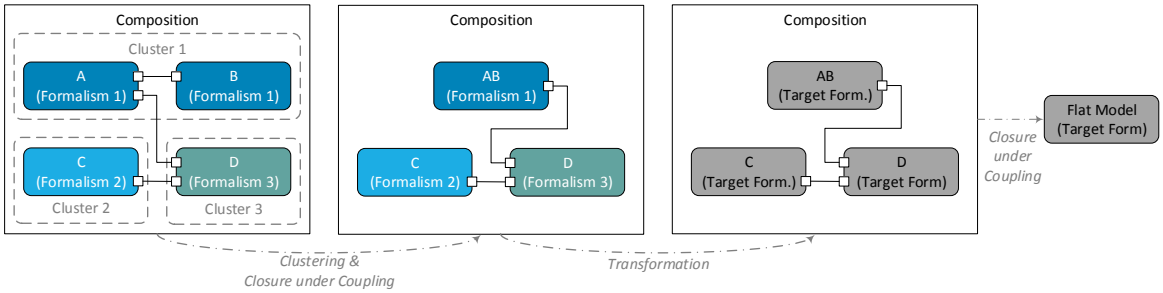

Figure 5.2: The transformation of a coupled multi-formalism model as described by Vangheluwe [2000]: After the consistency of the coupled model is checked, all submodels (components) that are specified in the same formalism are clustered. Then, each cluster of interacting submodels is flattened according to the closure under coupling property of the respective formalism. Afterward, the flattened models and remaining submodels are transformed into a suitable target formalism. Finally, the closure under coupling property of the target formalism is used to come up with a single, flat model. In the case that the coupled multi-formalism model is more complex, i. e., has more than one level, we apply the above procedure recursively, starting from the bottom level.

on the same or different simulation systems, which can be distributed among different computers. Interaction between submodels due to couplings is resolved at trajectory level, thus questions can only be answered at this level [Vangheluwe 2000]. More specifically, interaction is realized by the *interoperation* of simulators or simulation systems. This interoperation is the basis for, e. g., the "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)–Framework and Rules" [2010].

Due to the similarity of coupled multi-formalism models and model compositions, we can adopt the above approaches, described by Vangheluwe [2000], for the execution of model compositions, i. e., component-based simulation. In contrast to specifying the entire model composition in the same modeling formalism in which the composition is executed, transforming the composition into a common modeling formalism for execution allows us to use the "most suited" modeling formalism for specifying each model component[13]. Still all model components may be specified in the same formalism. Table 5.1 compares the different approaches with respect to source and target formalism and the simulation system. In the first approach (super formalism), the model composition is specified in the same formalism in which the composition is executed. Similarly, all components of a model composition are specified in the same formalism, i. e., the meta-formalism, in the second approach. However, in contrast to the first approach, the second approach provides more flexibility as components can be described in different modeling formalisms if they are part of the meta formalism. In the co-simulation approach, for each component the source formalism is also the target formalism in which the component is executed.

If multiple formalisms are used this "requires that not only individual model specifications are executed correctly, but also their compositions (i. e., the execution of multiple execution algorithms are well-defined with respect to the composition of their model specifications)" [Sarjoughian 2006].

5.4 Composition and Analysis Framework COMO

In the previous sections, we repeatedly mentioned the composition and analysis framework COMO [Röhl 2006, 2008; Röhl & Uhrmacher 2006, 2008], which allows a component-based

[13] As long as corresponding transformations between the different modeling formalisms exist.

	Approaches			
	Super formalism	Meta formalism	Transformation	Co-simulation
Source formalism(s)	1	$1^{(*)}$	$1 \dots n$	$1 \dots n$
Target formalism	1	1	1	$1 \dots n$
Simulation systems	1	1	1	$1 \dots n$

Table 5.1: Comparison of the different approaches for specifying and executing model compositions with respect to (i) the source formalisms in which the components are specified, (ii) the target formalism in which the model composition is executed, and (iii) the simulation system which executes the model composition.

model design as described in Section 5.1.3. Since COMO plays a role later on, we briefly describe it in the following (cf. also Steiniger and Uhrmacher [2013]).

The interface and composition description formalism that underlies COMO allows the modeler to specify interfaces of components, which encapsulate models, and their composition explicitly, in a modeling-formalism- and platform-independent way. Thereby COMO allows us to keep the implementations of components separate from their descriptions and interfaces. The description formalism itself is defined set-theoretically and represented by XML. Hence, composition descriptions consist of a number of XML documents that can be stored in or retrieved from a repository. If this repository is accessible by third parties, components can be reused by others. Addressing composability, COMO also analyzes the syntactic correctness of compositions and allows us to make a statement on the correctness of a given composition before executing it. For execution, COMO creates an executable simulation model in a suitable target formalism from a composition description and the implementations of the involved components by synthesizing parts of the simulation model automatically and transforming the component implementations into the target formalism. The execution itself is not carried out by COMO, as it does not provide an execution engine. However, COMO can be added as an additional specification and analysis layer on top of a simulation system, which is then used as the execution engine, such as JAMES II[14].

COMO is using P-DEVS as a target formalism but is not limited to it. The individual components can be "implemented" in different modeling formalisms as long as they support the concept of ports and there exists a transformation from the modeling formalism into the target formalism.

5.5 Summary

This chapter starts with giving a brief introduction and motivation of component-based modeling and simulation and its evolution over the last decades, starting from its roots in component-based software engineering. In doing so, the chapter lists different concepts that are closely related to component-based modeling, such as modular-hierarchical modeling, object-oriented modeling, multi-formalism modeling, and composability. The chapter shows that there is a difference between modular-hierarchical modeling and component-based modeling. The latter of which focuses on creating reusable, replaceable, retrievable, customizable, and

[14] `http://jamesii.org`; last accessed February 2018

self-contained components, which hide their implementation (separation between interface/-component description and component implementation) and can be used in different contexts or by third parties. The role of interfaces as a mean to decoupled composition descriptions from component implementations is also highlighted. Afterwards, the chapter describes the concepts of composability and interoperability, where the former plays a crucial role for component-based modeling. Finally, the chapter gives some remarks on the execution of composition made up of of-the-shelf components (component-based simulation) and describes briefly the composition and analysis framework COMO and its underlying composition and interface descriptions.

6 Dynamic Structure Systems and Variable Structure Models

> Nature means change, only some things change faster than others.
>
> Rick Marsi

Many interesting systems, real or imaginary ones, which should be studied by the means of modeling and simulation, are characterized by a dynamic or variable structure, i. e., their structure changes over time. We call such systems *dynamic structure systems*. Systems with a variable structure range from socio-technical systems (such as smart environments) over biological systems (such as eukaryotic cells) to demographic or sociological systems (such as the population of a certain geographic region or country).

Capturing the structure variability of a system of interest in its model is the objective of *variable structure modeling*, leading to *variable structure models*.

In this chapter, we take a closer look at the characteristics of dynamic structure systems and illuminate the different aspects of a variable structure model, which include more than just the composition of the model.

Parts of this chapter are based on and extend the following publications:

Steiniger, A. and Uhrmacher, A. M. (2016). "Intensional Couplings in Variable Structure Models: An Exploration Based on Multilevel-DEVS." In *ACM Transactions on Modeling and Computer Simulation* (TOMACS), 26(2). pp. 9-1–9-27.

6.1 Dynamic Structure Systems

When we are talking about the *dynamics* of a system, we usually think about how the system behaves over time (cf. Zeigler et al. [2000, p. 13]). According to Zeigler et al. [2000, p. 4] the *behavior* of a system is "the relationship it imposes between its input time histories [input trajectories] and output time histories [output trajectories]," whereas the *structure* of a system "includes its state and state transition mechanism (dictating how inputs transform current states into successor states) as well as the state-to-output mapping." The above definition of the system behavior assumes that a system has tangible outputs, which may not always be the case; or we may not be interested in the outputs of a system. Therefore, we will also refer to state time histories (state trajectories) when talking about the behavior of a system. In this thesis, we call systems that can change their state and possible outputs with respect to the flow of time and exogenous stimuli (system inputs) *dynamic systems*[1].

"Many kinds of real systems are most readily perceived as exhibiting changes simultaneously at structural and behavioral levels" [Zeigler & Praehofer 1990], especially biological and other adaptive systems [Zeigler & Ören 1986]. This means, such systems change not only their state and possible outputs over time but also their internal structure. "Generally, where living autonomous entities are involved, changes in interactions, composition, and behavior patterns occur frequently" [Uhrmacher 2001]. For instance, organelles of living, eukaryotic cells (e. g., mitochondria) can be degraded (*autophagy*) and reproduced (*biogenesis*) over the cells' lifespan (cf. Palikaras, Lionaki, and Tavernarakis [2015]); or connections between neurons of an organism's nervous system can change over time, allowing or prohibiting neurons to interact with each other (synaptogenesis, cf. Huttenlocher and Dabholkar [1997]). A further example of structure variability is the plasticity of the interfaces of some systems, such as *operons* that are part of the DNA of prokaryotes[2] and control the transcription of genes (*gene regulation* as described by Jacob and Monod [1961]). If the operator region of an operon (e. g., the Tryptophan operon or Trp operon) is bound to an active repressor protein, the repressor protein obstructs the RNA polymerase to bind to the operator region of the operon and thus inhibits gene transcription (cf. Uhrmacher et al. [2006]). In the tradition of Barros [1997], we call systems that can change their structure *dynamic structure systems*, which can be considered as a subset of the aforementioned dynamic systems.

Definition 6.1.1 (Dynamic Structure System)
A **dynamic structure system** is a system that is able to change its structure, e. g., its composition, autonomously or as an reaction to external inputs.

So each dynamic structure system is a system, by definition, but not all systems have a dynamic structure. Man-made, technical systems, such as cars and other vehicles, are examples for systems with a static structure (*static structure systems*).

6.2 Variable Structure Models and Variable Structure Modeling

When modeling dynamic structure systems, it seems to be intuitive and just natural to reflect the structure variability of such systems by models that can also change their structure during model execution, i. e., simulation [Deniz 2010; Hu et al. 2005]. According to Ören [1975], Ören and Zeigler [1986], or Zeigler and Praehofer [1990], this reflection requires a new simulation paradigm, *structural simulation*, which contrasts conventional *trajectory*

[1] According to Ljung and Glad [1994, p. 40], "the present output value [of a dynamic system] depends, in principle, on all earlier input values [of this system]." This can be achieved by encoding previous inputs in the state of the dynamic system. In contrast, Ljung and Glad characterize a static system as one "whose variations in the output [...] are directly coupled to the momentary value of the input."

[2] and some eukaryotes

simulation and prevents us to describe structure changes at the same level as behavior changes [Zeigler 1990, p. 15]. This leads us, eventually, to *variable structure models*[3] (or *dynamic structure models*), which "lend structure to the temporal dimension in describing [dynamic structure] systems" [Uhrmacher et al. 2006]. Although, "variable structure models can be 'simulated' by static structure models by expanding the state space in order to enfold all possible structural changes and by equipping the transition functions with intricate conditional structures" [Uhrmacher 2001], variable structures provide "a natural and effective way to model those complex systems that exhibit structure and behavior changes to adapt to different situations" [Hu et al. 2005]. "Encoding a variable structure model in a static frame makes for a less elegant and coherent model design" Uhrmacher [2001]. In general, "it is less the question whether a formalism is able to express certain phenomena, but how easily this can be done" [Uhrmacher et al. 2006, attributed to Uhrmacher and Kuttler [2006]]. "Some models are better represented by changes in their structure" [Barros et al. 1994].

In a variable structure model, only active model components need to be loaded dynamically [Hu et al. 2005] and thus only those components need to be computed and hold in the memory of the target computer on which the model is eventually executed. However, the adaptation of the model structure during execution induces an additional overhead. Therefore, variable structures may not prove beneficial for all kinds of models in terms of computational costs in comparison with static structures. According to Hu et al. [2005], variable structures are particularly useful for simulating complex systems with a huge number of components. We can assume that the impact of variable structures on the computational costs is even bigger, if only a few components of such systems are active at the same time and structure changes happen infrequently.

Remark. In this section, when talking about components we refer to model components or submodels rather than the components as described in Section 5.1.3.

So using variable structures when modeling dynamic structure systems has an influence on the model development and the model execution. Whereas it is hard to objectively measure how convenient and natural variable structure modeling are without extensive user studies and suitable metrics, it is, relatively, easy to compare the computational costs of executing static structure models with the computational costs of executing behaviorally equivalent variable structure models. A first evaluation of the impact of a variable composition on the overall performance of a simulation study can be found in Deniz [2010], which indicates that with more complex models (i. e., an increasing number of components and events) the performance gain of using variable structures instead of static structures increases. Bae, Bae, Moon, and Kim [2016] also investigate the overhead of executing variable structure models; and conclude that by using proper algorithms the simulation execution time of variable structure models can be significantly reduced. However, if the computational costs of a simulation study are the major concern, more elaborate investigations are required to identify characteristics of a system of interest that indicate whether or not variable structures are beneficial. Here, we focus on modeling aspects rather than performance or computational costs.

6.2.1 Aspects of Variable Model Structures

So far, the notion of a variable model structure has been introduced in a rather abstract manner, i. e., a variable structure model is one that is able to change its structure [Zeigler & Praehofer 1990]. Based on the concept of (general) systems and system specifications as proposed by Zeigler et al. [2000, pp. 99–133], we now identify and describe the different aspects a variable model structure may cover, in more detail. We distinguish between a variable

[3] According to Zeigler [1987, p. 196], the term "variable structure model" was coined by Tuncer I. Ören in the mid-seventies.

structure at the level of coupled systems (system networks) and the level of basic systems, where coupled systems and basic systems refer to coupled models and atomic models in the DEVS realm, respectively.

Coupled Models

At the level of coupled models, structure variability may cover the following aspects:

- **Variable composition:** Capturing the variable composition of a system of interest (i. e., system components can be created, destroyed, or replaced), the composition of a coupled model of such a system can change accordingly, during model execution. Possible changes of the composition of a coupled model are the addition, deletion, or exchange of components of the coupled model (cf. Pawletta et al. [1996]), where a component can be an atomic model or, in case the system specification is closed-under-coupling, another coupled model. Especially for the exchange of components, the unambiguous distinction between different components is of importance. Assuming that model components are identified by identifiers (names), we define an exchange as replacing one component with another that has the same identifier as the component that is replaced. Thereby, an exchange can be reproduced by a deletion followed by an addition. When exchanging components with each other during model execution, we often want to copy at least some values of state variables from one component to the other. This implies a dependency between the actual states of components and structure transitions, as discussed by Pawletta et al. [1996].

- **Variable couplings:** In case the communication structure in a coupled model is specified and constrained explicitly by couplings (or connections) between the components of the coupled models, these couplings can be subject to changes during the execution of the coupled model. All couplings together define the communication structure. Only model components that are coupled can communicate and thus interact with each other. Note that sometimes the communication structure is considered to be a part of the composition (e. g., by Röhl and Uhrmacher [2008]). Here, we expressly distinguish between composition and communication structure. For this reason, we sometimes simply refer to the couplings or coupling scheme in a coupled model, when talking about the communication structure.

- **Variable interface:** A coupled model can have inputs and outputs. The set of admissible inputs and the set of admissible outputs together define the interface of the respective model. This interface may change. Hence, inputs accepted by the coupled model at a certain time during execution may not be accepted at another time. Similarly, a coupled model may not be able to produce all outputs at all times[4]. Often the inputs and outputs of a system of interest are not opaque and plain but structured according to certain input variables and output variables, respectively. These leads us to the notion of *ports*. Therefore, we also refer to *variable ports* (cf. Uhrmacher and Priami [2005] or Uhrmacher et al. [2006]) when talking about variable interfaces.

Figure 6.1 shows two "snapshots" of a simple coupled model whose composition and couplings are changing during execution. We call a structure change at the level of a coupled model also *network transition*. Since composition and communication structure come closest to our intuitive understanding of the structure of a model, both aspects are often the primary focus of a variable model structure or topology[5]. Additionally, we can expect that a variable composition, in principle, has the most significant impact on the computational costs of executing models (i. e., simulation).

[4] Note that there is a difference between a model that, in principle, can produce a certain output but practically does not and a model that cannot produce a certain output at all.

[5] Barros [2012, 2014] uses the term "topology" to refer to the structure of a model, particularly model compositions and networks. Herein, we use the term" model structure".

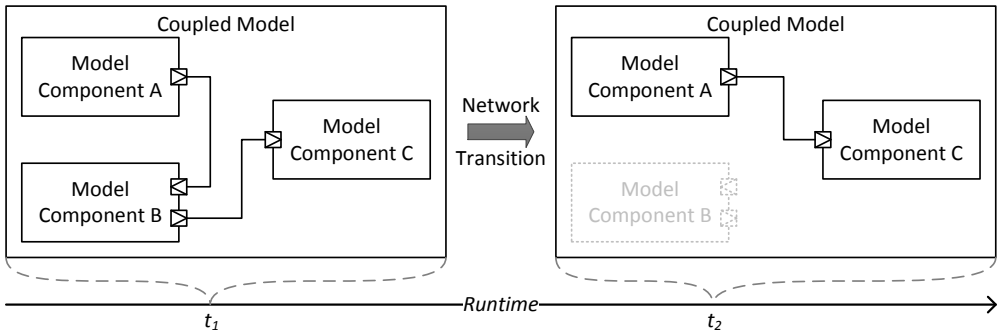

Figure 6.1: A simple coupled model that changes its composition during execution. At time t_2, model component "B" is removed from the composition and all couplings from and to this component become inconsistent and are deleted. In addition, a new coupling between model component "A" and "C" is established.

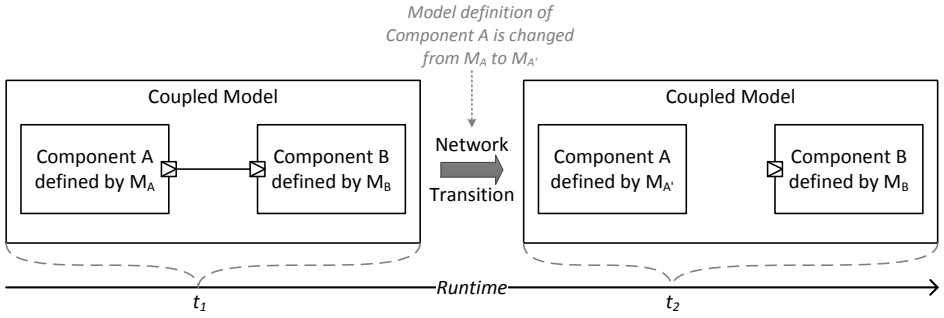

Figure 6.2: The coupled model changes its structure by changing the definition of component "A" and its coupling scheme. From an external viewpoint, the composition of the coupled model has not changed, i. e., at both instants component "A" and "B" are available. However, as the definition of component "A" has changed, so can its state space, interface, and characteristic functions. This perceptible structure variability at the level of atomic components is a direct result of structure variability at the level of coupled models.

The exchange of model components allows us to achieve structure variability at the level of atomic models implicitly or mimicked. Figure 6.2 illustrates the correlation between structure variability at the level of coupled models and the level of atomic models. From an external observer's perspective, the composition of the coupled model remains unchanged[6], whereas the component "A" seems to have a variable structure (see below). However, in Figure 6.2, component "A" is defined by two different models at time t_1 and time t_2; or both instants show different components with the same name.

Atomic Models

At the level of atomic models, structure variability may cover the following aspects:

- **Variable interface:** Similar to coupled models, the interface of an atomic model can change during model execution. In the case the interface of the model comprises ports, we also refer to variable ports.
- **Variable behavior patterns:** The behavior of an atomic model is defined by a set of characteristic functions, especially by one or more state transition functions. Although the

[6] At both depicted instants, the coupled model consists of two components: component "A" and component "B."

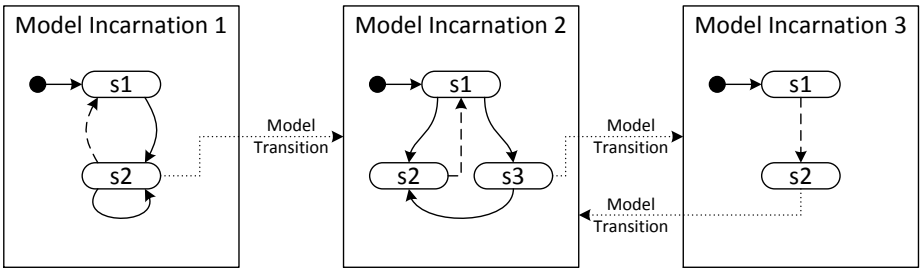

Figure 6.3: Different incarnations of the same atomic, variable structure model. The dotted arrows depict transitions between the incarnations and thus structure changes at the level of the atomic model (structural transitions). The solid and dashed arrows depict nonstructural transitions within the incarnations (here external and internal state transitions). The figure is adapted from Uhrmacher [2001], neglecting potential in- and outputs of the individual model incarnations.

outcome of these functions (function values) varies when executing the model (depending on the functions' arguments), the functions themselves remain unchanged. Variable behavior patterns refer to a change of these functions during model execution. Barros [2002, 2014] shows how such a change of the characteristic functions can be "emulated," by using indexed sets of functions and an index function that determines the currently active functions based on the current state. A behavior pattern can then be understood as the combination of the current incarnation of each characteristic function. Variable behavior patterns may reflect different roles a model can play, in different situations.

- **Variable state space:** During simulation, the state of a state-based model changes. However, its state space remains unchanged. Similar to a variable interface, an atomic model may, in principle, also have a variable state space, i. e., one that is changing during model execution. So each incarnation of a model may have its own state space and states that may be accessible at one instant during execution may not be accessible at another instant.

Especially the latter two aspects of structure variability at the level of atomic models lead to the idea of *model incarnations* that can change into each other, as discussed by Uhrmacher [2001]. The behavior of an atomic model is then defined by the behavior of its incarnations and the transitions between them (*structural transitions* or *model transitions*), where each incarnation is a model itself. Figure 6.3 illustrates the difference between structural transitions, i. e., transitions between distinct model incarnations, and nonstructural transitions, i. e., regular transitions such as state transitions that occur within a model incarnations.

Note that structure variability at the level of atomic models is more subtle and ambiguous than at the level of coupled models, as we can mimic variable behavior patterns by a suitable definition of model states and characteristic functions. For instance, we can extend the state of a model by certain "flags" or "indicators," which we then use to partition the state space and define the behavior of the model upon, similar to step or indicator functions. However, from a modeling point of view, it seems "easier" to model the behavior of a complex system of interest by a set of distinct, less complex models (model incarnations) that can change into each other—where each incarnation captures only some aspects of the behavior of the whole system—than to specify an overall, merged model of the system. So we are decomposing the behavior of a system within the temporal dimension in variable structure modeling.

In contrast to structure variability at the level of coupled models, the impact of structural variability at the level of atomic models on the computational costs of simulation studies is presumably negligible or even negative. If we are executing an atomic model that has a variable state space and variable behavior patterns, then still one incarnation of this

model is active at any time during model execution. Only if the complexity of the states of the different incarnations varies drastically, we may have an perceivable influence on the memory consumption. On the other hand, we introduce an additional overhead to the overall computational costs for switching between and initializing the different model incarnations. If we have to switch between model incarnations frequently, the computational costs of the model execution may increase. However, for reliable statements about the impact of structure variability at the level of atomic models on computation costs, further investigations and evaluations are required.

6.2.2 Structure Changes, Structure Transitions, and Structure Transition Functions

In the remainder of the thesis, we use the term "structure change" as a generalization for the different aspects of structure variability discussed in Section 6.2.1. Thereby, a structure change may refer to a change of the composition, interface, etc. "A variable structure changes a component-based system during runtime" [Hu et al. 2005]. As our focus is on computer simulation, runtime refers to the execution of a (variable structure) model, i. e., simulation. So regardless of the different levels and aspects of structure variability, structure changes occur during model execution [Steiniger & Uhrmacher 2013].

Corollary 1. *Changes of the model structure, i. e., structure changes, are a runtime phenomena, i. e., occur during simulation.*

Structure changes in a model can be triggered by a state change or input; or, more generally, by some sort of *event* that occurs during execution. Thus we cannot determine beforehand (i. e., before the model execution) which concrete structure changes will eventually take place at runtime. How and when the structure of a model is changing is often defined by dedicated structure transition functions, such as the network transition function of dynDEVS [Uhrmacher 2001] or the topology function of HFSS [Barros 2012, 2014]. From this viewpoint, a structure change corresponds to a transition of the structure, i. e., a *structure transition.*

Structure transition functions are part of a model's definition. Following Zeigler et al. [2000, p. 4], we can argue that structure transition functions belong to the model structure and thus can be changed as well. Nevertheless, the structure changes are part of the model's behavior. Thus, variable structure modeling is modeling rather than a configuration task, as traditional model composition in static structure models[7].

6.3 Related Work

The focus of my thesis is on formal models—especially those defined set-theoretically—that may not be directly executable on a computer[8]. Often, *executable models* need to be derived from formal models, before a simulation can take place. For this reason, we focus on modeling formalisms and languages that allow us to specify models with a variable structure formally (i. e., mathematically) rather than modeling approaches in which formal models refer to executable programs, written in a high-level programming language (such as C# or Java). In the latter, model entities are mapped onto objects of classes or onto templates of the respective programming language (cf. Zeigler [1990]). These classes and templates can be instantiated and destroyed on demand during runtime. Hence, structure changes can easily be achieved in such, implementation-centric, modeling approaches.

[7] In a hierarchical model with a static structure, the entire model composition and coupling scheme is just defined once, independent of events that can happen during the model execution.

[8] Still, formal models are executable at an abstract level, i. e., by some sort of execution semantics such as an abstract simulator.

In the early nineties, Zeigler and Praehofer [1990] and Barros et al. [1994] stated that "conventional modeling theory" focuses on representing changes at the level of a model's behavior, but gives little support for describing changes of the model's structure. In fact, several classic modeling formalisms e. g., plain Cellular automata [von Neumann 1966; Wolfram 1984], DEVS [Zeigler 1976, 1984], finite state machines [Harel 1987; Hopcroft et al. 2001], or Petri nets [Peterson 1981; Petri 1962] do not allow expressing structure changes explicitly.

Since the last decades, several classic modeling formalisms—those that did not support variable structures from the outset—have been extended in a way that structure changes became first-class abstractions, i. e., providing a support for variable structure modeling. In the following, we elaborate on respective extensions and adaptations of DEVS and P-DEVS.

6.3.1 Variable Structure Variants of DEVS

Since the mid-nineties Fernado J. Barros et al. introduced a number of variable structure variants and extension of DEVS and P-DEVS [Barros 1995a, 1995b, 1998, 2002; Barros et al. 1994]. All of these variants and extensions are based on similar ideas of how to specify variable structure models: structure changes are reflected at the level of coupled models (networks) and are carried out and controlled by a special atomic model component, the *network executive* (or *network controller*); one associated with each coupled model.

Barros et al. [1994] outline first ideas for a variable structure variant of classic DEVS, called Variable DEVS (V-DEVS), which serves as a blueprint for later variants and extensions introduced by Barros. Coupled models are replaced by *variable coupled models* whose composition and couplings can change. Each variable coupled model includes a special atomic model acting as a controller, which dictates structure changes for the coupled model. For this, the controller keeps composition and coupling information. Structure changes can only be triggered by internal or external transitions of the controller. Atomic models are defined just like in classic DEVS (cf. Zeigler et al. [2000, pp. 75–7]).

Barros [1995a] and Barros [1995b] introduce Dynamic Structure DEVS (DSDEVS), refining V-DEVS. A coupled model of DSDEVS is defined by an identifier of a *network executive* and a model definition of the executive. Like a controller in V-DEVS, the network executive is a special atomic model that controls and carries out structure changes, top-down. Each state of an executive comprises a definition of a coupled model of classic DEVS and other non-structural information. Changes of the structure-related state variables are automatically mapped onto changes of the network structure. Each component of the coupled model can be coupled to the network executive, allowing components to send events to the executive that can trigger structure changes. However, events that are sent to the executive cannot be sent to other components of the respective coupled model at the same time.

Barros [1997] introduces a modified and generalized version of DSDEVS, DSDE (Parallel Dynamic Structure DEVS), which is based on P-DEVS (see Section 4.2.1), instead of DEVS. The abstract simulator of DSDE is presented in Barros [1998]. Atomic DSDE models have a general state transition function that is invoked when the model has to perform an internal, external, or confluent state transition[9]. Like in DSDEVS, a coupled DSDE model consists of an identifier and a model definition of a network executive. In contrast to DSDEVS, all potential network structures, i. e., compositions and couplings, become a part of the defining tuple of the network executive and a dedicated structure function is responsible for changing the current network structure, based on the executive's state. Barros [2004] introduces another variable structure extension of P-DEVS, that is the Discrete Flow System Specification (DFSS), which is almost equivalent to DSDE, particularly with respect to specifying variable structure models.

[9] This general state transition function combines the three state transition functions (internal, external, and confluent) of atomic P-DEVS and is similar to the overall state transition function described by Zeigler et al. [2000, p. 155]

The Heterogeneous Flow System Specification (HFSS) [Barros 2002, 2003, 2012, 2014] is another extension of P-DEVS that allows specifying variable structure models. Moreover, HFSS facilitates the specification of continuous/discrete systems (hybrid systems) by using a concept called *sampling* and multivariate integration methods. However, in terms of a specifying variable model structure, HFSS is based on the same ideas as DSDE and DFSS.

A different approach to allow variable structure modeling based on DEVS was pursued by Adelinde M. Uhrmacher et al., resulting in a series of variable structure variants of DEVS and P-DEVS [Uhrmacher 2001; Uhrmacher et al. 2007, 2006]. In contrast to the top-down approach of Barros, these variants capture "the intrinsic reflective nature of variable structure models" [Uhrmacher 2001], in such a way that the models themselves, especially atomic models, are in control of changing their structure, bottom-up, and models are defined recursively. Moreover, these variants emphasize the ideas of model incarnations changing into each other, where each incarnation can be viewed as a conventional static model[10].

The first representative of these formalisms is Dynamic DEVS (dynDEVS), which is introduced by Uhrmacher [2001] and based on classic DEVS. Atomic dynDEVS models consists of sets of inputs and outputs, an initial model incarnation, and a set of possible model incarnations at least containing the initial model incarnation. Each model incarnation is similar to a classic atomic DEVS model, without the sets of inputs and outputs, which are the same for each model incarnation. In addition, a model incarnation has an initial state and a model transition function, which determines, based on the model state, the active model incarnation. Similarly, a coupled dynDEVS model (network) is defined by sets of network inputs and outputs, an initial network incarnation, and a set of possible network incarnations at least containing the initial network incarnations. Each network incarnation is defined similarly to a coupled DEVS model. However, a network incarnation has no inputs or outputs but a network transition function, which determines a network incarnation for the current state of all available components of the coupled dynDEVS model. For this, the network transition function resolves possible conflicts[11] regarding a change of the composition and ensures, along with the select function of a network incarnation, a deterministic change of the model/network structure.

Himmelspach [2007, pp. 89–97] and Himmelspach and Röhl [2009, pp. 514–20] introduce a variant of dynDEVS: Parallel Dynamic DEVS (PdynDEVS), which is based on P-DEVS instead of classic DEVS. In contrast to dynDEVS, PdynDEVS introduces two kinds of explicit structure change requests (structure change events): incoming requests (received by a model) and outgoing requests (sent to other models). These requests are separate from regular in- and outputs. State transitions of an atomic PdynDEVS model can create both kinds of requests. The model transition function of atomic PdynDEVS models operates on incoming requests, not on the model state as in dynDEVS. Similarly, the network transition function of a coupled PdynDEVS model considers incoming requests when determining the next incarnation of the network. Structure change requests may not lead to structure changes, since the requests can be "discarded" by the corresponding transition function. Due to its definition at the level of structure systems (see Section 4.2.2), PdynDEVS makes use of port-to-port couplings.

Uhrmacher et al. [2010, pp. 142–46] continue and revise the ideas of PdynDEVS and introduce DynPDEVS (Dynamic Parallel DEVS). In DynPDEVS, incoming and outgoing structure change requests (structure change events) extend the interfaces of atomic and coupled models. The model transition function of an atomic DynPDEVS model determines its next incarnation based on the model's state and incoming change requests. Moreover, each atomic model has an additional output function creating outgoing change requests based on the

[10] see Figure 6.3

[11] For instance, one component can initiate the removal of a another component, where at the same time the initiating component shall be removed on behalf of a third component.

current state. The network transition function of a coupled DynPDEVS model determines the next network incarnation based on the states of all available components and, unlike in dynDEVS, incoming change requests, received from the coupled model's components and parent. Like the atomic model, the coupled DynPDEVS model has a *structural output function* that creates outgoing change events.

Uhrmacher et al. [2006] and [Uhrmacher et al. 2010, pp. 146–50] present another parallel variant of dynDEVS, called ρ-DEVS, which is similar to DynPDEVS. However, in contrast to dynDEVS and DynPDEVS, ρ-DEVS supports variable interfaces and ports (as it is defined at the level of structured systems). Thereby, each incarnation of an atomic and coupled ρ-DEVS model has its own in- and output interface, which can be subject to model and network transitions. To define the couplings between model components with variable ports consistently, Uhrmacher et al. [2006] introduce the concept of an intensional, directed coupling mechanism, called *multi-couplings*, based on port names. A multi-coupling relates a set of pairs of component and port names (sources) with another set of pairs of component and port names (targets). In addition, a multi-coupling contains a special select function[12] that determines which target components will eventually receive inputs and allows realizing other distribution strategies than a simple broadcast. The availability of ports during simulation implies the existence of concrete, directed port-to-port couplings. This means that each available source port specified in a multi-coupling is coupled to each available target port of components that are returned by the select function. Similar to DynPDEVS, the interfaces of atomic and coupled ρ-DEVS models also consist of incoming and outgoing structure change events. In contrast to ports, the sets of structure change events are static, i. e., are not changed by model nor network transitions.

6.3.2 Classification and Discussion

As shown above, several modeling formalisms and languages addressing variable model structures exist, in the realm of DEVS. From the point of view of DEVS, Barros [2014] distinguishes between two families of modeling formalisms for representing dynamic structure systems: (i) centralized and (ii) distributed. Herein, we distinguish between three general approaches that allow us to model dynamic structure systems, leading to the following three classes:

1. Top-down (or centralized),
2. Bottom-up (or distributed), and
3. Implementation-centric.

Members of the first two classes provide the means to capture structure variability formally, as intrinsic part of the modeling formalism itself. Members of the third class, on the other hand, extend the simulation or execution environment by structure change operations, which can then be used by the modeler when creating executable models of dynamic structure systems for the respective environment (e. g., DEVSJAVA [ACIMS 2009], DEVS++ [Zeigler, Moon, Kim, & Kim 1996], or JAMES II [Himmelspach & Uhrmacher 2009]). Examples for implementation-centric approaches are Hu et al. [2005] or Deniz [2010]; Deniz, Alpdemir, Kara, and Oğuztüzün [2012]; Deniz, Kara, Alpdemir, and Oğuztüzün [2009].

In top-down approaches, each network (coupled model) is associated with a special, central model component: the network controller or network executive, which is responsible for carrying out structure changes and maintaining the structural consistency of the network. As an atomic model, the network executive has a state of its own, based upon which the network

[12] This select function is not to be confused with the tie-breaking function known from variants of classic DEVS.

executive takes decisions about the structure[13]. Since the network executive is coupled with all other model components of the respective network, these components can send events to the network executive and thus trigger structure changes. Top-down approaches do not allow carrying out structure changes at the level of atomic models explicitly, because only the network executive is in charge of structure maintenance, whereas the regular atomic models remain static. Especially in natural systems, we often do not find a central component that is control of the structure, instead the "topology [structure] emerges without any dedicated entity being responsible for topology [structure] maintenance" [Barros 2014]. For this reason, top-down approaches may seem less natural than bottom-up or distributed approaches.

In bottom-up approaches, atomic models can change their own structure (in terms of model transitions[14]) and initiate structure changes at the level of coupled models (in terms of network transitions). Similar to top-down approaches, structure changes at the level of coupled model are carried out by the respective coupled model. However, coupled models have no state of their own based on which they can make decisions regarding structure changes. Instead, a coupled model (i. e., the network transition function) takes the states of all its components into account when it is about to change its structure. Recollecting Section 5.1.2, one could think that this is a violation of modularity (cf. Barros [2014]). However, Zeigler et al. [2000, pp. 149–62] describe non-modular modeling formalisms as those that employ non-modular couplings, where components directly access the states of their siblings (i. e., the components they coupled with).

Even if a modeling formalism allows variable structure modeling, the formalism does not necessarily support all of the aspects of a variable model structure discussed in Section 6.2.1. In fact, just a few of the considered DEVS-based modeling formalisms support variable interfaces, whereas a variable composition and variable couplings are supported by all of the modeling formalisms. Also, most of the discussed modeling formalisms do not support variable behavior patterns and state spaces at the level of atomic models explicitly; more specifically, top-down approaches generally lack of a support of these aspects of structure variability. However, as mentioned in Section 6.2.1, structural variability at the level of atomic models can be achieved implicitly by replacing model components with each other. Table 6.1 compares the modeling formalisms with respect to the support of the different aspects and their membership to one of the three classes mentioned earlier.

Table 6.1 shows that different modeling formalisms support the same aspects of structure variability (e. g., DSDE, DFSS, and HFSS). However, the way how these aspects are supported can differ. Moreover, the modeling formalisms may have additional features beyond the support of structure variability. Here, the focus is merely on the latter.

Allowing atomic models to change their interfaces requires a different approach to specify couplings at the level of coupled models consistently. For instance, a port that is part of a certain coupling can become unavailable without the knowledge of the respective coupled model. This is the reason why just a few DEVS-based modeling formalisms support variable interfaces explicitly (e. g., ρ-DEVS).

Supporting the different aspects of structure variability, as discussed in Section 6.2.1, is just one prerequisite for creating variable structure models. Allowing models to actually reason about their structure and the structure their are part of is another, important facet of variable structure modeling. In strictly modular approaches such as classic DEVS, where the knowledge of a module (i. e., model component) ends at its boundary (interface), structural reasoning is hampered. While a model component can still reason about and change its own structure, it cannot reason about the structure it is embedded into without further ado, as the model component per se has no notion about its environment. For instance, a model

[13] Barros [2014] distinguishes between decision making and decision enforcement when talking about modeling dynamic structure systems. The former relates to gathering information and making decisions (e. g., about structure changes), whereas the latter relates to enforcing decisions (e. g., structure changes).

[14] see Figure 6.3

Table 6.1: Comparison of DEVS-based modeling formalisms regarding their support of different aspects of structure variability.

modeling formalisms	Aspects of structure variability: Level of atomic models: state space	behavior patterns	interface	Level of coupled models: composition	couplings	interface
V-DEVS	✗	✗	✗	✓	✓	✗
DSDEVS	✗	✗	✗	✓	✓	(✗)
DSDE	✗	✗	✗	✓	✓	✗
DFSS	✗	✗	✗	✓	✓	✗
HFSS	✗	✗	✗	✓	✓	✗
dynDEVS	✓	✓	✗	✓	✓	✗
PdynDEVS, DynPDEVS	✓	✓	✗	✓	✓	✗
ρ-DEVS	✓	✓	✓	✓	✓	✓
D-HFSS	✗	(✓)	✗	✓	✓	✗

component may trigger the removal of another component that actually does not exist.

To improve and ease structural reasoning for the modeler at this point, each (atomic) model needs to have and maintain an internal model of the structural context (environment or world) in which the model exists; leading to endomorphic models as discussed by Zeigler [1990][15]. This internal model needs to be updated whenever structure changes occur to keep it in synchrony with the actual model structure. The information about the environment of a model can be provided as an input triggering an update of the internal model of the environment. All DEVS-based variable structure variants discussed in this chapter support such an approach; however, the modeler needs to implement it manually, which can be tedious and error-prone. Here, additional support would be desirable from the modeler's perspective.

6.4 Summary

The beginning of this chapter characterizes dynamic structure systems, i. e., systems that can change their structure over time. Then the chapter describes variable structure models that allow the modeler to capture structure changes of a system of interest explicitly, as a central part of a model's dynamics. In this context, the different aspects of structure variability at the level of atomic and coupled models are illuminated. The chapter concludes with a survey of modeling approaches that allow variable structures, with a focus on variable structure variants of DEVS. In addition, the chapter outlines a way to classify the different variants of DEVS.

[15] "Endomorphism is a hoary mathematical concept which refers to the existence of a homomorphism from an object to a sub-object within it, the part (sub-object) then being a model of the whole" [Zeigler 1990, p. 16]

Part II

Concept and Implementation

7 Composition of Variable Structure Models

> Failure is the key to success; each mistake teaches us something.
>
> Morihei Ueshiba

This chapter recollects and elaborates on peculiarities of conventional component-based modeling[1] and variable structure modeling[2] and discusses commonalities and discrepancies between the both. In addition, the chapter examines ideas and approaches on how to bring both paradigms together, at least to some extent.

This chapter recaps and takes up the discussion from Section 1.1. It is mainly based on ideas presented in:

Steiniger, A. and Uhrmacher, A. M. (2013). "Composing Variable Structure Models: A Revision of COMO." In *Proceedings of the 3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications* (SIMULTECH 2013). pp. 286–93.

These ideas are revisited and extended in the following.

[1] as described in Chapter 5
[2] as described in Chapter 6

7.1 Commonalities and Differences

Existing formalism-independent component-based modeling approaches, such as COMO [Röhl 2008] or CODES [Szabo & Teo 2007], introduce an additional specification layer at which the assembly and interaction of prefabricated, off-the-shelf components can be described using a special composition methodology; e. g., a composition formalism that is usually different from the underlying modeling formalisms used to specify the behavior of the individual components. Often, these approaches (e. g., COMO) require the explicit definition of component interfaces, based upon which a composition is then defined (*interface-based composition*). Thereby the implementation of a component, defining the component's *internal behavior*, has to adhere to the component's interface definition. Since components are accessed and composed via their well-defined interfaces, we can keep composition descriptions separate from the actual implementations of the components (cf. Verbraeck [2004] or Röhl and Uhrmacher [2008]). In the following, we refer to these kind of component-based modeling as *traditional composition* (or *conventional composition*).

Variable structure modeling allows the modeler to explicitly describe a system with a variable structure (see Definition 6.1.1) by structure changes on top of behavioral changes, where structure changes become first-order abstractions in the respective variable structure models. In contrast to traditional composition, modeling variable structures is not done at an additional specification layer, but is an integral part of the modeling formalism that is used to create simulation models of dynamic structure systems. Often, the respective modeling formalism is equipped with one or more structure change functions (see Section 6.2.2). In the following, we refer to variable structure modeling as *temporal composition*, since it describes the composition of a model at the temporal dimension and how the composition is changing over time (based on certain events).

Section 1.1 has already outlined that both traditional composition and temporal composition deal with specifying the structure of a model or model composition, by providing certain means to do so. In traditional composition, we specify a system model as an assembly of distinct, self-contained components interacting with each other via predefined interfaces, where each component encapsulates one aspect of the behavior of the overall system. As we mentioned above, the internal behavior of the involved components (i. e., their implementations) is kept separate from the interface and composition descriptions. The composition itself can be done just based on the components' interfaces. However, for the execution we also need to know the initial states and internal behavior of the corresponding components. In variable structure modeling, we specify possible incarnations of a model with a variable structure and how and when one incarnation changes into another. Each incarnation can be considered as a static snapshot of the dynamic structure system, consisting of model components that interact with each other via couplings. So both types of composition, describe the composition and communication structure of an assembly of smaller model units, i. e., components.

Traditional composition, i. e., constructing syntactically (and semantically) correct simulation models from prefabricated, off-the-shelf components, assumes a static model structure, i. e., the model structure is not changing during execution. Or in other words, once a composition is assembled and configured, its structure remains unchanged. Furthermore, traditional composition is done at configuration time [Petty & Weisel 2003a], before the execution. So regardless of what happens during model execution, a composition methodology assures, if applied correctly, at least syntactic correctness (*syntactic composability*) of the composed model independent of the implementations of the the involved components[3]. To do so, the components need to adhere to interface definitions (cf. the *refinement relations* as discussed by Röhl and Uhrmacher [2008]). When using a composition approach such as COMO, composition descriptions as well as the implementations of the involved components need to

[3] cf. Section 5.2

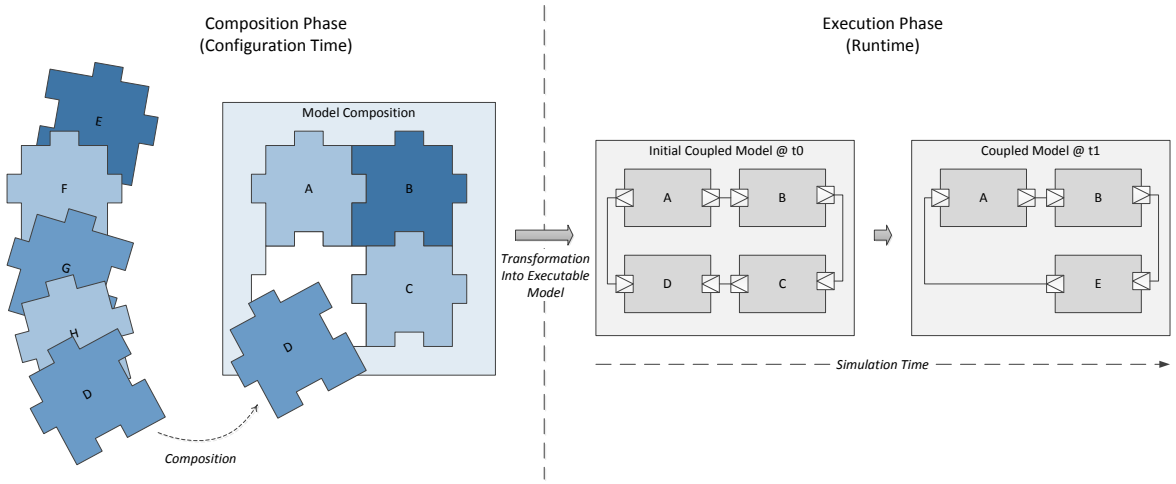

Figure 7.1: Relationship between traditional composition that is done before a model is executed and a corresponding simulation model whose internal structure is changing during execution.

be transformed into a platform-dependent target formalism[4] for the execution of a model composition, i. e., an executable simulation model has to be derived or synthesized. How the states of the components are evolving during execution depends on their implementations and cannot be determined at configuration time. In fact, the states of the components are transparent to the composition layer, they are part of the components' implementations.

In contrast, as Corollary 1 already stated, variable structures and structure changes are a runtime phenomena. For this reason, we cannot determine which and when structure changes take place at configuration time, when assembling a model composition. Nonetheless, the modeler has to define beforehand, which structure changes can, in principle, take place and in which situations, as one essential part of a model's behavior. Therefore, variable structure modeling is more like traditional *behavior modeling*[5] than simply configuring a model before the beginning of simulation; just like the name indicates. The syntactic consistency of a variable structure model is assured by the actual modeling formalism, which is used, by providing the means to create syntactically consistent models[6]. Semantic consistency, on the other hand, has to be ensured by the modeler. Figure 7.1 illustrates the relationship between the configuration phase (composition) and runtime phase (execution). So although both traditional composition and temporal composition describe the structure of a model, they do so at different times: at configuration time (traditional composition) and during execution (temporal composition).

7.2 Combination and Contradiction

Section 1.1 and Chapter 6 state that many complex systems of interest exhibit a variable structure, which we want to capture when modeling these systems. Therefore, it just seems to be natural to ask: if and how we can bring traditional composition and temporal composition together to benefit from the advantages of both? Ending up with self-contained components

[4] In the case of COMO, the target formalism is P-DEVS (see Section 4.2.1) and the platform on which the derived simulation model is eventually executed is JAMES II [Himmelspach & Uhrmacher 2007].

[5] When talking about behavior modeling we refer to specifying how the state and potential outputs of a model are changing based on certain events.

[6] Of course, a modeler can also create inconsistent models either intentionally or by mistake.

whose internal structure is not time invariant and that can be reused in different simulation studies and contexts, also by third-parties.

When we were thinking in more detail about components encapsulating variable structure models, whose interfaces may in addition be variable as well, the following questions came into our minds:

- What aspects of a variable structure, as described in Section 6.2.1, can and should be supported by a composition methodology?
- How and to which extent can we reflect variable structures and interfaces in a composition methodology?
- What are the impacts of structure variability on the general notion of composability, i. e., assuring the correctness of a model by composition before the model is executed? Especially since composability is typically defined without making any assumptions on the dynamics of the model structure (cf. Petty and Weisel [2003a]).
- Can we, at all, assure correctness beyond the initial composition of a variable structure model in the configuration phase?
- In the case that encapsulated models have variable interfaces, how can we define connections between the corresponding components without knowledge about the availability of the respective interaction points (ports) during execution?
- Is a variable interface well-defined according to traditional composition?
- Can we describe structure changes that occur during model execution independent of the implementations of the composed components?
- Should the executable simulation model derived from a composition description and the implementations of the composed components be a variable structure model itself?
- Since variable structure modeling often is different from traditional static structure modeling, can we simply reuse components that were designed with a variable structure in mind in another context?

In the remainder of the thesis, we will give answers to the above questions. However, for some of them, there is more than one satisfying answer.

Section 7.1 already implies a fundamental contradiction between traditional and temporal composition. In traditional composition, we want to keep interface and composition descriptions separate from the implementations of the involved components. Components also often encapsulate atomic models rather than coupled models, since coupled models are simple containers without a behavior that can be directly expressed as composite components in the composition methodology. For the execution of such a model composition, coupled models are then automatically synthesized from the composition descriptions, i. e., composite components (cf. Röhl [2008]; Röhl and Uhrmacher [2008]). This is straightforward when not facing variable structures or interfaces. In variable structure models, however, coupled models are more than just plain containers (see Section 6.2.1). They also describe how their components and the interaction between them is changing, based on certain events and sometimes also on the coupled models' states. Synthesizing coupled models with such a behavior from a formalism-independent composition descriptions requires the incorporation of structure change functions into the composition descriptions. Components would then not only and simply be wired together, but the modeler would also need to specify influences between these components and impacts on the composition, which is not done in traditional composition. In fact, we would end up with re-specifying the structure variability of the model at the composition layer. This, at least in the case of state induced structure changes[7], contradicts the separation between composition descriptions and component implementations.

[7] For instance, as in the case of the modeling formalism DSDE [Barros 1997]

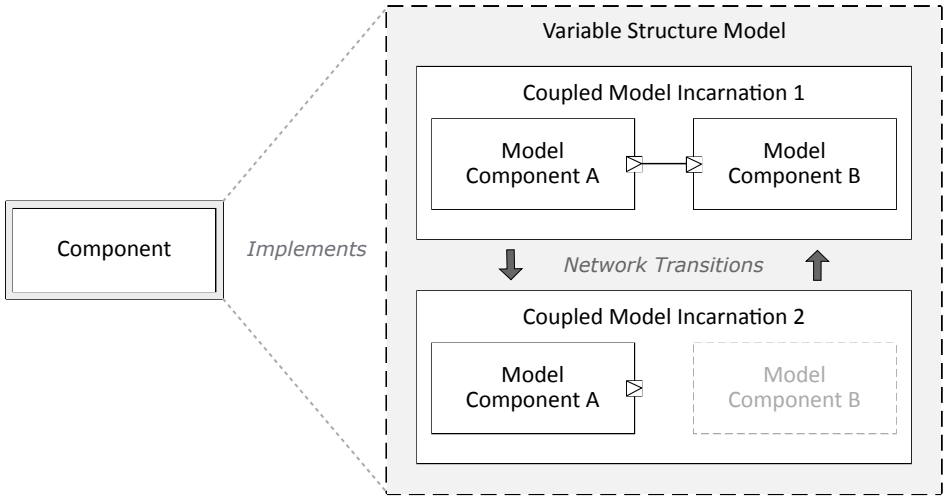

Figure 7.2: A component that is implemented by a simple variable structure model, which only consists of two incarnations; for simplicity. Traditionally, components are implemented by static structure models whose structure is not changing (i. e., there is only one incarnation of the coupled model).

In general, since the interplay between the coupled model and its components in variable structure modeling is more intricate than in the case of static structure models, we need to ask ourselves the following questions:

- Can we reuse variable coupled models without their components?
- Can we reuse the components of a variable coupled model without the coupled model itself?

In this context, have in mind that the components in top-down approaches for variable structure modeling are often defined as static structure models (i. e., without yielding any structure variability), such as in Barros [1995a] or Barros [2004]. This would, in principle, allow us to reuse these components, even in a static context. However, due to the reasons listed above it often, in our opinion, makes no sense to tear variable coupled models and their components apart. A variable coupled model cannot simply be used without components. Instead, we may want to use the overall variable structure model, including behavioral and structural dynamics, as a self-contained component.

At this point we could assume that we cannot combine traditional and temporal composition at all. However, as we show in the following, their exist some approaches to bring both types of composition together, at least to some extent.

7.3 Hiding Structure Variability

Although it may seem to be odd at the first sight, one simple approach to combine traditional and temporal composition, especially when we do not have to consider variable interfaces, is to hide structural variability and complexity within components. This complies with the fundamental principle that a component is hiding its internal structure, as defined by Verbraeck [2004]. In other words, a component is now implemented by a variable structure model instead of a static structure model. This component can encapsulate a variable atomic model (see Section 6.2.1) or a variable coupled model (variable structure network, see Section 6.2.1) including its components. Figure 7.2 shows a component that is implemented by a variable structure model instead of a static structure model.

The actual composition of components, regardless of whether they encapsulate static or variable structure models, remains static. So a possible structural variability of components

is hidden at the composition layer. No further adaptations of the composition methodology are required. We only need to make sure that there exists a transformation from the source formalism, which is used to implement the components, into the target formalism, which is used to create executable simulation models from the composition descriptions and the component implementations. Even if the target formalism does not support variable structures, we still have all the benefits of variable structure modeling, as listed in Section 6.2, when "implementing" the components[8]. Keep in mind, that the transformation into the target formalism is done automatically by the composition framework, i. e., is not the responsibility of the modeler. However, since we are restricted to execute static structures, we may loose potential performance gains when executing variable structures (cf. Deniz [2010]).

Since only the interface of a component is know at the composition layer, structure changes within components with a variable structure can only be triggered by interaction points declared by the interface (i. e., inputs) or by the flow of time, as known from top-down approaches discussed in Section 6.3, in which dedicated structure change events are propagated through a model hierarchy. Reflective approaches to express structure changes, in the spirit of, e. g., Uhrmacher [2001], cannot be achieved by hiding structure variability within components.

7.4 Supersets, Loose Connections, and the Revision of COMO

Our first approach to combine traditional and temporal composition, which is more extensive than the one discussed in the previous section, resulted in the extension of the composition and analysis framework COMO and its underlying composition methodology, as presented by Röhl [2008] and Röhl and Uhrmacher [2008]. This approach was published in Steiniger and Uhrmacher [2013]. The basic ideas presented therein were, among other things, the use of:

- supersets (as defined in Section A.1.1) and
- an intensional coupling mechanism, called *loose connections*.

We showed how both concepts can help us to (i) incorporate structure variability, including variable interfaces, at the composition layer to some extent and (ii) allow us to make statements about the consistency beyond the initial configuration (at configuration time). Furthermore, we changed the target formalism, which is used by COMO to come up with executable simulation models, from P-DEVS[9], which is limited to static model structures, to the first version of ML-DEVS, as presented by Uhrmacher et al. [2007]. The latter of which allows variable structures. Hence and in contrast to Section 7.3, a composition is transformed into a model that can, in principle, change its structure during execution (simulation). This, however, does not necessarily mean that we now can express variable structures at the composition layer, without further ado and to full extent.

Before a composition is transformed into an executable simulation model, components are instantiated and configured according to a given parameterization, leading in component instances. A single component can serve as a blueprint for an arbitrary number of component instances. These component instances are eventually translated into the target formalism by COMO (see Figure 7.3).

In the following, we discuss the general ideas presented in and insights gained from Steiniger and Uhrmacher [2013] in more detail. However, as the focus of this thesis is on the concepts presented in Chapter 8, we are not going into too much details and foregoing the formal definitions presented in the paper.

[8] Section 6.2 indicates that a variable structure model can be represented by a behaviorally equivalent static structure model.

[9] see Section 4.2.1

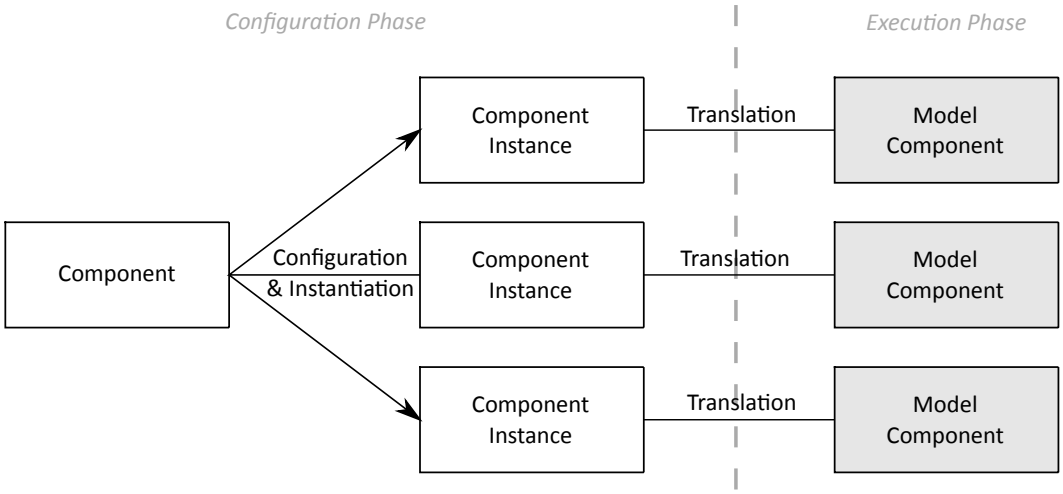

Figure 7.3: A component description serves as a blueprint for component instances in COMO, like classes in object-oriented programming language. Before an executable simulation model is derived, a component is instantiated and configured, according to a given parameterization. Finally, each component instance is translated into a component of the executable simulation model in the target formalism.

7.4.1 Description of Variable Interfaces

As noted in Chapter 5 and Section 7.1, well-defined interfaces play an important role in traditional composition approaches; leading to an interface-based composition. They serve as central contracts between components and their surroundings and allow hiding and abstracting from implementation details of the components. Therefore, interface descriptions contain all information that are required for composing components and checking the consistency of the resulting compositions, at least when dealing with static model structures.

We call the different information that is declared by an interface *attributes*. Beside less obvious attributes such as *parameters* via which a component can be customized, *ports* are of crucial importance for creating compositions by assembling and connecting components. Ports are a well-known concept in the realm of modeling and simulation for describing the inputs and outputs of a model or model component. These ports serve as communication points through which a component can send outputs and receive inputs to and from other components, respectively.

Variable ports refer to ports that can change their availability during time, e. g., ports that are available at a certain time may not be available at another time. A variable or dynamic interface is then one whose ports are variable. At this point, we are not considering other interface attributes, such as parameters, to be variable. The concrete incarnation of an interface, i. e., the availability of its ports, depends on the internal state of the corresponding component, the flow of time, and potential inputs.

When thinking about variable interfaces in the context of traditional composition the question arise, whether or not a variable interface is well-defined or well-specified according to Verbraeck [2004]? In Steiniger and Uhrmacher [2013], we argued that if we neglect the possibility to create new and arbitrary ports during simulation, we can assume that the superset of the variable ports, i. e., all ports a component can in principle exhibit during its lifetime, can be known before the execution. During model execution, ports from this superset can be selected or become available, whereas others ports can be deselected or become unavailable. Regardless of the actual availability of the ports, the superset of the ports remains unchanged and can be defined. So we can use this superset of ports to define the interface of a component encapsulating a model with a variable interface. The specification of how and when

ports are changing are part of its internal structure and is not reflected at the composition layer. This does not violate the notion of a well-defined interface, in accordance with Verbraeck and Valentin [2008], since all possible communication capabilities of the respective component are part of its interface. However, to execute the component properly, we need to extend its interface so that it also declares the initially available ports. This information can also be considered as a part of the initial state of the respective component, which needs to be known as well when we want to execute the component.

From the outside, the component can be viewed as one with a static interface that comprises all the potential ports that can become available at one point or another. Within the model execution, the execution engine can make sure that the model encapsulated by the surrogating component only receives inputs that are sent to currently available ports. In turn, the execution engine can make sure that only outputs of available ports are forwarded to other components in a composition.

In Steiniger and Uhrmacher [2013], we had an extensional definition of supersets of ports in mind, in which each port is listed or enumerated individually (cf. Section 3.2). This approach, as already indicated by Section 3.2, can be rather tedious and inflexible. Alternatively, a superset of ports can be defined intensionally (cf. Section 3.3). In other words, we define the members of this superset based on shared attributes, such as names or value ranges. For this, we could make use of the set-builder notation and predicates. Chapter 8 gives more details on this matter.

Either way, although an interface definition that uses supersets of ports (defined extensionally or intensionally) to declare its communication capabilities can be viewed as static, there is a difference to static interfaces. Variable ports have a direct impact on the definition of the communication structure in an assembly of components (i. e., model composition) and, thus, the ability of checking the correctness by composition at configuration time. We just know for sure which ports are available initially.

7.4.2 Description of Variable Communication Structures

Structural variability at the level of the communication structure includes two major aspects:

- variable communication channels as a result of variable interfaces (to maintain structural consistency) and
- communication channels that can change on their own.

In Steiniger and Uhrmacher [2013], we were focusing on the former aspect, as a result of dealing with variable interfaces. For this, we introduced a new way of specifying communication channels at the composition layer intensionally, by using so-called *loose connections*. These adopt and extend the idea of multi-couplings as described in Uhrmacher et al. [2007] and Steiniger et al. [2012]. At its core, a loose connection can "encode" an arbitrary number of communication channels between two components that are specified as pairs of names, where the first name corresponds to the name of the source port and the second name corresponds to the name of the target port, based upon the interfaces of the respective components. When deriving an executable simulation model, loose connections are transformed into multi-couplings, which ultimately need to be resolved into concrete couplings during execution. Hence, to make use of such an intensional coupling mechanism, the underlying target formalism has to be equipped with an equally expressive coupling mechanism, into which loose connections can be translated.

As with ports, we need to know the initially available loose connections, for a proper execution. However, we can still specify the communication structure consistently by a static set of loose connections that is not changing during execution. This is possible because loose connections are translated into concrete couplings ad hoc, while discarding inconsistent concrete couplings. Chapter 8 provides more information on this matter.

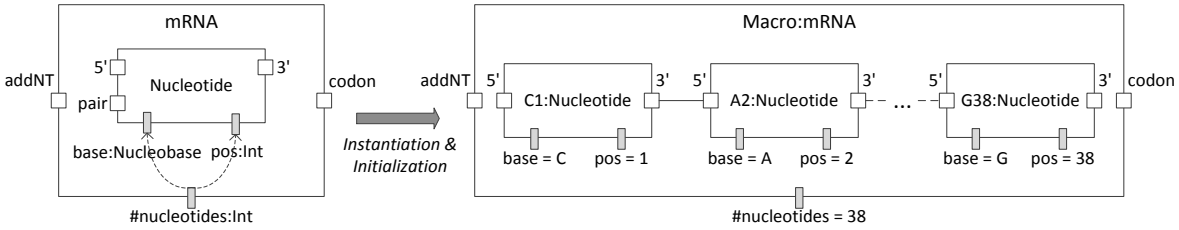

Figure 7.4: The extension of COMO, as described by Steiniger and Uhrmacher [2013], allows us to derive a composition as depicted on the right side from the a component, interface, and composition description as depicted on the right side by instantiation and configuration. The communication structure on the right side, connecting the 3' site of one nucleotide with the 5' site of the subsequent nucleotide, is specified by a single loose connection and works independently of the concrete number of nucleotides that should be instantiated (denoted by "#nucleotides").

Steiniger and Uhrmacher [2013] demonstrate how a single loose connection can be used to specify an arbitrary complex coupling scheme, such as the bi-directional bindings between two binding sites of nucleotides, organic molecules that form nucleic acids, in the mRNA (messenger ribonucleic acid). Figure 7.4 illustrates the example.

7.4.3 Description of Variable Compositions

One of the most obvious aspects of a variable model structure and the model structure in general, is the actual composition of a complex model (see Section 6.2.1), which specifies the constituent parts (i. e., components) of the model. In traditional composition, we usually distinguish between two types of components: atomic and composite. The latter of which are composed of components themselves, allowing the modeler to (i) express is-part-of-relationships between components at different levels of abstraction or detail and (ii) create component hierarchies of arbitrary depths, since a component of a composite component can be a composite component itself.

Similar to variable ports, we can often assume that the superset of all potential components of a composite component can be known beforehand and remains unchanged, regardless of the availability of the individual components during execution.

In Steiniger and Uhrmacher [2013], we, again, had an extensional definition in mind, when talking about supersets of components of composite components, where the individual components become available or unavailable at one point or another during execution, as depicted in Figure 6.1. Herein, we also argue for an intentional definition of supersets of components constraining the components that can be part of a composite component. This is less tedious and more flexible than enumerating all possibly available components extensionally.

As with variable ports, regardless of the actual definition of a superset of potential components, we also need to know the components that should be available initially for a proper execution of the corresponding composite component. In the case of a static composition, the set of the initially available components corresponds to the superset of all potential components, because all of them are available from the beginning and their availability does not change during execution. Whereas, in the case of a variable composition, the set of the initially available components can be a proper subset of the superset of all potential components; but does not necessarily have to be. In any case, the following relation holds:

$$\text{set of initially available components} \subseteq \text{superset of all potential components}.$$

Both the superset of potential components and the set of initially available components need to become a part of the description of a composite component, so we can make use of this information, when assembling and configuring components.

7.4.4 Correctness and Composability

As mentioned in Section 5.2, allowing to check the composability, at least at a syntactic level, at configuration time, is another important aspect of traditional composition. In other words, once an executable model is derived from a composition description, we can assume that this model is at least syntactically correct.

When dealing with variable interfaces and structures, we only know the initially available components, connections, and ports for sure. Thus, one could think, that we can only check the consistency of the initial state at configuration time. However, since we also know the supersets of potential components, connections, and ports, which can become available or unavailable during execution, we can check the consistency of all possible couplings which can be created. In Steiniger and Uhrmacher [2013], we adapt the notions of well-formed connections and complete component descriptions as introduced by Röhl and Uhrmacher [2008] by incorporating the corresponding supersets. As we will see later, some kinds of inconsistencies with respect to variable ports and couplings cannot occur, if the mechanism that translates intensional couplings into concrete model couplings during execution assures correctness by construction, by preventing inconsistent couplings from being established during execution. More details can be found in Chapter 8 and Chapter 9.

Although the component implementations are decoupled from the component interfaces and descriptions, due to the separation between component implementation and component description, both have to relate to each other. Therefore, the composition framework has to make sure that an implementation adheres to a respective interface and component description and vice versa. Röhl and Uhrmacher [2008] are using the term "refinement" for this bilateral relationship. More specifically, we have to make sure that all ports declared in an interface are actually part of the encapsulated model. In terms of models with a variable interface, we also have to make sure that all ports that can be exhibited by the model are part of the superset of ports of the corresponding component. In turn, all ports that are used in the model implementation have to be reflected in the interface description of the respective component. In the case of variable interfaces, we have to make sure that the model is not creating new port during execution that are not part of the interface. Accordingly, Steiniger and Uhrmacher [2013] modifies both refinement relations to work with our adaptations of COMO and its underlying composition descriptions.

7.5 Summary

This chapter motivates the appeal of traditional component-based and variable structure modeling approaches when modeling complex systems that are characterized by a complex composition and a variable structure. Afterward, the chapter identifies fundamental differences between both types of modeling, which make a combination of them rather difficult and have implications on aspects such as composability, especially syntactic composability. In the following, two general approaches are presented that show how these differences can be overcome, at least to some extent. The second approach is the revision of the composition and analysis framework COMO. In a nutshell, both approaches rely on the use of supersets and intensional (coupling) definitions, as described in Chapter 3. Using supersets of ports and components as well as intensional coupling definitions allows us to make statements about the composability of a variable composition, if the involved components adhere to their interface descriptions.

Chapter 8 revives and extends the ideas presented in this chapter.

8 Interfaces, Interface Instances, and Intensional Couplings

> I think complexity is mostly sort of crummy stuff that is there because it's too expensive to change the interface.
>
> Jaron Lanier

This chapter introduces general concepts and definitions of attributes, attribute assignments, interfaces, interfaces instances, and intensional couplings, which all can be used to define couplings in variable structure models consistently. These concepts and definitions serve as a foundation for the introduction of the modeling formalism ML-DEVS in Chapter 9.

The chapter makes use of the ideas discusses in Chapters 3 and 7 and is based on concepts and definitions presented in our TOMACS article:

Steiniger, A. and Uhrmacher, A. M. (2016). "Intensional Couplings in Variable Structure Models: An Exploration Based on Multilevel-DEVS." In *ACM Transactions on Modeling and Computer Simulation* (TOMACS), 26(2). pp. 9-1–9-27.

Here, however, we revise and extend some of the definitions given therein.

More details about the mathematical concepts and notations used within this chapter can be found in Appendix A.

8.1 Attributes

The states, inputs, and outputs of systems of interest can often be imagined or perceived as being structured according to certain variables (i. e., state, input, and output variables) rather than being flat, abstract, or opaque (cf. *structured systems* as described by Zeigler et al. [2000, p. 123]). In other words, a state, input, or output of such a system can be described as a vector of values rather than a scalar value, where each coordinate of the vector refers to a characteristic variable. When modeling structured systems, we may have some knowledge about these variables or at least some assumptions about them, which we want to confirm by using simulation. The former is often the case when modeling purely technical, man-made systems (e. g., sensors or wireless access points), whereas the latter often applies to biological systems that include living organisms (e. g., cells or cell organelles).

In the following, we generalize the idea of variables and introduce *attributes* as a more general and abstract concept. An attribute of a model is "something," some characteristic, property, or feature of interest, which is usually subject to changes over time. In the most common case, variables characterizing or defining the state of a model are attributes, like in the rule-based modeling language ML-Rules [Maus et al. 2011]. Other, prominent model attributes are (i) ports that are points of interaction and (ii) parameters that enable the modeler to configure and customize a model [Röhl & Uhrmacher 2008] without redefining its characteristic functions (i. e., changing the model).

In the remainder of this chapter, let $\mathcal{N}$ be a set of names or identifiers (i. e., strings) and $\mathbb{A}$ be a superset[1] of relevant attributes. We then define an attribute as follows:

Definition 8.1.1 (Attribute Definition)
An **attribute definition** (or simply **attribute**), denoted by $attr$ and with $attr \in \mathbb{A}$, is defined by the ordered pair

$$(an, X),$$

where

- an is the name of the attribute with $an \in \mathcal{A}$;
- X is a set of values (value range) that can be assigned to the attribute; and
- $\mathcal{A}$ is a set of attribute names with $\mathcal{A} \subseteq \mathcal{N}$.

Given an attribute with $attr = (an, X)$, we access an and X by writing $attr.an$ and $attr.X$, respectively. Furthermore, we assume that:

$$\forall attr, attr' \in \mathbb{A}\colon\ attr \neq attr' \Leftrightarrow attr.id \neq attr'.id. \qquad \text{(uniqueness of attribute names)}$$

In other words, an attribute (definition) consists of a name and a value range. The name of an attribute needs to be unique to allow referencing the attribute unambiguously.

Remark. According to Definition 8.1.1, there is a difference between an attribute and its name. The latter is part of the attribute, i. e., its definition.

From the modeler's perspective, it is often useful to distinguish between different kinds of attributes explicitly, such as state variables or ports. Therefore, we sometimes adapt the notation from Definition 8.1.1. For instance, let $\mathbb{P}$ be a superset of relevant ports, we define a port as follows:

[1] Section A.1.1 gives a formal definition of supersets and its relationship to universal sets.

Definition 8.1.2 (Port Definition)
A **port definition** (or simply **port**), denoted by *port* with $port \in \mathbb{P}$, is defined by the ordered pair

$$(pn, X),$$

where

- pn is the name of the port with $pn \in \mathcal{P}$;
- X is a set of values (value range) that can be assigned to the port; and
- $\mathcal{P}$ is a set of port names with $\mathcal{P} \subseteq \mathcal{N}$.

Given a port with $port = (pn, X)$, we access pn and X by writing $port.pn$ and $port.X$, respectively.

As with attributes, we assume that port names are unique. When we compare Definition 8.1.2 to Definition 8.1.1, it becomes apparent that both definitions are isomorphic.

8.2 Models

Attributes and ports are associated with models or submodels between which we want to define couplings.

In the following, let $\mathbb{M}$ be a superset of models, usually the components of a certain composition, and let $\mathbb{P}$ be a superset of relevant ports (i. e., port definitions), where each port is as in Definition 8.1.2. Then we define each $m \in \mathbb{M}$ as follows:

Definition 8.2.1 (Model Definition)
A **model definition** (or simply **model**), denoted by m with $m \in \mathbb{M}$, is defined by the ordered pair

$$(id, \Sigma),$$

where

- id is the identifier (or name) of the model with $id \in \mathcal{M}$;
- Σ is a model specification (or model implementation); and
- $\mathcal{M}$ is a set of model identifiers with $\mathcal{M} \subseteq \mathcal{N}$.

Given a model with $m = (id, \Sigma)$, we access id and Σ by writing $m.id$ and $m.\Sigma$. Furthermore, we assume that:

$$\forall m, m' \in \mathbb{M}\colon\ m \neq m' \Leftrightarrow m.id \neq m'.id. \qquad \text{(uniqueness of model identifiers)}$$

Since we are interested in coupling definitions, we assume that a model exhibits ports, via which it communicates with other models, and can be composed of submodels (model components), where a composition forms a tree or hierarchy, i. e., an undirected, acyclic graph, in which any two vertices are connected by exactly one edge[2]. Apart from this, we make no further demands on a model specification at this point. For instance, a model can be specified by a n-tuple of sets and relations on this sets such as finite state machines. Furthermore, we

[2] This reflects the natural structure of complex systems, where one component cannot be a component of itself or a component of one of its subcomponents. In contrast, Dalle, Zeigler, and Wainer [2008] introduces a variation of DEVS in which instances of model components are shared within a model composition.

suppose the existence of the following two auxiliary functions:

$$getPorts\colon \mathbb{M} \to 2^{\mathbb{P}}$$

and

$$getSubmodels\colon \mathbb{M} \to 2^{\mathbb{M}},$$

where the first function, $getPorts(m)$, returns all ports that belong to a given model m and the second function, $getSubmodels(m)$, returns the set of submodels the model m is composed of. The actual implementation of both functions depends on the modeling formalism in which the models are specified and is of no further interest at this point.

8.3 Extensional Couplings

Ports, as a special kind of model attributes, allow us to define *port-to-port couplings* between different models extensionally. This means that each, existing port-to-port coupling is listed or enumerated in the overall coupling definition (cf. extensional definitions in Section 3.2). Port-to-port couplings are well-known in the modeling realm (see, e. g., Zeigler et al. [2000]) and used in modeling formalisms such as DEVS, SysML[3], or Modelica[4].

Given port and model identifiers from the previous sections, we now define *extensional couplings* as follows:

Definition 8.3.1 (Extensional Couplings)
An **extensional coupling definition** for a *coupled model* $n \in \mathbb{M}$ (also *composed model* or *network*) is defined as the set

$$Cplg_{ext} \subseteq Cplg^{*}_{ext} \text{ with } Cplg^{*}_{ext} = \big\{((id_s, pn_s), (id_t, pn_t)) \in (\mathcal{M}_n \times \mathcal{P}_n) \times (\mathcal{M}_n \times \mathcal{P}_n)\big\},$$

where

- id_s and id_t are the identifiers of the source and target model and
- pn_s and pn_t are the names of the source and target port.

$\mathcal{M}_n \subseteq \mathcal{N}$ is the set of the identifier of n and of all its possible submodels and $\mathcal{P}_n \subseteq \mathcal{N}$ is the set of the names of all possible ports of n and of all its possible submodels with

$$\mathcal{M}_n = \bigcup_{m \in \{n\} \cup getSubmodels(n)} \{m.id\},$$
$$\mathcal{P}_n = \bigcup_{m \in \{n\} \cup getSubmodels(n)} \bigcup_{p \in getPorts(m)} \{p.pn\}.$$

The sets $\mathcal{M}_n$ and $\mathcal{P}_n$ define the *namespace* in which couplings can be defined. Each element $((id_s, pn_s), (id_t, pn_t))$ of $Cplg_{ext}$ is called **extensional coupling** or *concrete coupling*. $Cplg^{*}_{ext}$ is the superset of all possible extensional couplings that can be defined for n.

So an extensional coupling definition is a set of (1:1) port-to-port couplings, where each coupling specifies which port of a certain model shall be coupled to which port of another model, using identifiers as references. In other words, an extensional coupling definition lists or enumerates all existing couplings individually and explicitly. All couplings together form the *extension* of the overall coupling scheme (cf. Section 3.1).

[3] `http://sysml.org`; last accessed Februrary 2018
[4] `https://www.modelica.org`; last accessed February 2018

Remark. In the tradition of DEVS and its variants, we denote a coupled model by n rather than m, where $n \in \mathbb{M}$ such as m.

The above coupling definition is very permissive, as only names are taken into consideration when defining couplings. However, it often makes sense to define further constraints on couplings or coupling sets. For instance, to ensure that values sent by a source port are accepted by the target port (*compatibility* or *type coherency*) or to restrict the direction of message flow. For two coupled ports p_s and p_t we typically expect that $p_s.X \subseteq p_t.X$, so that all values originated from the source port p_s are accepted by the target port p_t (cf. Zeigler et al. [2000, p. 130]). In type theory, this relation corresponds to the *subtype relation* or the *principle of safe substitution* (cf. Pierce [2002, pp. 182–7] or Röhl and Uhrmacher [2008]). So in addition to Definition 8.3.1, we can define further constraints on a given coupling set that need to be met. But before we define such constraints, we define another auxiliary function

$$getRange\colon\ \mathcal{N} \rightarrow 2^{\bigcup_{port\,\in\,\mathbb{P}}\{port.X\}} \tag{8.1}$$

that returns the value range for a given port name, denoted by pn with $pn \in \mathcal{N}$, from the set of ports $\mathbb{P}$ with:

$$\forall pn \in \mathcal{N}\colon\ getRange(pn) = \begin{cases} X & \text{if } \exists (pn, X) \in \mathbb{P}, \\ \varnothing & else. \end{cases}$$

By using the above function, we can define useful constraints on extensional coupling definitions, such as:

Definition 8.3.2 (Compatible Ports)
Let c_{ext} be an extensional coupling of a coupled model $n \in \mathbb{M}$, i. e., $c_{ext} \in Cplg_{ext}$, with

$$c_{ext} = ((id_s, pn_s), (id_t, pn_t))$$

as in Definition 8.3.1, then the source port and target port referenced by the names pn_s and pn_t, respectively, are **compatible**, if and only if:

$$X_{pn_s} \subseteq X_{pn_t}$$

with

$$X_{pn_s} = getRange(pn_s), \qquad \text{(value range of source port)}$$
$$X_{pn_t} = getRange(pn_t), \qquad \text{(value range of target port)}$$

and $getRange(pn)$ as in Equation 8.1.

Definition 8.3.3 (Compatible Components)
Let c_{ext} be an extensional coupling of a coupled model $n \in \mathbb{M}$, i. e., $c_{ext} \in Cplg_{ext}$, with

$$c_{ext} = ((id_s, pn_s), (id_t, pn_t))$$

as in Definition 8.3.1, then the source model and target model referenced by the identifiers id_s and id_t, respectively, are **compatible**, if and only if:

$$id_s \neq id_t.$$

In other words, two components are compatible if they are different.

The following example illustrates how couplings can be defined extensionally, according to Definition 8.3.1.

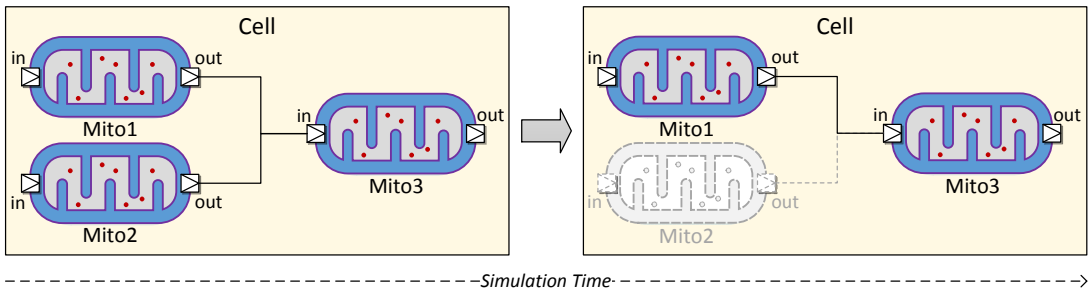

Figure 8.1: A simple variable model composition representing a *mitochondrial network* at two different instants of simulation time. (Left) The composition comprises four components (incl. "Cell") and two port-to-port couplings at time t_1. (Right) At time t_2, the component "Mito1" and its coupling to component "Mito3" are removed.

Example 8.3.1 (Extensional Couplings)
As Section 1.2 describes, mitochondria, cell organelles of eukaryotic cells, that a close to each other can form networks. When modeling such dynamic mitochondrial networks, defining the couplings between mitochondria is of particular interest. Suppose a composition as shown in Figure 8.1 (left)—mimicking a mitochondrial network at a certain time—consisting of three components representing mitochondria. Each of which has two ports (one input and one output port) used for interacting with other mitochondria in the network. Following Definition 8.3.1, an extensional definition of the depicted coupling scheme can now be specified by the following set:

$$Cplg_{ext} = \{((\text{“Mito1”}, \text{“out”}), (\text{“Mito3”}, \text{“in”})), ((\text{“Mito2”}, \text{“out”}), (\text{“Mito3”}, \text{“in”}))\},$$

where the sets $\mathcal{P}_n$ and $\mathcal{M}_n$, based upon which the set $Cplg^*_{ext}$ is defined, are defined as follows:

$$\mathcal{P}_n = \{\text{“in”}, \text{“out”}\},$$
$$\mathcal{M}_n = \{\text{“Cell”}, \text{“Mito1”}, \text{“Mito2”}, \text{“Mito3”}\}.$$

If the model changes its structure, e. g., as shown in Figure 8.1 (right), the coupling definition $Cplg_{ext}$ may need to be adapted as well to reflect the structure change. In this case, the coupling set would be defined by the following singleton set:

$$Cplg_{ext} = \{((\text{“Mito1”}, \text{“out”}), (\text{“Mito3”}, \text{“in”}))\}.$$

Notation 8.3.1 (Variables vs. Values). To distinguish between the names of (bound) variables and values (or literals), we put values into double quotation marks (except numerical values).

Example 8.3.1 already shows a limitation of an extensional coupling definition: This kind of coupling definition can only describe a static snapshot of the actual coupling scheme. In the case of static structure models, this limitation is negligible, as the coupling scheme is not changing during simulation. In variable structure models, however, the coupling scheme can change and the extensional coupling definition thus needs to change as well, otherwise it eventually yields structural inconsistencies.

Moreover, listing all existing couplings exhaustively can become tedious, even for static structure models, especially when there is a great number of components that have to be couple. This brings us to the idea of defining couplings and coupling schemes intensionally.

8.4 Intensional Couplings

Instead of explicitly listing or enumerating all concrete couplings that exist either throughout the whole simulation or within a certain structural context (i. e., at a certain instant of simulation time), an *intensional coupling definition* describes couplings by certain attributes (properties) of the model and by constraints from which concrete couplings can be derived during simulation. Therefore, the intensional coupling definition has to be translated into a concrete coupling scheme, whenever events occur. For this translation, the respective attributes and coupling definitions have to be evaluated. This is the responsibility of the simulator, as the simulator tracks the state and other properties of a model during simulation.

Adapting intensional definitions as described in Section 3.3 to coupling schemes, the *intension* of a coupling definition (the entire scheme) refers to the attributes or characteristics shared by the couplings that are part of the scheme. As a basis for an intensional coupling definition, we propose to use functions, i. e., *coupling functions*, that determine, given the current composition, the concrete coupling scheme.

Definition 8.4.1 (Intensional Couplings)
An **intensional coupling definition** for a coupled model $n \in \mathbb{M}$ is defined as the following set of functions

$$Cplg_{int} \subseteq Cplg^{*}_{int} \text{ with } Cplg^{*}_{int} = \left\{ c_{int} \mid c_{int} \colon 2^{\mathbb{M}_n} \to 2^{Cplg^{*}_{ext}} \right\},$$

where

- c_{int} is an **intensional coupling** (or intensional coupling function);
- $\mathbb{M}_n$ is the superset of all submodels of n and n itself, i. e.,

  $$\mathbb{M}_n = \{n\} \cup getSubmodels(n)$$

  with $\mathbb{M}_n \subseteq \mathbb{M}$;
- $Cplg^{*}_{int}$ is the superset of all possible intensional couplings that can be defined for n;
- $Cplg^{*}_{ext}$ is the superset of all possible extensional couplings as in Definition 8.3.1.

Each argument of such an intensional coupling function of a coupled model, i. e., $M \in 2^{\mathbb{M}_n}$ or $M \subseteq \mathbb{M}_n$, can be interpreted as a concrete incarnation of the coupled model, i. e., the network, at a certain time. Only models that are available at that time, i. e., their identifiers and definitions, appear in the respective argument representing that instant.

The intensional coupling functions, i. e., the concrete mapping, can be specified, e. g., by using predicates defined on the functions' argument and their components. The following example illustrates how an intensional coupling function could look like.

Example 8.4.1 (Intensional Couplings)
Based on Example 8.3.1, suppose we now want to specify a coupling scheme for all possible incarnations of a mitochondrial network, in which all mitochondria are coupled intensionally. Following Definition 8.4.1, such an intensional coupling definition, i. e., the set $Cplg_{int}$, could consist of the function

$$c_{int}(M) = \left\{ ((id_s, \text{“out”}), (id_t, \text{“in”})) \mid m, m' \in M \right. \\ \left. \wedge\, id_s = m.id \wedge id_t = m'.id \wedge \text{“out”} \in \mathcal{P}_m \wedge \text{“in”} \in \mathcal{P}_{m'} \right\}$$

with

$$\mathcal{P}_m = \bigcup_{p \in getPorts(m)} \{p.pn\},$$
$$\mathcal{P}_{m'} = \bigcup_{p \in getPorts(m')} \{p.pn\},$$

where $M \subseteq \mathbb{M}_n$. The set M shall only contain the definitions of available mitochondria (incl. the definition of the cell). To be more specific, for the two instants depicted in Figure 8.1, denoted by t_1 (left-hand side) and t_2 (right-hand side), the corresponding arguments, denoted by M_{t_1} and M_{t_2} with $M_{t_1}, M_{t_2} \subseteq \mathbb{M}_n$, are defined as follows:

$$M_{t_1} = \{(\text{“Cell”}, \Sigma_{\text{Cell}}),\ (\text{“Mito1”}, \Sigma_{\text{Mito1}}),\ (\text{“Mito2”}, \Sigma_{\text{Mito2}}),\ (\text{“Mito3”}, \Sigma_{\text{Mito3}})\},$$
$$M_{t_2} = \{(\text{“Cell”}, \Sigma_{\text{Cell}}),\ (\text{“Mito1”}, \Sigma_{\text{Mito1}}),\ (\text{“Mito3”}, \Sigma_{\text{Mito3}})\},$$

where Σ_{Cell}, Σ_{Mito1}, Σ_{Mito2}, and Σ_{Mito3} are the specifications of the respective models (see Definition 8.2.1). Furthermore,

$$\textit{for } m = (\text{“Cell”}, \Sigma_{\text{Cell}})\colon\ getPorts(m) = \varnothing$$

and

$$\textit{for all } m \in \{(\text{“Mito1”}, \Sigma_{\text{Mito1}}),\ (\text{“Mito2”}, \Sigma_{\text{Mito2}}),\ (\text{“Mito3”}, \Sigma_{\text{Mito3}})\}\colon$$
$$getPorts(m) = \{(\text{“in”}, X_{\text{in}}),\ (\text{“out”}, X_{\text{out}})\}$$

with X_{in} and X_{out} being the value ranges of the two respective ports, assuming that the ports of the corresponding mitochondria have the same value ranges, i. e., X_{in} and X_{out}.

This rather simple coupling definition works for this example, because the cell model has no ports and contains only models of mitochondria. For a more elaborate example, the identifiers of the models would need to be taken into account to prevent unintended couplings, e. g., between mitochondria and the cell. Furthermore, based on the above intensional coupling, all mitochondria are coupled regardless of their actual distance to each other, which does not reflect the nature of mitochondrial networks as described in Section 1.2. The above coupling does also not prevent direct feedback loops or type incoherences of the coupled port. Such constraints can be considered by the actual translation mechanism, which derives concrete, consistent couplings from the intensional coupling definition and the network structure.

For the two incarnations of the mitochondrial network, encoded in M_{t_1} and M_{t_1}, the above intensional coupling function c_{int} returns the following sets of extensional couplings:

$$\begin{aligned} c_{int}(M_{t_1}) = \big\{&((\text{“Mito1”}, \text{“out”}), (\text{“Mito1”}, \text{“in”})),\ ((\text{“Mito1”}, \text{“out”}), (\text{“Mito2”}, \text{“in”})),\\ &((\text{“Mito1”}, \text{“out”}), (\text{“Mito3”}, \text{“in”})),\ ((\text{“Mito2”}, \text{“out”}), (\text{“Mito1”}, \text{“in”})),\\ &((\text{“Mito2”}, \text{“out”}), (\text{“Mito2”}, \text{“in”})),\ ((\text{“Mito2”}, \text{“out”}), (\text{“Mito3”}, \text{“in”})),\\ &((\text{“Mito3”}, \text{“out”}), (\text{“Mito1”}, \text{“in”})),\ ((\text{“Mito3”}, \text{“out”}), (\text{“Mito2”}, \text{“in”})),\\ &((\text{“Mito3”}, \text{“out”}), (\text{“Mito3”}, \text{“in”}))\big\} \end{aligned}$$

and

$$c_{int}(M_{t_2}) = \big\{((\text{“Mito1”}, \text{“out”}), (\text{“Mito1”}, \text{“in”})),\ ((\text{“Mito1”}, \text{“out”}), (\text{“Mito3”}, \text{“in”})),\\ ((\text{“Mito3”}, \text{“out”}), (\text{“Mito1”}, \text{“in”})),\ ((\text{“Mito3”}, \text{“out”}), (\text{“Mito3”}, \text{“in”}))\big\}.$$

To understand the power of this approach, we have to realize that the actual coupling scheme is automatically derived whenever necessary, based on the given coupling functions and the current network structure, i. e., composition. It is possible to specify a coupling between the port “out” of every single mitochondrion with the port “in” of every other mitochondrion by a single intensional coupling, regardless of the actual number of available mitochondria during simulation. This allows us to reduce the specification effort drastically.

Still, Definition 8.4.1 per se does not prevent the modeler to specify mappings of inconsistent couplings, since the range of an intensional coupling function is not constrained by its arguments. The following example illustrates the situation:

Example 8.4.2 (Inconsistent Couplings)
Suppose the following intensional coupling function:

$$c_{int}(M) = \begin{cases} \{((\text{“Mito2”}, \text{“out”}), (\text{“Mito3”}, \text{“in”}))\} & \text{if } M = M_{t_2}, \\ \{((\text{“Mito1”}, \text{“out”}), (\text{“Mito3”}, \text{“in”}))\} & \text{else,} \end{cases}$$

where M_{t_2} as in Example 8.4.1. So for the argument M_{t_2}, the coupling function returns the following singleton set:

$$c_{int}(M_{t_2}) = \{((\text{“Mito2”}, \text{“out”}), (\text{“Mito3”}, \text{“in”}))\},$$

which is an element of the power set $2^{Cplg^*_{ext}}$, thus complies with Definition 8.4.1. However, the singleton set violates our interpretation, since it establishes a coupling between the mitochondria “Mito2” and “Mito3”, although mitochondrion “Mito2” is not present in the argument M_{t_2}, which reflects the situation on the right-hand side of Figure 8.1.

Taking this shortcoming into account, we introduce an additional mechanism that translates intensional couplings into concrete, consistent couplings, which can be used to forward events during simulation. This mechanism (i) evaluates all intensional couplings of a coupled model and (ii) checks certain constraints that need to be fulfilled by the derived concrete couplings so that structural consistency is maintained. Extensional couplings that are returned by an intensional coupling and violate one or more of these constraints are discarded. Hence, the translation of intensional couplings into a concrete coupling scheme ensures *correctness by construction*, without the need for the modeler to take care about structural consistency when defining intensional couplings. One essential constraint refers to the availability of models between which an extensional coupling should be establishes by using the models' identifiers as references.

Definition 8.4.2 (Consistent Coupling)
Let c_{ext} be an extensional coupling of a coupled model $n \in \mathbb{M}$ with $c_{ext} \in Cplg^*_{ext}$, $Cplg^*_{ext}$ as in Definition 8.3.1, and

$$c_{ext} = \{((id_s, pn_s), (id_t, pn_t))\},$$

then c_{ext} is **consistent** with respect to a set M with $M \subseteq \mathbb{M}_n$, where $\mathbb{M}_n$ as in Defini-

tion 8.4.1, if and only if:

$$\exists m, m' \in M : \; m \neq m' \wedge id_s = m.mid \wedge id_t = m'.mid$$

As mentioned in Section 3.2, we may also want to consider further constraints when deriving concrete coupling schemes from an intensional coupling definition and the current model state.

Algorithm 8.1: Simple reference algorithm for translating intensional couplings into a concrete, consistent coupling scheme based on the actual network structure

Input: M {M is the set of currently available models}
Output: *result* {result is the set of concrete, consistent extensional couplings}

```
1:  result ← ∅;
2:  for all c_int in Cplg_int do
3:      Cplg_ext ← c_int(M);
4:      for all c_ext = ((id_s, pn_s), (id_t, pn_t)) in Cplg_ext do
5:          if c_ext is consistent for M and pn_s is compatible with pn_t and id_s is compatible
            with id_t then
6:              add c_ext to result
7:          end if
8:      end for
9:  end for
10: return result
```

Now, Algorithm 8.1 shows how such a translation of an intensional coupling definition into a concrete, consistent coupling scheme can look like. This algorithm checks whether couplings are consistent according to Definition 8.4.2 (line 5). In addition, the algorithm also checks whether the value ranges of the coupled ports are coherent (i. e., compatible according to Definition 8.3.2) and the coupled submodels are different (see Definition 8.3.3).

Depending on the actual application domain, i. e., the domain for which models should be created, more consistency constraints may be necessary. If so, Algorithm 8.1 needs to be extended accordingly (particularly line 5). Herein, we focus on classic consistency constraints as known from the DEVS realm (cf. Zeigler et al. [2000, p. 86]).

Example 8.4.3 shows how an intensional coupling definition is translated into a concrete, consistent coupling by Algorithm 8.1.

Example 8.4.3 (Translation of Intensional Couplings)
Assume an intensional coupling definition as in Example 8.4.1. For the set M_{t_1} with M_{t_1} as in Example 8.4.1, the Algorithm 8.1 returns the following set of concrete, consistent couplings:

$$\big\{((\text{“Mito1”}, \text{“out”}), (\text{“Mito2”}, \text{“in”})), ((\text{“Mito1”}, \text{“out”}), (\text{“Mito3”}, \text{“in”})),$$
$$((\text{“Mito2”}, \text{“out”}), (\text{“Mito1”}, \text{“in”})), ((\text{“Mito2”}, \text{“out”}), (\text{“Mito3”}, \text{“in”})),$$
$$((\text{“Mito3”}, \text{“out”}), (\text{“Mito1”}, \text{“in”})), ((\text{“Mito3”}, \text{“out”}), (\text{“Mito2”}, \text{“in”}))\big\}$$

For the set M_{t_2} with M_{t_2} as in Example 8.4.1, the algorithm returns the following set of concrete, consistent couplings:

$$\big\{((\text{“Mito1”}, \text{“out”}), (\text{“Mito3”}, \text{“in”})), ((\text{“Mito3”}, \text{“out”}), (\text{“Mito1”}, \text{“in”}))\big\}.$$

Now assume the intensional coupling definition given in Example 8.4.2. Algorithm 8.1 returns the empty set for M_{t_1}, i. e., $\varnothing$, since the set M_{t_1} does not contain a model with the

identifier "Mito2", so the extensional coupling returned by the function c_{int} is discarded by the translation algorithm.

In a nutshell, an intensional coupling definition along with a corresponding translation mechanism allows modelers to address structure variability, at a more general level, without considering each possible incarnation of the model structure.

Now the questions are:

- What model attributes should be accessible to determine concrete couplings?
- How can such attributes be accessed?

Example 8.4.1 already gives us an idea of attributes that are of interest for defining couplings, such as names of models and ports and their availability during simulation. For accessing these attributes, we, so far, rely on the existence of auxiliary functions, which return the corresponding attributes from given model definitions and whose implementations are tailored to the underlying modeling formalism. To get rid of such functions, we generalize and emphasize the notion of *interfaces*, based upon which we want to define couplings.

8.5 Interfaces

The idea of defining *model interfaces* explicitly and utilize them for constructing complex, composed models is not new. Already Thomas [1994] introduced a formal definition of model interfaces on top of the in- and output sets of classic, static DEVS models. Interfaces allow us to "separate communication from behavior" [Rowson & Sangiovanni-Vincentelli 1997] and are a prerequisite for a component-based design [Verbraeck 2004; Verbraeck & Valentin 2008], be it software or model design[5]. As such, interfaces form the basis of component-based approaches—e. g., Varga [2001], de Alfaro and Henzinger [2001], Brim, Černá, Vařeková, and Zimmerova [2005], Röhl and Uhrmacher [2008], Rogovchenko and Malenfant [2010], or Peckham, Hutton, and Norris [2013]—for (i) model composition in general and (ii) checking and assuring syntactic and, eventually, semantic composability of composition of model components or models[6].

According to the system-theoretic world view, which we focus on, a system (and thus its representation as a model) is characterized by an internal state (or internal structure), a system boundary, and a system environment, where the internals of the system is not directly accessible from the outside (system as a *black box*). Consistently, some attributes of a system model are not visible to or observable from the outside (system environment), whereas other attributes are. These accessible attributes of a model define its interface, which is the system (model) boundary from a system-theoretic point of view. Or, in other words, the interface of a model declares those of its attributes that shall be accessible from the outside.

Let, in the following, $\mathbb{I}$ be a superset of relevant interfaces. Pursuing and extending the ideas of Thomas [1994] and Röhl and Uhrmacher [2008] on the one hand and the concept of attributes described in Section 8.1 on the other hand, we define an interface of a model as follows:

Definition 8.5.1 (Interface Definition)
An **interface definition** (or simply **interface**), denoted by i with $i \in \mathbb{I}$, is defined by the ordered pair

$$(id, Attr),$$

[5] Simulation models that can directly be executed on a computer (i. e., executable models) are software.

[6] Note that a model component can be viewed as a self-contained model. Thus, we will use the terms "model" and "model component" interchangeably.

where

- $id \in \mathcal{I}$ is a unique identifier, i. e., the interface's name;
- $Attr$ is a set of attribute definitions, which can be published or declared by a model implementing the interfac, where $Attr \subseteq \mathbb{A}$ with $\mathbb{A}$ as in Definition 8.1.1; and
- $\mathcal{I}$ is a set of interface identifiers with $\mathcal{I} \in \mathcal{N}$.

Given an interface with $i = (id, Attr)$, we access id and $Attr$ by writing $i.id$ and $i.Attr$, respectively. Furthermore, we assume that:

$$\forall i, i' \in \mathbb{I}\colon\ i \neq i' \Leftrightarrow i.id \neq i'.id. \qquad \text{(uniqueness of interface identifiers)}$$

Now, different models can implement the same interface, which then declares a set of common attributes shared by the corresponding models. In case that the declared attributes refer solely to ports exhibited by the respective model, our definition of interfaces is equivalent to the notion of model interfaces introduced by Thomas [1994]. In the tradition of DEVS, Thomas explicitly distinguishes between input ports and output ports. We can achieve a similar distinction between different kinds of attributes by allowing the set $Attr$ in Definition 8.5.1 to be defined as a *disjoint union* (see Section A.1.3) of different sets of attribute definitions, where each set represents a different kind of attribute. For instance, if we distinguish between two types of attributes representing input ports and output ports, we can define the set $Attr$ as follows:

$$Attr = Attr_{inputports} \oplus Attr_{outputports}.$$

This way of defining the set of attributes allows us to reuse attribute names for attributes of different types. We will come back to this idea later.

Example 8.5.1 (Interface Definition)
If we recap the mitochondria described in Example 8.3.1 and depicted in Figure 8.1, all mitochondria realizing the same interface that consists of two ports: "in" and "out." If we assume that natural numbers can be assigned to both ports, this simple interface is, according to Definition 8.5.1, defined by the tuple

$$\big(\text{“mito”}, \underbrace{\{(\text{“in”}, \mathbb{N}), (\text{“out”}, \mathbb{N})\}}_{Attr}\big),$$

where "mito" is the identifier of the interface.

If we explicitly defining an interface for an already existing model, as described above, we have to make sure that the model actually adheres to the interface. This means that a model has all the attributes that are declared in its interface. On the other hand, all accessible attributes of a model have to be a part of the model's interface. For this purpose, Röhl and Uhrmacher [2008] and Röhl [2008] introduce special refinement relations, which are used to check whether a model refines its interface and vice versa. In Chapter 9, we show how we can derive interface definitions automatically, without the need to define them by hand. In doing so, the refinement relations hold by construction.

So far, interface definitions neither address structure variability, such as a changing availability of accessible attributes (e. g., variable ports), nor do they allow us to draw any conclusions on the concrete values that are assigned to the accessible attributes during simulation. However, based on these values, we want to define and restrict couplings; which brings as to attribute assignments and interface instances.

8.6 Attribute Assignments and Interface Instances

At a certain instant of time, an attribute is characterized by a value that is assigned to the attribute, where the value is an element of the value range of the attribute, i. e., the set X in Definition 8.1.1. We formally express the relation between attributes and their current values, by a set of *attribute assignments*.

Definition 8.6.1 (Attribute Assignments)
Given a set of attributes, denoted by $Attr$, in which each attribute is as in Definition 8.1.1, we define a set of assignments for $Attr$, denoted by $Assg_{Attr}$, as follows:

$$Assg_{Attr} \subset Assg_{Attr}{}^{*} \text{ with } Assg_{Attr}{}^{*} = \{(an, v, X) \mid (an, X) \in Attr \wedge v \in X \cup \{\varepsilon\}\},$$

where

- an is the name of an attribute from the set $Attr$, i. e., $an \in \mathcal{A}$, to which the value v from the value range X or ε is assigned and
- ε represents the *null value*, i. e., that the corresponding attribute has no value.

Each element of the above set is an attribute assignment assigning a value to a certain attribute of the set $Attr$ by referencing the attribute's name. We access an, v, and X of an assignment $assg$ with $assg \in Assg_{Attr}$ by writing $assg.an$, $assg.v$, and $assg.X$, respectively. Furthermore, we assume that

$$\forall assg, assg' \in Assg_{Attr} \colon\ assg \neq assg' \Leftrightarrow assg.an \neq assg'.an,$$

so that to each attribute from the set $Attr$, at most, one value is assigned.

In a nutshell, for an attribute (an, X) an assignment is the triple (an, v, X), where v is any element of the attribute's value range X or ε.

Example 8.6.1 (Attribute Assignments)
Following Example 8.5.1, let the set Attr contain two attributes, denoted by "in" and "out", whose value ranges are the set of natural numbers, i. e.,

$$Attr = \{(\text{“in”}, \mathbb{N}),\ (\text{“out”}, \mathbb{N})\}.$$

Then, according to Definition 8.6.1, the following sets are consistent attribute assignments for the above set of attributes:

$$\begin{aligned}
Assg_{Attr} &= \varnothing, && \text{(no attribute is available)}\\
Assg_{Attr} &= \{(\text{“in”}, 42)\}, && \text{(only one attribute is available)}\\
Assg_{Attr} &= \{(\text{“in”}, 12),\ (\text{“out”}, 23)\}, && \text{(both attributes are available)}\\
Assg_{Attr} &= \{(\text{“in”}, 12),\ (\text{“out”}, \varepsilon)\}. && \text{(one attribute is empty)}
\end{aligned}$$

To address structure variability at the level of interfaces and assignments to interface attributes, we introduce the notion of *interface instances*, which are runtime (during simulation) instances of interfaces implemented by certain models. Interface instances are similar to object instances as known from the object-oriented programming paradigm. For each model or model component only one of these instances exists at a time.

Definition 8.6.2 (Interface Instance)
An **interface instance** of a model m implementing the interface i is defined by the 3-tuple

$$(mid, iid, Assg),$$

where

- $mid = m.id$ is the unique identifier of the model m;
- $iid = i.id$ is the unique identifier of the interface i;
- $Assg$ is a set of attribute assignments with

$$Assg = Assg_{i.Attr},$$

  where $Assg_{i.Attr}$ as in Definition 8.6.1.

Interface instances are derived and updated by the simulator, which is responsible for executing a given model according to its definition and the respective execution semantics and for keeping track of the model's state, which evolves during simulation.

Now let n be a coupled model with $n \in \mathbb{M}$, then I^n denotes the set of all potential interface instances of n and all of its possible submodels and is defined as follows:

Definition 8.6.3 (Set of Interface Instance Sets)
Given a set of models as in Section 8.3 and Definition 8.2.1, denoted by $\mathbb{M}$ and a set of interface definitions as in Definition 8.5.1, denoted by $\mathbb{I}$, and given a coupled model n with $n \in \mathbb{M}$, we define the superset of all possible interface instance sets for n, denoted by I^n, as follows

$$I^n \subset 2^{\left\{(mid,iid,Assg) \mid m \in \mathbb{M}_n \wedge mid = m.id \wedge i \in \mathbb{I} \wedge iid = i.id \wedge Assg = Assg_{i.Attr}\right\}},$$

where $\mathbb{M}_n \subseteq \mathbb{M}$ is the set of submodels of n including n, i. e.,

$$\mathbb{M}_n = getSubmodels(n) \cup \{n\},$$

and where

- $\mathbb{M}_n \subseteq \mathbb{M}$ is the set of submodels of n including n, i. e.,

$$\mathbb{M}_n = getSubmodels(n) \cup \{n\};$$

- $Assg_{i.Attr}$ as in Definition 8.6.1 for the model m and the interface i.

Furthermore, we assume that

$$\forall i^n \in I^n \; \forall i, i' \in i^n \colon \; i \neq i' \Leftrightarrow i.mid \neq i'.mid, \qquad \text{(uniqueness of interface instances)}$$

so that each element of I^n, which is a set of interface instances, contains, at most, one interface instance for each model, i. e., model identifier. Keeping the idea of well-defined model interfaces in mind[a], we also assume that

$$\forall i^n, i^{n\prime} \in I^n \; \forall i \in i^n \; \nexists i' \in i^{n\prime} \colon \; i \neq i' \wedge i.mid = i'.mid \wedge i.iid \neq i'.iid,$$

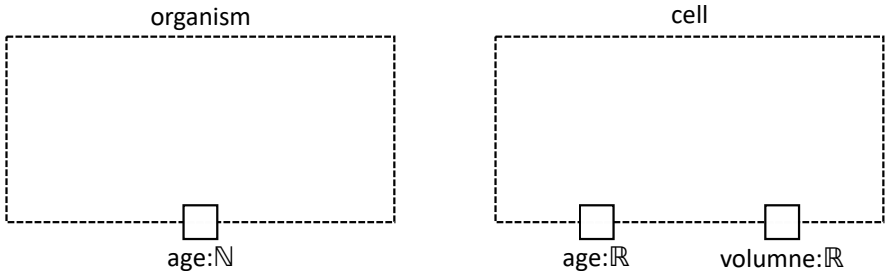

Figure 8.2: The figure shows two different interfaces, denoted by "organism" and "cell." Both interfaces have an attribute with the name "age," however, the value ranges of both attributes is different, i. e., incompatible. The interface "cell" has an additional attribute with the name "volumne."

or, in other words, that each model implements exactly one interface. Still, several models can implement the same interface.

[a] Only if an interface is well-defined, i. e., we know its potential communication and interaction capabilities beforehand, we can make statements about composability based solely upon the interface. See Section 7.4.1.

Since the actual composition of a model is not an explicit part of the model according to Definition 8.2.1, the above definition still relies on the existence of an auxiliary function *getSubmodels*, which returns the submodels of a given coupled model, if there are any. Although Definition 8.6.3 prohibits models to change their interfaces, it still allows changing the availability of interface attributes from one interface instance to another (cf. *variable ports*). All attributes whose names do not appear in the set *Assg* of a particular interface instance are not available. In contrast, changing its interface would allow a model to change the value ranges of its interface attributes in addition to their availability (see Figure 8.2). Whether or not changing interfaces is desirable or makes sense, shall not be further discussed at this point. If necessary, the respective constraint can simply be removed from Definition 8.6.3, so that models can change their interfaces.

Each element of the set I^n can be interpreted as a snapshot of the interface instances of all submodels of a coupled model available at a certain time during simulation (incl. the instance of the coupled model's interface); reflecting a particular incarnation of the model structure, i. e., the including model interfaces. Example 8.6.2 illustrates this interpretation of the set I^n:

Example 8.6.2 (Interface Instances)
Recreating the model described by Patel et al. [2013], in which mitochondria in close proximity exchange "health units," we define a corresponding interface that is implemented by all mitochondria, denoted by i_{mito}, as follows:

$$i_{\text{mito}} = (\text{“mito”}, \{(\text{“fuse”}, \mathbb{N}_0), (\text{“pos”}, \mathbb{R} \times \mathbb{R})\})$$

The interface consists of two attributes: "fuse" and "pos." The first attribute represents a bidirectional port via which a number of abstract health units can be exchanged between interacting mitochondria within the same mitochondrial network. The second attribute indicates the spatial location of a mitochondria in the cell, where, for simplicity, we assume two-dimensional locations. Furthermore, let there be the following interface:

$$i_{\text{cell}} = (\text{“cell”}, \varnothing)\,,$$

which is implemented by the cell containing the mitochondria and has no attributes, i. e., $i_{\text{cell}}.Attr$ is the empty set, since we are only interested in intracellular activities.

Now suppose the two different situations depicted in Figure 8.3 (left and right side), the corresponding sets of interface instances would look as follows:

$$\begin{aligned} I_{t_1} = \big\{&(\text{“Mito1”},\ \text{“mito”},\ \{(\text{“fuse”},\ 0),\ (\text{“pos”},\ (1.9,\ 2.3))\}),\\ &(\text{“Mito2”},\ \text{“mito”},\ \{(\text{“fuse”},\ 0),\ (\text{“pos”},\ (1.8,\ 2.4))\}),\\ &(\text{“Mito3”},\ \text{“mito”},\ \{(\text{“fuse”},\ 0),\ (\text{“pos”},\ (1.9,\ 2.4))\}),\\ &(\text{“Cell”},\ \text{“cell”},\ \varnothing)\big\} \end{aligned}$$

and

$$\begin{aligned} I_{t_2} = \big\{&(\text{“Mito1”},\ \text{“mito”},\ \{(\text{“fuse”},\ 0),\ (\text{“pos”},\ (1.9,\ 2.3))\}),\\ &(\text{“Mito2”},\ \text{“mito”},\ \{(\text{“fuse”},\ 0),\ (\text{“pos”},\ (8.0,\ 7.8))\}),\\ &(\text{“Mito3”},\ \text{“mito”},\ \{(\text{“fuse”},\ 0),\ (\text{“pos”},\ (1.9,\ 2.4))\}),\\ &(\text{“Mito4”},\ \text{“mito”},\ \{(\text{“fuse”},\ 0),\ (\text{“pos”},\ (8.0,\ 7.9))\}),\\ &(\text{“Cell”},\ \text{“cell”},\ \varnothing)\big\}, \end{aligned}$$

where ‘0’ is assigned to all the ports and both sets I_{t_1} and I_{t_2} are proper subsets of the respective set I^n, i. e., $I_{t_1}, I_{t_2} \subset I^n$, with

$$\begin{aligned} \mathbb{M} = \big\{&(\text{“Mito1”},\ \Sigma_{\text{Mito1}}),\ (\text{“Mito2”},\ \Sigma_{\text{Mito2}}),\\ &(\text{“Mito3”},\ \Sigma_{\text{Mito3}}),\ (\text{“Mito4”},\ \Sigma_{\text{Mito4}}),\\ &(\text{“Cell”},\ \Sigma_{\text{Cell}})\big\} \end{aligned}$$

and

$$\mathbb{I} = \big\{\underbrace{(\text{“mito”},\ \{(\text{“fuse”},\ \mathbb{N}_0),\ (\text{“pos”},\ \mathbb{R}\times\mathbb{R})\})}_{i_{\text{mito}}},\ \underbrace{(\text{“cell”},\ \varnothing)}_{i_{\text{cell}}}\big\},$$

where

$$n = (\text{“Cell”}, \Sigma_{\text{Cell}})$$

In the above cases, all attributes are available at both instants of simulation time (i. e., t_1 and t_2). Another consistent interface instance, which illustrates a change of the availability of attributes in interface instances, would be

$$(\text{“Mito1”},\ \text{“mito”},\ \{(\text{“pos”},\ 4.2)\})\,.$$

The above interface instance indicates that the port “fuse” of the model “Mito1” is currently not available and thus cannot be used for exchanging health units, even if the model is in direct proximity to models of other mitochondria.

Although interface instances can change during simulation, we can use them to revise the above definition of intensional couplings. Furthermore, attributes of an interface instance can reflect state variables of the respective model and their current values.

8.7 Intensional Interface Couplings

Using interface instances as introduced in the previous section, we can revise Definition 8.4.1 and define intensional couplings based upon time-variant interface instances rather than static

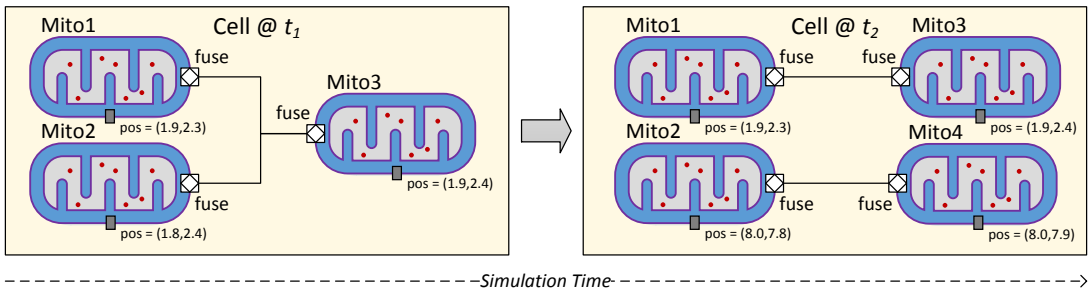

Figure 8.3: The coupled model "Cell" comprises mitochondria whose locations in the cell can change and which can form a mitochondrial network. Only mitochondria close to each other belong to the same spatial cluster and can communicate, i. e., are coupled.

model definitions, where interface instances reflect the availability of certain model attributes and their values during simulation.

Definition 8.7.1 (Intensional Interface Coupling)
Given a set of models denoted by $\mathbb{M}$, as in Section 8.1, and a set of interface definitions denoted by $\mathbb{I}$, as in Section 8.5, an **intensional coupling definition** for a coupled model $n \in \mathbb{M}$, denoted by $Cplg_{int}$, is defined as a set of functions:

$$Cplg_{int} \subseteq Cplg^*_{int} \text{ with } Cplg^*_{int} = \left\{ c_{int} \mid c_{int} \colon I^n \to 2^{Cplg^*_{ext}} \right\},$$

where

- $Cplg^*_{int}$ is the superset of all possible intensional interface couplings for n;
- $\mathbb{I}$ is the superset of interface instance sets for n as in Definition 8.6.3; and
- $Cplg^*_{ext}$ is the set of all possible extensional couplings for n as in Definition 8.3.1.

Each function c_{int} is an **intensional interface coupling** (or short **intensional coupling**), where both $Cplg^*_{int}$ and c_{int} are defined different than in Definition 8.4.1.

An intensional interface coupling maps sets of interface instances of a coupled model, each of which represents a specific incarnation of the structure of the coupled model, to concrete coupling schemes consisting of extensional couplings. In other words, an intensional interface coupling determines a concrete coupling scheme based on the current model structure, which is the argument of the coupling function. These concrete, derived coupling schemes may still be subject to further constraints (see Section 9.2.2)[7].

Since we are now dealing with runtime instances of interfaces of available models[8] rather than static model definitions, we can take changes of interesting attributes, i. e., their values, or their availability during simulation into account when defining coupling schemes for variable structure models.

The following example illustrates, how an intensional interface coupling can be used to define a complex coupling scheme for a variable structure model concisely.

Example 8.7.1
As indicated in Example 8.6.2, Patel et al. [2013] describe a model in which mitochondria

[7] For this reason, not all extensional couplings that are returned by an intensional coupling may be used by the simulator of the respective modeling formalism to forward events.

[8] The interface instances of models that are not available at a certain time do not appear in the respective set of interface instances.

that are close to each other can exchange health units, representing the influence of impaired mitochondria on healthy ones. Mitochondria that are far away from each other cannot interact in such a manner. Consistently, when we want to define a coupling scheme that "connects" only mitochondria that are close to each other, we need to incorporate their current spatial location in the cell, which can change over time, into the coupling definition.

Using interface instances and intensional interface couplings allows us to achieve such a distance-based coupling scheme, without the need to consider each possible situation individually. Given the two interfaces defined in Example 8.6.2, we can define a distance-based interaction between mitochondria, e. g., by the following intensional coupling:

$$\begin{aligned} c_{int}(I) = \big\{((id_s,\ \text{“fuse”}),\ (id_t,\ \text{“fuse”})) \mid i, i' \in I \wedge i \neq i' \wedge id_s = i.mid \wedge id_t = i'.mid \\ \wedge\ asg, asg' \in i.Assg \wedge asg'', asg''' \in i'.Assg \wedge asg.an = asg''.an = \text{“fuse”} \\ \wedge\ asg'.an = asg'''.an = \text{“pos”} \wedge dist(asg'.v,\ asg'''.v) \leqslant 1.0\big\}, \end{aligned}$$

where $I \in I^n$ with I^n as in Definition 8.6.3 and the binary function $dist(x, y)$ calculates the Euclidean distance between the two locations x and y.

Now suppose the two situations depicted in Figure 8.3 (left and right side) represented by the interface instances sets I_{t_1} and I_{t_2} as defined in Example 8.6.2. For I_{t_1}, the above intensional interface coupling c_{int} returns the following set of extensional couplings:

$$\begin{aligned} c_{int}(I_{t_1}) = \big\{&((\text{“Mito1”},\ \text{“fuse”}),\ (\text{“Mito2”},\ \text{“fuse”})), \\ &((\text{“Mito1”},\ \text{“fuse”}),\ (\text{“Mito3”},\ \text{“fuse”})), \\ &((\text{“Mito2”},\ \text{“fuse”}),\ (\text{“Mito1”},\ \text{“fuse”})), \\ &((\text{“Mito2”},\ \text{“fuse”}),\ (\text{“Mito3”},\ \text{“fuse”})), \\ &((\text{“Mito3”},\ \text{“fuse”}),\ (\text{“Mito1”},\ \text{“fuse”})), \\ &((\text{“Mito3”},\ \text{“fuse”}),\ (\text{“Mito2”},\ \text{“fuse”}))\big\}. \end{aligned}$$

Accordingly, for I_{t_2}, c_{int} returns the following set of extensional couplings:

$$\begin{aligned} c_{int}(I_{t_2}) = \big\{&((\text{“Mito1”},\ \text{“fuse”}),\ (\text{“Mito3”},\ \text{“fuse”})), \\ &((\text{“Mito2”},\ \text{“fuse”}),\ (\text{“Mito4”},\ \text{“fuse”})), \\ &((\text{“Mito3”},\ \text{“fuse”}),\ (\text{“Mito1”},\ \text{“fuse”})), \\ &((\text{“Mito4”},\ \text{“fuse”}),\ (\text{“Mito2”},\ \text{“fuse”}))\big\}. \end{aligned}$$

So from a single coupling function that considers the spatial locations of the available mitochondria, concrete coupling schemes for all possible situations can be derived. The actual location of a mitochondrion during simulation is accessible via its interface, i. e., is part of the interface instance that can change during simulation, capturing the movement of mitochondria (cf. Park et al. [2011]). The set I^n defines the domain for all intensional interface couplings that can be defined for a coupled model n (see Definition 8.6.3).

In the remainder of this thesis, we refer to intensional interface couplings as in Definition 8.7.1 when talking about intensional couplings.

8.8 Translation of Intensional Interface Couplings

Similar to the intensional couplings as in Definition 8.4.1, Definition 8.7.1 per se does not prevent the modeler to violate our interpretation of concrete couplings derived from intensional interface couplings. For this reason, we adapt the translation mechanism introduced in Section 8.4 to cope with interface instances instead of model definitions.

Definition 8.8.1 (Consistent Concrete Couplings)
Let c_{ext} be an extensional interface coupling of a coupled model $n \in \mathbb{M}$ with $c_{ext} \in Cplg^*_{ext}$, $Cplg^*_{ext}$ as in Definition 8.7.1, and

$$c_{ext} = \{((id_s, pn_s), (id_t, pn_t))\},$$

then c_{ext} is **consistent** with respect to a set of interface instances i^n with $i^n \in I^n$, where I^n as in Definition 8.6.3, if and only if:

$$\exists i, i' \in i^n : \; i \neq i' \wedge id_s = i.mid \wedge id_t = i'.mid.$$

The actual translation algorithm needs to be adapted accordingly. Algorithm 8.2 shows the result of this adaption.

Algorithm 8.2: Reference algorithm for translating intensional interface couplings into a concrete, consistent coupling scheme based on a set of interface instances

Input: i^n, $Cplg_{int}$
Output: *result* {set of concrete, consistent extensional couplings}

```
1:  result ← ∅;
2:  for all c_int in Cplg_int do
3:    Cplg_ext ← c_int(i^n);
4:    for all c_ext = ((id_s, pn_s), (id_t, pn_t)) in Cplg_ext do
5:      if c_ext consistent for i^n and pn_s compatible with pn_t and id_s compatible with id_t
        then
6:        add c_ext to result
7:      end if
8:    end for
9:  end for
10: return result
```

In contrast to Example 8.4.3, Algorithm 8.2 will not discard any extensional couplings returned by the intensional coupling function c_{int} from Example 8.7.1, since the function already ignores the same components. Note that this constraint does not need to be part of the intensional coupling, because it is already part of the translation algorithm. However, it shows that we can encode such constraints in the intensional coupling itself.

8.9 Summary

This chapter presents a fundamental concept for defining couplings in variable structure models based on interface instances (which can change) and by exploiting intensional definition techniques as presented and discussed in Chapter 3. The concept itself is not bound to a certain modeling formalism, instead it can be incorporated into any modeling formalism that supports the notion of ports and emphasizes on a clear separation between model specification and simulation algorithm. Structural consistency is maintained by the fact that intensional couplings are translated into concrete couplings during execution while checking and enforcing certain consistency rules (correctness by construction). Thereby intensional couplings do not correspond to concrete couplings, instead they serve as blueprints for deriving concrete couplings during execution. This chapter represents algorithms that translate intensional couplings into concrete couplings for a given state and model incarnation. Furthermore, the chapter provide some examples that illuminate the potential of intensional coupling definitions.

9 Revision of the Multi-Level Discrete Event System Specification

> The computing scientist's main challenge is not to get confused by the complexities of his own making.
>
> Edsger W. Dijkstra

This chapter introduces a major revision of the modeling formalism *Multi-Level Discrete Event Systems Specification* (ML-DEVS), which is one of the primary outcomes and contributions of this thesis. In general, we use the formalism to model and simulate systems of interest and their constituents (system components[1]), such as smart environments and their components or cells and their organelles. With respect to the component-based modeling methodology presented in Chapter 8, ML-DEVS is used to specify the behavior of model components on the one hand (as source formalism[2]) and as a suitable target formalism for deriving and synthesizing executable simulation models from composition descriptions and model specifications on the other hand.

Major parts of this chapter are based on and adopted from the following publications, especially the last one:

Steiniger, A., Krüger, F., and Uhrmacher, A. M. (2012). "Modeling Agents and their Environment in Multi-Level-DEVS." In *Proceedings of the 2012 Winter Simulation Conference* (WSC'12). Article No. 233.

Steiniger, A. and Uhrmacher, A. M. (2016). "Intensional Couplings in Variable Structure Models: An Exploration Based on Multilevel-DEVS." In *ACM Transactions on Modeling and Computer Simulation* (TOMACS), 26(2). pp. 9-1–9-27.

[1] A system component can be considered as a system itself.

[2] As Chapter 5 describes, we may also want to use different modeling formalisms to specify the behavior of different model components (multi-formalism modeling).

9.1 Multi-Level Discrete Event System Specification

ML-DEVS—at its core—is based on Parallel DEVS (P-DEVS), a parallel variant of the modular, hierarchical modeling formalism for *parallel discrete event simulation* (see Section 4.2). As a member of the DEVS family, ML-DEVS describes a system of interest as a *reactive, discrete event system*. Uhrmacher et al. [2007] and Uhrmacher et al. [2010] proposed first ideas and concepts of ML-DEVS in the domain of computational systems biology, where the following aspects of particular interest: (i) the seamless combination of different levels of behavior and organization ranging from proteins over cells to cell populations and (ii) the interrelations and interactions between those levels [Maus et al. 2011]. Consequently, ML-DEVS addresses the above interests by allowing the modeler to model and combine multiple levels of behaviors and by providing dedicated mechanisms to specify interdependencies between the different levels of behavior explicitly (*up- and downward causation*). To capture the structural variability intrinsic to biological systems, ML-DEVS supports—in the tradition of other DEVS variants such as Variable DEVS (V-DEVS) [Barros et al. 1994], Dynamic Structure DEVS (DSDEVS) [Barros 1995a, 1996], or Extended Dynamic Structure DEVS [Hagendorf et al. 2009]—variable structures[3] in a top-down manner[4]. However, especially in our latest revision of ML-DEVS, models at lower levels of behavior can trigger structure changes at higher levels (via upward causation). As we will see later, we can also mimic the decentralized, bottom-up approach of changing the model structure as realized in DEVS variants such as dynDEVS [Uhrmacher 2001] or ρ-DEVS [Uhrmacher et al. 2006]. Moreover, ML-DEVS supports *variable ports*[5] as discussed by Uhrmacher and Priami [2005] or Hu et al. [2005] and proposed in ρ-DEVS [Uhrmacher et al. 2006].

So far, ML-DEVS has been used in computational systems biology [Maus 2008; Uhrmacher et al. 2007, 2010], computational demography [Zinn 2011], and business informatics [Stiffel 2014]. As part of this thesis, ML-DEVS has been explored for modeling and simulation in the area of *ubiquitous computing*, especially for modeling and simulating *smart environments* [Krüger et al. 2012; Steiniger et al. 2012], which are closely related to *multi-agent systems*. Furthermore, we evaluated the applicability and suitability of ML-DEVS for continuous-time, demographic microsimulation [Steiniger et al. 2014]. In smart environments as well as demographic systems, the accessibility to certain, often global information plays a central role. In addition, we are also dealing with macro behavior that is *emerging* from micro behavior (*micro-macro effect* or upward causation) in ubiquitous computing [Poslad 2009, p. 333]. As a result of the exploration of ML-DEVS, we revised and extended the original formalism (as presented by Uhrmacher et al. [2007]) to (i) increase its suitability for modeling and simulating multi-agent systems such as smart environments, (ii) ease the specification of multi-level and variable structure models in general, and (iii) provide a sound, consistent, and rigorous definition of the formalism. The question, how to ease the specification of variable structure models, especially their communication structure, in a system-theoretic modeling approach such as ML-DEVS is addressed by the introduction of a novel, flexible coupling scheme, which is the central concept in our revision of the formalism[6]. In general, our revision of ML-DEVS, presented in the remainder of this chapter, includes, among other things:

- An overhaul and adaptation of the formal definition of ML-DEVS;
- The fusion of input ports and output ports;

[3] also called *dynamic structures* (cf. Barros [1995a])

[4] Section 6.3 gives a more comprehensive overview of variable structure variants of DEVS.

[5] In contrast to static ports, the availability of variable ports can change during simulation. Thereby, variable ports mimic the plasticity of interfaces that is characteristic for some systems [Uhrmacher et al. 2006] resulting in variable model interfaces.

[6] The proposed coupling mechanism is not limited to ML-DEVS, but can also be adapted for other, system-theoretic modeling approaches (e. g., other DEVS variants or SysML).

- The introduction and emphasis of interface incarnations (i. e., interface instances);
- The introduction of a novel, more expressive intensional coupling mechanism;
- The separation between public states and private states of ML-DEVS models;
- A formal proof of the *closure under coupling* of ML-DEVS;
- The adaption of the abstract simulator and simulation protocol of ML-DEVS.

Whenever necessary, we motivate the individual changes regarding the original version of ML-DEVS in more detail in the following sections. The original definition of ML-DEVS and its abstract simulator can be found in Uhrmacher et al. [2007].

According to Sarjoughian [2006, attributed to Sarjoughian and Zeigler [2000]] a modeling formalism consists of a model specification and an execution algorithm (i. e., the execution semantics). In the next section, we first show how models are specified in our major revision of ML-DEVS.

9.2 Model Specification in Multi-Level DEVS

As stated in Section 4.2, DEVS and its variants usually distinguish between *atomic models* and *coupled models* (also called *networks*). The former are basic models of the respective formalism, whereas the latter describe networks of interacting basic models and/or coupled models (if the formalism is closed under coupling). Similarly, ML-DEVS distinguishes between two types of models: Micro-DEVS models and Macro-DEVS models[7]. The former correspond to the atomic models (leaves of a composition hierarchy[8]) of other DEVS variants, whereas the latter correspond to the coupled models (inner nodes of a composition hierarchy). However, in contrast to traditional coupled models, which merely serve as containers for their components (submodels), Macro-DEVS models have a state and behavior of their own. In contrast to dynDEVS, PdynDEVS, or ρ-DEVS, the state and behavior of a Macro-DEVS model can cover more than the states of its components and changing the network structure, respectively. Realizing such kind of central control in traditional DEVS variants, would require the specification and addition of extra components that represent certain dynamics exhibited by coupled models and beyond the behavior that emerges from the mere interaction of the coupled models' components. Such an additional component usually becomes a central component through which all other components of a coupled model have to communicate (see Figure 9.1).

Remark. In our previous publications on ML-DEVS, we used, for brevity, the terms "micro model" and "macro model" as short forms for Micro-DEVS model and Macro-DEVS model, respectively. However, as we see below, this may be confusing with respect to the common understanding of the terms "micro model" and "macro model." To avoid confusion and ambiguities, we will not use these terms interchangeably herein. Instead we will use atomic model and coupled model as short forms for Micro-DEVS model and Macro-DEVS model, respectively, according to the established terminology in the realm of DEVS.

With respect to multi-level modeling, a Macro-DEVS model (short coupled model) usually represents a macroscopic level of behavior (*macro level*), e. g., a group of individuals or an ensemble of devices. The behavior of the coupled model's components and their interaction,

[7] The naming is inspired by the formalism's focus on modeling and simulating multiple levels of behavior and their interdependencies, i. e., multi-level modeling. These different levels reflect microscopic and macroscopic views on the system of interest at the same time.

[8] As ML-DEVS allows hierarchical modeling, we can view a ML-DEVS model as a tree whose nodes are model components; where each model component is either a Micro-DEVS or Macro-DEVS model and only Macro-DEVS models can have child nodes.

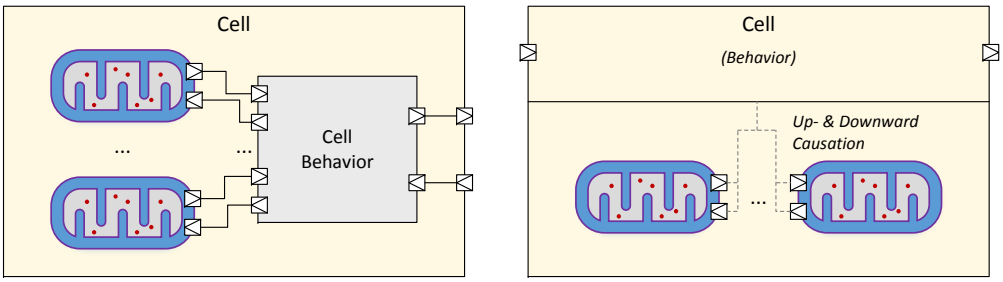

Figure 9.1: Modeling the macroscopic behavior of an eukaryotic cell. (Left) The behavior of the cell is represented by the additional, central atomic component "Cell Behavior" that can interact with the components of the cell reflecting up- and downward causation between the macro and micro level. (Right) The coupled model "Cell" itself has a state and behavior of its own. Up- and downward causation are explicitly expressed by mechanisms additional to the classic (horizontal) couplings.

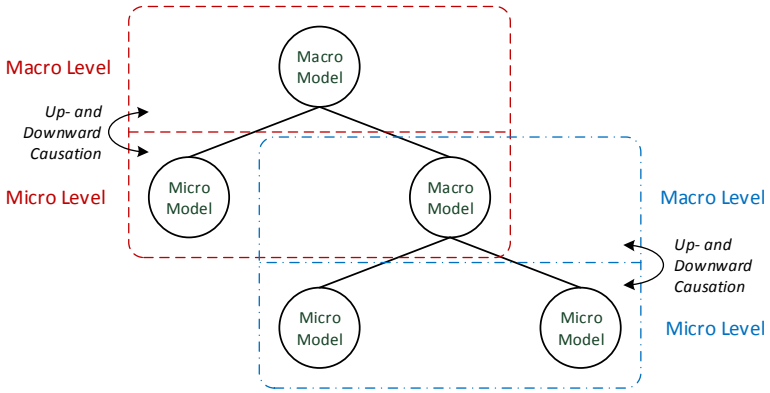

Figure 9.2: A hierarchy of multiple levels of behavior in ML-DEVS. The hierarchy is structured into pairs of micro and macro levels. According to its location in the hierarchy, a Macro-DEVS model can be part of the micro level of its parent.

on the other hand, usually represent a microscopic level of behavior (*micro level*)—from the point of view of the superordinate macro level. Note that due to the *closure under coupling* of ML-DEVS, the components of a coupled model can be coupled models themselves, leading to a hierarchy of multiple levels of behaviors structured into pairs of micro and macro levels[9] (see Figure 9.2). Depending on the desired level of abstraction for a modeling problem at hand, a Micro-DEVS model (short *atomic model*) can represent a certain individual or an entire population whose internal structure is of no further interest. In the former case, the atomic model can be viewed as a micro model in the classic sense. In the latter case, the atomic model can be viewed as a traditional macro model aggregating the behavior and interaction of its components. In the following, both types of models, Micro-DEVS and Macro-DEVS, are formally defined and their operational semantics is described informally. Section 9.3 gives a formal definition of the semantics of ML-DEVS by means of an *abstract simulator*. Both types of models are defined at the level of *structured systems*[10], assuming that states, inputs, and outputs of the models are structured according to certain variables—state variables, input variables, and output variables. As characteristic for the DEVS realm and automata theory[11] in general, Micro-DEVS and Macro-DEVS are defined set-theoretically, by a number of

[9] Poslad [2009, p. 333] also uses the terms "global level" and "local level" as synonyms for "macro level" and "micro level", respectively.

[10] Note that the structured system level does not correspond to the level of *structure systems* of the *System Specification Hierarchy* presented by Zeigler et al. [2000, p. 13].

[11] As Section 4.2 describes, we can view a DEVS model as an extension of finite state automata.

characteristic sets and functions or relations on these sets.

9.2.1 Micro-DEVS Models

As in Chapter 8, let $\mathcal{N}$ denote a superset of names and identifiers. In the most general case, $\mathcal{N}$ consists of all possible strings, like a *universal set*. A MICRO-DEVS model (short atomic ML-DEVS model) is then formally defined as follows:

Definition 9.2.1 (Micro-DEVS)
A **Micro-DEVS model** is defined as the structure

$$\langle id, XY, S_p, S_a, s_{init}, p, \delta, \lambda, ta \rangle$$

where

- id is an *identifier* with $id \in \mathcal{N}$;
- XY is a structured set of *in- and outputs* that is defined as the partial Cartesian product $\widetilde{\prod}_{pn \in \mathcal{P}} XY^{\varepsilon}_{pn}$ of a family of arbitrary sets $\{XY_{pn}\}_{pn \in \mathcal{P}}$ indexed by the nonempty set $\mathcal{P}$, where
  - $\mathcal{P} \subseteq \mathcal{N}$ is a set of port names (ports),
  - XY_{pn} denotes the value range of the port whose name is pn,
  - $XY^{\varepsilon}_{pn} = XY_{pn} \cup \{\varepsilon\}$ with ε being the *non-value* indicating an *empty port*,
  - for all $pn \in \mathcal{P}:\ \varepsilon \notin XY_{pn}$;

  S_p is a structured sets of *private states* that is defined as the generalized Cartesian product $\prod_{psvn \in \mathcal{V}_p} S_{psvn}$ of a family of arbitrary sets $\{S_{psvn}\}_{psvn \in \mathcal{V}_p}$ indexed by the nonempty set $\mathcal{V}_p$, where
  - $\mathcal{V}_p \subseteq \mathcal{N}$ is a set of names of private state variables,
  - S_{psvn} denotes the value range of the private state variable whose name is $psvn$;
- S_a is a structured sets of *accessible states* that is defined as the generalized Cartesian product $\prod_{asvn \in \mathcal{V}_a} S_{asvn}$ of a family of arbitrary sets $\{S_{asvn}\}_{asvn \in \mathcal{V}_a}$ indexed by the nonempty set $\mathcal{V}_a$, where
  - $\mathcal{V}_a \subseteq \mathcal{N}$ is a set of names of accessible (or public) state variables,
  - S_{asvn} denotes the value range of the accessible state variable whose name is $asvn$;
- s_{init} is an initial state with $s_{init} = (s_{p,i}, s_{a,i})$, where
  - $s_{p,i} \in S_p$ is the *initial private state*,
  - $s_{a,i} \in S_a$ is the *initial accessible state*;
- $p\colon\ S_p \to 2^{\mathcal{P}}$ is the *port selection function*;
- $\delta\colon\ Q \times XY^b \rightsquigarrow S_p \times S_a$ is the *state transition function* with Q being the set of *total states* $\{(s_p, s_a, e) \mid s_p \in S_p, s_a \in S_a, 0 \leqslant e \leqslant ta(s_p)\}$, e being the time elapsed since the last state transition, and XY^b being a set of bags (i. e., bag set) over the elements in XY, where

$$dom(\delta) = \left\{ ((s_p, s_a, e), xy^b) \in Q \times XY^b \mid \forall xy \in xy^b\colon\ dom(xy) \subseteq p(s_p) \right\};$$

- $\lambda\colon\ S_p \to XY^b$ is the *output function*;
- $ta\colon\ S_p \to \mathbb{R}^+_0 \cup \{\infty\}$ is the *time advance function*.

The MICRO-DEVS model is subject to the following constraints:

1. For all $s_p \in S_p$ holds $|p(s_p)| = k$ with $k \in \mathbb{N}$ (*finiteness of selected port names*);
2. For all $s_p \in S_p$ and $xy \in \lambda(s_p)$ holds $dom(xy) \subseteq p(s_p)$ (*only selected ports can be used for outputs*)[a].

Given a MICRO-DEVS model m with $m = \langle id, XY, S_p, S_a, s_{init}, p, \delta, \lambda, ta \rangle$ we may access the different elements of m (i. e., the sets and functions) by writing $m.id$, $m.XY$, $m.S_p$, $m.S_a$, $m.s_{init}$, $m.p$, $m.\delta$, $m.\lambda$, and $m.ta$.

[a] The unary function $dom()$ returns the domain of a function (see Definition A.1.12).

Thus each MICRO-DEVS model is defined by: (i) a unique *identifier id*, that is the model's name[12]; (ii) the structured sets XY, S_p, and S_a; (iii) an initial state s_{init}; and (iv) the functions p, δ, λ, and ta. The set XY denotes the structured set of *in- and outputs* of an atomic ML-DEVS model, where *ports* are used as central points of interaction via which the atomic model communicates with its surroundings, i. e., input and output variables refer to ports[13]. Different from other DEVS variants, ML-DEVS does not explicitly distinguish between in- and output ports. Instead, ports can be used for both receiving inputs and sending outputs (see Section 9.3), similar to bidirectional ports in UML and SysML. The set S_p denotes the set of all *private states* an atomic model might enter (private state space). The set S_a, on the other hand, denotes the set of all *accessible* (or *public*) *states* of an atomic model (accessible state space). The accessible state of an atomic model can be accessed by its parental coupled model. Thereby, changes of the accessible state can cause changes at the macro level (more details on up- and downward causation can be found in Section 9.2.2). In contrast to Steiniger et al. [2012], the accessible state space of an atomic model is now defined by a distinct set, i. e., S_a, and is not included in the set XY. Thus we are not using ports to announce the accessible state anymore, instead public states are separate from the regular outputs of an atomic model. The sets $\mathcal{P}$, $\mathcal{V}_p$, and $\mathcal{V}_a$ denote the sets of port names, private state variable names, and accessible state variable names, respectively, according to which the sets XY, S_p, and S_a, are structured. The initial state of an atomic model is defined by s_{init} and comprises an initial private state $s_{p,i}$ and an initial accessible state $s_{a,i}$.

Remark. In contrast to previous iterations of ML-DEVS, we use *generalized Cartesian products*[14] instead of *multivariable sets*, as defined by Zeigler et al. [2000, pp. 123–5], to refine the sets of states, inputs, and outputs, i. e., to explicitly structure the characteristic sets according to certain variables. Unlike multivariable sets, generalized Cartesian products allow us to work with ordinary sets instead of ordered sets, while they still make the relation between variables (indices) and their values explicit. Furthermore, generalized Cartesian products are well-established in mathematical set-theory. Appendix A.2 elaborates on structuring sets and Appendices A.2.1 and A.2.2 define multivariable sets and generalized Cartesian products, respectively.

The functions p, δ, λ, and ta define the actual behavior (dynamics) of an atomic model. The variability of ports of ML-DEVS is made explicit by the *port selection function p* determining which port names, and thus ports, are available (selected) in a given private state. Since not all ports of an atomic model may be available in a certain private state, unavailable ports can neither be used for sending outputs nor for receiving inputs (and thus trigger external state transitions). To capture the variability of ports formally, we define XY as a set of partial

[12] In ML-DEVS, each model (MICRO-DEVS and MACRO-DEVS) holds its own name, i. e., the model's name is part of the model definition. Whereas in other DEVS variants, only coupled models hold a set of names, labels, or references of their components (see Section 4.2).

[13] However, in contrast to state variables, ports are not scalar as they may contain multiple values at once.

[14] also called infinite or arbitrary (Cartesian) products

functions—a partial Cartesian product—that map subsets of port names to admissible values the corresponding ports accept (including the special literal ε). Let $xy \in XY$ be an in- or output and $pn \in \mathcal{P}$, then $xy(pn) = \varepsilon$ indicates that the port with the name pn is available but empty[15].

Remark. The introduction of a partial Cartesian product is a simplification of the notation in Steiniger and Uhrmacher [2016], which allows a more compact definition of Micro-DEVS (and Macro-DEVS). Appendix A.2.3 defines such partial Cartesian products.

As ML-DEVS is a variant of P-DEVS, in which more than one model component can create an output at the same time, we are dealing with bags of in- and outputs rather than single in- and outputs. A bag (or multiset) is a generalization of a set in which elements can occur multiple times[16].

The *time advance function ta* assigns a time interval (lifespan) to each private state, in which an atomic model resides if no *external event* (the model receives an input bag) occurs. If the lifespan has expired and no input bag has been received in the meantime, an *internal event* takes place, i. e., an internal state transition is triggered. If an external event occurs at the very end of the current private state's lifespan, a *confluent event* takes place instead. In both cases, internal event and confluent event, the *output function* λ is invoked right before the actual state transition takes place and creates an output bag for the current private state of the atomic model. Afterwards or when an external event has occurred, the *state transition function* δ is invoked and determines the new private and accessible state of the atomic model. For the clarity of the formalism, only one, general state transition function δ exists in ML-DEVS [Uhrmacher et al. 2007], comparable to the state transition function presented by Zeigler et al. [2000, p. 155]. This state transition function is invoked whenever an atomic model is about to perform an internal, external, or a confluent state transition. Thereby, the modeler has to explicitly distinguish between the different kinds of state transitions when specifying the state transition function δ based upon its arguments[17]. Figure 9.3 illustrates the different kinds of state transitions depending on the elapsed time e and an input bag xy^b, which are passed to the function δ.

The variability of ports has also an impact on the definition of the state transition function δ, which is defined as a *partial function* instead of a *total function*, since not all inputs can occur in each total state $q \in Q$ of the atomic model. For each state of the atomic model, the port selection function determines the set of admissible inputs; hence the domain of the function δ depends on the function p.

Finally, Figure 9.4 shows the input trajectory (top), private and accessible state trajectories (middle), and output trajectory of an exemplary atomic model. For the sake of illustration, inputs, outputs, and states are considered as plain elements of the respective sets, ignoring their structuring according to certain variables. Also note that visualizing in- and output bags as done in the figure is ambiguous, as a bag can contain elements, such as xy_1, not only once but several times. As characteristic for DEVS variants, the input and output trajectories are *event segments*, whereas the two state trajectories are *piecewise constant segments*. A particularity of ML-DEVS is the fact that the accessible state of an atomic model can be altered by the superordinate Macro-DEVS model as a result of *upward causation*—as

[15] Please note the difference between an empty port and an empty input (nonevent). In the former case other ports may not be empty, whereas in the latter case all ports are empty.

[16] Appendix A.1.4 gives more details on and a definition of bags and bag sets.

[17] So unlike Mittal [2013] indicates, there exists not only an external state transition function (i. e., δ_{ext}) in ML-DEVS, instead the state transition function δ combines the external, internal, and confluent state transition functions of other P-DEVS variants. Still, it is reasonable to argue that having only one state transition function makes the definition of this function more verbose and the overall model definition less structured. However, we can easily adapt the model definition of ML-DEVS and its abstract simulator to distinguish between different kinds of state transitions functions.

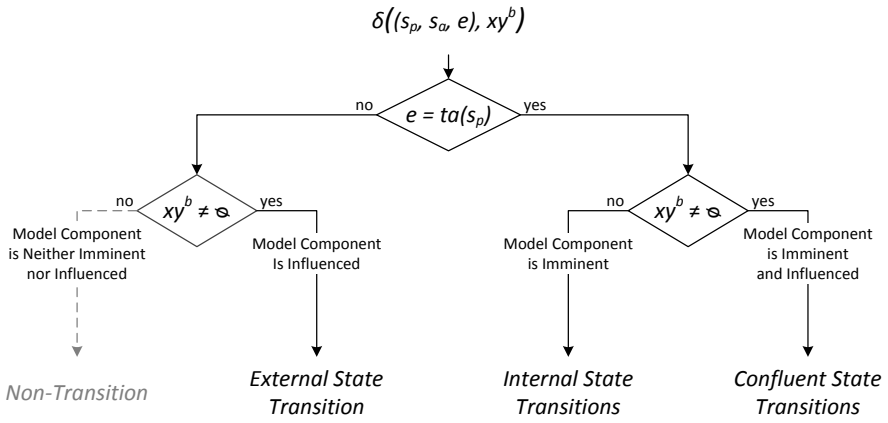

Figure 9.3: Different kinds of state transitions in a MICRO-DEVS model. The abstract simulator of ML-DEVS assures that the first case ($e \neq ta(s_p) \wedge xy^b = \varnothing$) cannot occur (see Section 9.3).

depicted in the Figure 9.4 at simulation time t_3. Section 9.2.2 gives more details on how up- and downward causation are realized in ML-DEVS.

9.2.2 Macro-DEVS Models

MACRO-DEVS models correspond to coupled models (networks of components) of other DEVS variants, because MACRO-DEVS models comprise submodels (components) and allow coupling them. For this reason, we also use the term "coupled model" to refer to a MACRO-DEVS model in the following. However, in contrast to traditional coupled models that merely serve as container for their components, MACRO-DEVS models have a state and behavior of their own [Uhrmacher et al. 2007] and can communicate and interact with their components and vice versa—beyond regular external couplings. A MACRO-DEVS model is formally defined as follows:

Definition 9.2.2 (Macro-DEVS)
A **Macro-DEVS model** is a structure

$$\langle id, XY, S_p, S_a, C, MC, s_{init}, \delta, \lambda, p, ta, sc, \lambda_{down}, v_{down}, act_{up} \rangle$$

where id, XY, S_p, S_a, λ, p, and ta are as in Definition 9.2.1 and where

- C is a superset of potential *components* that are of type MICRO-DEVS;
- MC is a superset of potential *multi-couplings* that are as in Definition 9.2.3;
- s_{init} is the *initial state* with $s_{init} = (s_{p,i}, s_{a,i}, C_{init}, MC_{init})$, where
  - $s_{p,i} \in S_p$ is the *initial private state*,
  - $s_{a,i} \in S_a$ is the *initial accessible state*,
  - $C_{init} \subseteq C$ is a finite set of initially available components (*initial components*),
  - $MC_{init} \subseteq MC$ is a finite set of initially available multi-couplings (*initial multi-couplings*);
- $\delta\colon\ Q \times S^n \times XY^b \rightsquigarrow S_p \times S_a$ is the *state transition function* with Q and XY^b as in Definition 9.2.1 with

$$dom(\delta) = \left\{ ((s_p, s_a, e), s^n, xy^b, a^b) \in Q \times S^n \times XY^b \mid \forall xy \in xy^b\colon\ dom(xy) \subseteq p(s_p) \right\};$$

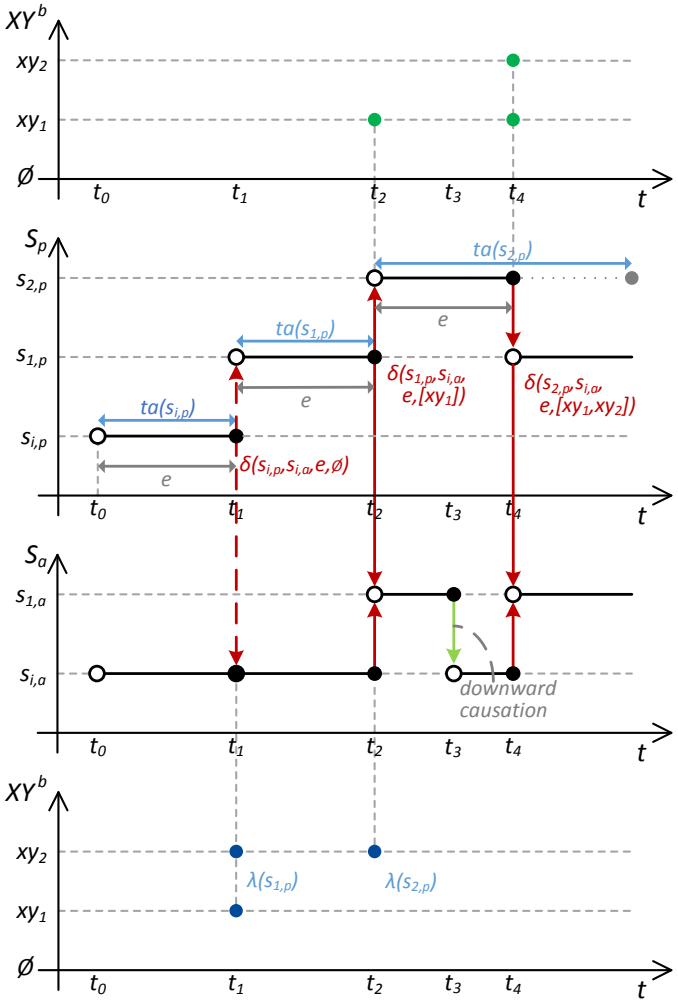

Figure 9.4: Simplified trajectories of a MICRO-DEVS model: (Top) Input trajectory with events at t_2 and t_4. (Middle) State trajectories (public and private) with three state transitions (directed edges). (Bottom) Output trajectory with regular outputs.

- $sc\colon\ S_p \times S^n \to 2^C \times 2^{MC}$ is the *structure change function*;
- $\lambda_{down}\colon\ S_p \times I^n \to XY^n$ is the *downward activation function*, where
  - I^n is the superset of interface instances as in Section 9.2.2,
  - XY^n is defined as below;
- $v_{down}\colon\ S_p \to 2^{\mathcal{V}_p}$ is the *value coupling* downward with $\mathcal{V}_p = variables(S_p)$;
- $act_{up}\colon\ S_p \times S^n \to \mathbb{B}$ with $\mathbb{B} = \{\top, \bot\}$;

and where

- $S^n \subset 2^{\left\{(id,s_a)\,|\,\exists\, c \in n.C\colon\, id=c.id \,\wedge\, s_a \in S_a^c\right\}}$ is the set of all possible *network states* with

$$\forall s^n \in S^n\ \forall\,(id, s_a), (id', s_a') \in s^n\colon\ \left(id = id'\right) \Rightarrow \left(s_a = s_a'\right);$$

- $XY^n \subset 2^{\left\{(id,xy)\,|\,\exists c \in n.C\colon\, id=c.id \,\wedge\, xy \in XY^c\right\}}$ is the set of all possible *network in- and outputs*

with

$$\forall xy^n \in XY^n \; \forall (id, xy), (id', xy') \in xy^n \colon \left(id = id'\right) \Rightarrow \left(xy = xy'\right);$$

and where for each $c \in C$:

- $S_a^c \subset 2^{\left\{(vn,v,X) \mid vn \in variables(c.S_a) \wedge X = range_{vn}(c.S_a) \wedge v \in X \cup \{\varepsilon\}\right\}}$ with

$$\forall s_a^c \in S_a^c \; \forall (vn, v, X), \, (vn', v', X') \in s_a^c \colon \left(vn = vn'\right) \Rightarrow \left(\left(v = v'\right) \wedge \left(X = X'\right)\right).$$

- $XY^c \subset 2^{\left\{(pn,v,X) \mid pn \in variables(c.XY) \wedge X = range_{pn}(c.XY) \wedge v \in X \cup \{\varepsilon\}\right\}}$ with

$$\forall xy^c \in XY^c \; \forall (pn, v, X), \, (pn', v', X') \in xy^c \colon \left(pn \neq pn'\right) \Rightarrow \left(\left(v = v'\right) \wedge \left(X = X'\right)\right);$$

The functions *variables*() and *range*() that take a generalized Cartesian product as argument are defined as in Appendix A.2.2. Furthermore, a Macro-DEVS model is subject to same constraints as a Micro-DEVS model and, in addition, to the following constraints:

1. For all $s_p \in S_p$ and $i^n \in I^n$ holds: $dom(\lambda_{down}(s_p, i^n)) \subseteq dom(i^n)$ (*only available components can be activated*);
2. For all $s_p \in S_p$, $vn \in dom(v_{down}(s_p))$, $s^n \in S^n$, $c \in \pi_1(sc(s_p, s^n))$, and $vn' \in \mathcal{V}_{a,c}$ holds: $(vn = vn') \Rightarrow (range_{vn'}(c.S_a) \subseteq range_{vn}(S_p))$ with $\mathcal{V}_{a,c} = variables(c.S_a)$ (*compatibility of value coupled state variables*);
3. For all $s_p \in S_p$ and $s^n \in S^n$ holds:

$$\begin{aligned} |\pi_1(sc(s_p, s^n))| &= k \\ |\pi_2(sc(s_p, s^n))| &= l \end{aligned} \quad \text{with } k, l \in \mathbb{N}.$$

   So all subsets of components and multi-couplings returned by the structure change function *sc* have to be finite;
4. For all $c \in dom(sc\,[S_p \times S^n])$ holds: $id \neq c.id$ (*no available component has the same identifier as the Macro-DEVS model*).

Let n be a Macro-DEVS model, then we may access the different elements of the structure that defines the model n by writing $n.id$, $n.XY$, $n.S_p$, $n.S_a$, $n.C$, $n.MC$, $n.s_{init}$, $n.\delta$, $n.\lambda$, $n.p$, $n.ta$, $n.sc$, $n.\lambda_{down}$, $n.v_{down}$, and $n.act_{up}$.

If we compare Definition 9.2.1 and Definition 9.2.2 models, it becomes apparent that Micro-DEVS and Macro-DEVS have a number of sets and functions in common. Thereby a Macro-DEVS model can be considered as a combination of a Micro-DEVS model and an extended, traditional coupled model—as known from other DEVS variants—that is equipped with further functions to carry out structure changes and realize up- and downward causation. A Micro-DEVS model, on the other hand, can be viewed as a stunted version of a Macro-DEVS model, i. e., one without components, couplings, and up- and downward causation. Consistently, one could argue that we do not need Micro-DEVS models, as we can use solely Macro-DEVS models instead. However, as "syntax matters" (cf. Henzinger et al. [2009] or Winsberg [2009]), we distinguish between Micro-DEVS and Macro-DEVS models, which eases the modeling and leads to more compact, less cluttered model specifications. In addition, the distinction between two different types of models[18] is important for a sound and rigorous definition of the modeling formalism, as we discuss in Section 9.4.

[18] No matter whether we distinguish between atomic and coupled ML-DEVS models or between coupled ML-DEVS models with and without components.

In addition to the sets and functions already defined for a MICRO-DEVS model (with the exception of the state transition function δ that is defined differently for MACRO-DEVS), a coupled ML-DEVS model is also defined by: (i) the sets C and MC and (ii) the functions sc, λ_{down}, v_{down}, and act_{up}. The sets S^n and XY^n that were introduced in Steiniger and Uhrmacher [2016] as well as the new sets $\{S_a^c \mid c \in C\}$ and $\{XY^c \mid c \in C\}$ do not need to be specified by the modeler explicitly, instead they are derived according to Definition 9.2.2. The set S^n contains sets of pairs of unique component identifiers and sets of accessible state variable assignments of the associated components, denoted by S_a^c. Similar to Uhrmacher [2001], we call an element of S^n the *network state*, although S^n only covers the accessible states of the components but not their private state. The set XY^n contains sets of pairs of unique component identifiers and sets of value assignments to available ports of the associated components, denoted by XY^c. Thus, XY^n captures the availability of ports. During simulation, for each private state $s_p \in S_p$ of the MACRO-DEVS model, $s^n \in S^n$ and $xy^n \in XY^n$ are derived based on the state transitions and port selection functions of the respective components (see Section 9.3.2). Also the set I^n, which denotes the superset of interface instances, does not need to be specified. In the following subsections we will explain the functions and remaining sets of a MACRO-DEVS model in more detail.

Variable Composition

The set C (in Definition 9.2.2) contains all potential components (submodels) of a coupled ML-DEVS model, which can be available during simulation. However, only a finite subset of these potential components can be available at all time, otherwise a coupled model would not be executable on a computer with limited resources. The structure change function sc determines which components[19] and multi-couplings are available. In contrast to our previous work on ML-DEVS, e. g., Uhrmacher et al. [2007] or Steiniger et al. [2012], the structure change function takes, in addition to the current private state of the coupled model $s_p \in S_p$, also the corresponding network state $s^n \in S^n$ into account. This allows the modeler to adapt the network structure without the necessity of a "detour via the private state of the coupled model" by reflecting certain structure-related information in the private state of the coupled model[20], but directly based upon the network state[21].

Since the structure change function sc maps into a Cartesian product that includes the power set of C and C is part of the defining tuple of a MACRO-DEVS model, the definition of C itself is, also from a formal point of view, of particular interest. Insights we gained from creating models in computational systems biology (e. g., a model of self-replicating mitochondria) let us conclude that an extensional definition of C (listing all components explicitly and in advance) does not cope well with the desired flexibility of "creating components on demand," during simulation. Instead, we propose an intensional definition of C. In the most general case, the set C contains all consistent MICRO-DEVS and MACRO-DEVS models (the latter due to the closure under coupling of ML-DEVS) and thus does not need to be specified at all[22]. However, from the modeler's perspective it is desirable to constrain the kind of submodels that can be created (e. g., only eukaryotic cells have mitochondria). This, however, requires a notion of different types or classes of models, in addition to the traditional distinction between atomic and coupled models; for instance, as discussed by Uhrmacher [1995] or similar to the ideas of aspects and specializations in the System Entity Structure (SES) [Rozenblit & Zeigler

[19] With this regard, the structure change function sc of ML-DEVS is comparable to the network transition function ρ_N in dynDEVS and ρ-DEVS or the γ-function in DSDE and its extensions.

[20] As done in, e. g., Barros [1997].

[21] The size of the overall state space of the MACRO-DEVS model remains unaffected by this design decision, as the overall state of a MACRO-DEVS model includes its private state as well as the corresponding network state (see Section 9.4).

[22] In this case C can be viewed as a universe or universal set of all possible, consistent ML-DEVS models.

1993; Zeigler 1984; Zeigler & Hammonds 2007]. Distinguishing between such different classes of models is subject of future work and briefly discussed in Section 11.2.

For the execution of a coupled model, the initial composition (i. e., the initially available components) has to be made explicit by the modeler. This is done by specifying the set C_{init}, which is part of the initial state of the coupled model. Note that the initially available components are not determined by calling the structure change function sc with the initial private state of the MACRO-DEVS model. In fact, the available components returned by the function sc for the initial private state and the corresponding network state can differ from those in the set C_{init}.

Interfaces and Interface Instances

Interfaces and their instances are central for realizing the concept of intensional couplings between model components (horizontal couplings), as proposed in Chapter 8. In addition, interfaces and interface instances are used for achieving up- and downward causation (vertical couplings) in ML-DEVS. Traditionally, the interface of a DEVS model (atomic or coupled) consists of the set of inputs X and the set of outputs Y of the model[23]. In the case the model is defined at the level of structured systems, we often refer to input and output ports when talking about the interface of a model. In ML-DEVS, the interface of MICRO-DEVS and MACRO-DEVS model comprises its identifier, accessible state variables, and ports.

Let c be a ML-DEVS model, MICRO-DEVS or MACRO-DEVS, with

$$c = \langle id, XY, S_p, S_a, s_{init}, p, \delta, \lambda, ta \rangle$$

or

$$c = \langle id, XY, S_p, S_a, C, MC, s_{init}, \delta, \lambda, p, ta, sc, \lambda_{down}, v_{down}, act_{up} \rangle,$$

where $c.id$ is the model's identifier, $c.XY$ is a structured set of the model's in- and outputs, and $c.S_a$ is a structured set of the model's accessible states and where $c.XY$ and $c.S_a$ are defined as generalized Cartesian products according to Definition 9.2.1. Following our terminology proposed in Chapter 8, the ports and accessible state variables are the *attributes* of the model component c that shall be made accessible to the outside via the interface of c and that can be derived from the structured sets $c.XY$ and $c.S_a$, respectively. As we distinguish between two types of attributes: ports and accessible state variables, we define the set of interface attributes of the model c, denoted by $Attr^c$, as follows:

$$Attr^c \triangleq Attr^c_p \oplus Attr^c_a \tag{9.1}$$

with

$$Attr^c_p = \{(an, X) \mid an \in variables(c.XY) \wedge X = range_{an}(c.XY)\},$$
$$Attr^c_a = \{(an, X) \mid an \in variables(c.S_a) \wedge X = range_{an}(c.S_a)\},$$

where $Attr^c_p$ and $Attr^c_a$ refer to the ports and accessible state variables of the model c, respectively, and where $Attr^c_p \oplus Attr^c_a$ denotes the *disjoint union* of the sets $Attr^c_p$ and $Attr^c_a$. Because of the disjoint union of two "subsets" of attributes, the model c can have accessible state variables and ports with the same names. Please note that this is a relaxation of the definition of a plain set of attributes in Chapter 8, which however ease the model specification. From the above definitions of $Attr^c_p$ and $Attr^c_a$ follow:

$$dom(Attr^c_p) = variables(c.XY),$$
$$dom(Attr^c_a) = variables(c.S_a).$$

[23] cf. Section 4.2

Given a ML-DEVS model c, the interface of c, denoted by i^c, would, according to Definition 8.5.1, be defined as follows:

$$i^c \triangleq (c.id, Attr^c)\,,$$

where $Attr^c$ is the set of interface attributes of c as defined in Equation 9.1. The identifier of the interface is the identifier of the model c itself, implying that each model has its own interface. This is because ML-DEVS does not bother the modeler with specifying interfaces explicitly, instead ML-DEVS derives the interface of each model from its defining tuple individually[24].

According to Definition 8.6.2, an instance of an interface assigns actual values to all interface attributes, which are available at a certain instant of simulation time (in which this instance exists). In ML-DEVS, this corresponds to assigning values to the available ports and all accessible state variables. Similar to the set of interface attributes $Attr^c$ of the model c, we now define the set of assignments for $Attr^c$, denoted by $Assg^{Attr^c}$, such that the set consists of two "subsets," i. e.,

$$Assg^{Attr^c} \triangleq Assg^{Attr^c_p} \oplus Assg^{Attr^c_a} \tag{9.2}$$

with

$$Assg^{Attr^c_p} \subset \left\{(an, v, X) \,|\, (an, X) \in Attr^c_p \wedge v \in X \cup \{\varepsilon\}\right\},$$
$$Assg^{Attr^c_a} \subset \left\{(an, v, X) \,|\, (an, X) \in Attr^c_a \wedge v \in X \cup \{\varepsilon\}\right\},$$

where

$$\forall asg, asg' \in Assg^{Attr^c_p}\colon \left(asg.id = asg'.id\right) \Rightarrow \left(\left(asg.v = asg'.v\right) \wedge \left(asg.X = asg'.X\right)\right),$$
$$\forall asg, asg' \in Assg^{Attr^c_a}\colon \left(asg.id = asg'.id\right) \Rightarrow \left(\left(asg.v = asg'.v\right) \wedge \left(asg.X = asg'.X\right)\right).$$

Based on Definition 8.6.2, an instance of the interface i^c of a model c, denoted by ii^c, is then defined by

$$ii^c \triangleq \left(c.id, c.id, Assg^{Attr^c}\right), \tag{9.3}$$

where $Assg^{Attr^c}$ is defined as in Equation 9.2. As model and interface identifier are equal and interfaces do not need to be explicitly defined in ML-DEVS, we define, for notational convenience and consistency, the interface instance ii^c of a ML-DEVS model c also directly based on the defining tuple of c as follows:

$$ii^c \triangleq (c.id, (xy^c, s^c_a)) \tag{9.4}$$

with $xy^c \in XY^c$ and $s^c_a \in S^c_a$ and XY^c and S^c_a as in Definition 9.2.2. If we compare the above definitions of xy^c and s^c_a with the definitions of $Assg^{Attr^c_p}$ and $Assg^{Attr^c_a}$ carefully, it becomes apparent that we can transform xy^c and s^c_a into $Assg^{Attr^c_p}$ and $Assg^{Attr^c_a}$, respectively, and vice versa.

Now for defining intensional couplings, i. e., multi-couplings, in a coupled model n with

$$n = \langle id, XY, S_p, S_a, C, MC, s_{init}, \delta, \lambda, p, ta, sc, \lambda_{down}, v_{down}, act_{up}\rangle,$$

based upon the interface instances of n and its components, we define the superset of all possible sets of interface instances, denoted by I^n, as follows:

$$I^n \subset 2^{\{ii^c \,|\, c \in n.C \cup \{n\}\}} \tag{9.5}$$

[24] As we will see later, ML-DEVS directly makes use of interface instances without deriving interfaces first.

with ii^c as defined in Equation 9.4 and

$$\forall i^n \in I^n \; \forall \big(id, \big(xy^c, s_a^c\big)\big), \big(id', \big(xy^{c\prime}, s_a^{c\prime}\big)\big) : \; \big(id = id'\big) \Rightarrow \Big(\big(xy^c = xy^{c\prime}\big) \wedge \big(s_a^c = s_a^{c\prime}\big)\Big) . \quad (9.6)$$

Pleas note that we denoted the superset of interface instances by $\mathscr{I}_C$ or I_C in our previous publications on ML-DEVS. However, to keep consistency with other names of sets used for defining Macro-DEVS, we changed the name to I^n, where n is the definition of the corresponding Macro-DEVS model from which the superset is derived.

Multi-Couplings

A direct consequence of the variable composition of ML-DEVS is the necessity to adapt the communication structure (i. e., the coupling scheme) accordingly to preserve *structural consistency* (i. e., a consistent model specification). This necessity is addressed by:

1. The intensional multi-couplings and their translation into an transitory, concrete coupling scheme;
2. The structure change function *sc* that can add and remove multi-couplings.

Taking (i) the variable composition of a coupled model and (ii) the changing interface instances of the coupled model and its components into account, the *horizontal communication* between the components of a coupled model (*inter-level communication*) is enabled and constrained by a flexible coupling mechanism called multi-couplings. Multi-couplings[25] are an implementation of the ideas of an intensional coupling definition and intensional coupling functions based on interface instances as proposed in Chapter 8. Multi-couplings cover both internal couplings and external couplings[26], which are usually distinguished in DEVS variants (especially in those that employ *port-to-port couplings*). Due to their intensional definition, multi-couplings have to be evaluated and translated into concrete coupling schemes during simulation, whenever the model composition or interfaces are changing. Each of the derived coupling schemes captures one particular structural context, i. e., one incarnation of the composition including the concrete interface instances of all available components. Note that although the composition of a coupled model may not change between two instants of simulation time, the interface instances of the involved components can. The evaluation and translation of multi-couplings into concrete coupling schemes is done by the abstract simulator of ML-DEVS and explained in more detail in Section 9.3.2. Informally we can summarize the translation of multi-couplings as follows: The availability of components with properties (including the component's name and ports) defined in a multi-coupling at a certain time in a simulation implies the existence of concrete couplings between available and compatible ports (i. e., port-to-port couplings); into which multi-couplings are eventually translated. These concrete couplings are then used by the abstract simulator of ML-DEVS to forward events. The availability of ports is determined by the port selection functions of the coupled model and those of its components, whereas the availability of components is determined by the structure change function *sc*.

The original definition of multi-couplings given by Uhrmacher et al. [2007] and modified in Steiniger et al. [2012] was based solely on port names, meaning that concrete ports were coupled if their names match those in the coupling definitions and their value ranges are

[25] We call the intensional coupling functions in ML-DEVS multi-couplings due to historical reasons. Uhrmacher et al. [2007] adopt the term from ρ-DEVS [Uhrmacher et al. 2006]. In contrast to classic 1:1 couplings as known from the real of DEVS, a single *multi-coupling* can represent multiple concrete couplings between different components at the same time.

[26] Internal couplings are those between the components of a coupled model, whereas external couplings are those from the coupled model to its components (*external input couplings*) or from the components to their coupled model (*external output couplings*). As external couplings link the components of a coupled model with the coupled model and vice versa, external couplings can be viewed as a special kind of vertical coupling.

"compatible." No further constraints could be made by the modeler and exclusive couplings between two components could only be achieved by using port names that were globally unique[27]. In Steiniger and Uhrmacher [2016], we adapt this rather generic but still intensional definition to come up with a more expressive and powerful mechanism to specify couplings within variable structure models intensionally, following Definition 8.7.1.

Definition 9.2.3 (Multi-Coupling)
Let n be a MACRO-DEVS model with

$$n = \langle id, XY, S_p, S_a, C, MC, s_{init}, \delta, \lambda, p, ta, sc, \lambda_{down}, v_{down}, act_{up} \rangle,$$

where for each $c \in C$, c is defined as a MICRO-DEVS model as in Definition 9.2.1. Following Definition 8.7.1, a **multi-coupling** $mc \in MC$ is defined as a function

$$mc\colon\ I^n \rightarrow 2^{Cplg^n_{ext}} \text{ with } Cplg^n_{ext} = (\mathcal{M}^n \times \mathcal{P}^n) \times (\mathcal{M}^n \times \mathcal{P}^n),$$

with I^n as defined in Equation 9.5. Each tuple

$$((id_s, pn_s), (id_t, pn_t)) \in Cplg^n_{ext}$$

corresponds to a concrete, directed port-to-port coupling, which may exist during simulation, with id_s and pn_s being the names of the source component and source port and id_t and pn_s being the names of the target component and target port. The sets $\mathcal{M}^n \subseteq \mathcal{N}$ and $\mathcal{P}^n \subseteq \mathcal{N}$ denote the sets of potential model identifiers and port names (i. e., a namespace), respectively, which can be used to define concrete couplings upon, where

$$\mathcal{M}^n = \bigcup_{c \in \{n\} \cup n.C} \{c.id\},$$
$$\mathcal{P}^n = \bigcup_{c \in \{n\} \cup n.C} variables(c.XY).$$

Accordingly, the set $Cplg^n_{ext}$ denotes the superset of all possible port-to-port couplings (i. e., extensional couplings) that can be defined for the model n.

In a nutshell, a multi-coupling is an unary function that maps a set of incarnations of the interfaces of the MACRO-DEVS model and its components to a set of potential port-to-port couplings, which may or may not exist due to certain consistency constraints. These constraints are necessary as port-to-port couplings relate ports simply based on their names, but do not make any statements about the coherence of the ports' value ranges. Similar to Zeigler et al. [2000, p. 86 and 130], we formulate certain requirements for the consistency of port-to-port couplings that are returned by a multi-coupling. However, we can asses the consistency of a port-to-port coupling only with respect to a certain structural context but not in general, as the availability of components and their ports can change. A concrete structural context is given by a set of interface instances of a MACRO-DEVS model and its components, i. e., $i^n \in I^n$.

Definition 9.2.4 (Consistency of a Concrete Coupling)
Let n be a MACRO-DEVS model, $mc \in n.MC$ be a multi-coupling of n, and $i^n \in I^n$ be a set of interface instances of n and its components, where I^n is defined as in Equation 9.5, then a port-to-port coupling, denoted by $cplg$, with $cplg \in mc(i^n)$ and

$$cplg = ((id_s, p_s),\ (id_t, p_t)),$$

[27] or at least unique for a pair of micro and macro level

is called **consistent**, denoted by $consistent(i^n, cplg)$, if:

1. $id_s, id_t \in dom(i^n)$ (*respective components are available*),
2. $id_s \neq id_t$ (*no direct feedback loop to prevent algebraic loops*),
3. $pn_s \in dom(xy^c)$ with $(id_s, (xy^c, s_a^c)) \in i^n$
4. $pn_t \in dom(xy^{c\prime})$ with $(id_t, (xy^{c\prime}, s_a^{c\prime})) \in i^n$

   (*respective ports are available*),
5. $X \subseteq X'$ with $(pn_s, v, X) \in xy^c$ and $(id_s, (xy^c, s_a^c)) \in i^n$ and with $(pn_t, v', X') \in xy^{c\prime}$ and $(id_t, (xy^{c\prime}, s_a^{c\prime})) \in i^n$ (*subset relation holds, i.e., ports are compatible*).

Note that we do not require that all port-to-port couplings returned by a given multi-coupling have to be consistent, instead only consistent couplings are considered when deriving a transitory, concrete coupling scheme according to which events are exchanged. However, we can argue that a violation of the last requirement in Definition 9.2.4 (subset relation) may indicate a poor model design or even a faulty model, about which the modeler should be informed. Such a feedback can be given by a suitable model editor or modeling environment that supports ML-DEVS.

Although the overall communication structure of a coupled model can, in principle, be defined by a single multi-coupling, we decided to derive the concrete coupling schemes from a number of multi-couplings—the set MC. This set can still be a singleton, if desired. The motivation for allowing the modeler to define a set of multi-couplings is threefold:

1. Representing an entire coupling scheme by only one multi-coupling may be impractical due to the resulting complexity of this very multi-coupling, especially in complex systems with divers communication paths.
2. Breaking the coupling scheme down to several multi-couplings allows reducing the complexity of the individual multi-couplings (*divide and conquer*), where each multi-coupling can focus on a certain aspect of the overall coupling scheme (*separation of concerns*).
3. The modeler can easily extend or curtail the coupling scheme by adding or removing multi-couplings during simulation via the structure change function sc. Thus the modeler neither has to consider nor resolve every possible ambiguity that can result from defining contradicting multi-couplings, if such multi-couplings are not active at the same time.

The structure change function sc allows the modeler to add and remove multi-couplings in addition to change the availability of components during simulation. Only active multi-couplings are considered, when deriving a concrete coupling scheme. However, a well thought out definition of the multi-couplings and their intensional character should make it only seldom necessary to add or remove multi-couplings. In other variable structure variants of DEVS such as DSDEVS, DSDE, or dynDEVS, we have to define a consistent coupling scheme for each incarnation of a network (i. e., structural context) extensionally. In ML-DEVS, on the other hand, a consistent model specification is not the result of the structure change function (i. e., adding and removing multi-couplings) but the intensionality of multi-couplings and their translation into concrete, consistent coupling schemes during simulation.

For a given private state s_p of a coupled model n and the corresponding structural context that is encoded in the set of current interface instances $i^n \in I^n$, the concrete coupling scheme, denoted by $Cplg_{conc}$, is defined by the union of all consistent extensional couplings returned by the currently available multi-couplings, i. e.,

$$Cplg_{conc} = \bigcup_{mc \in \pi_2(sc(s_p))} \left(\bigcup_{cplg \in \{cplg' \in mc(i^n) \mid consistent(i^n, cplg')\}} \{cplg\} \right) \tag{9.7}$$

The following example illuminates how the output of a multi-coupling can be compactly and flexibly specified based on the multi-coupling's argument, by using a *set-builder notation*.

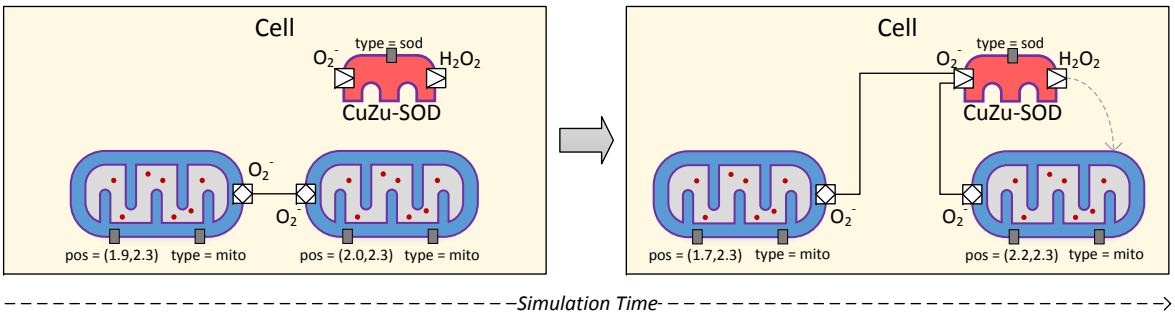

Figure 9.5: If mitochondria are close to each other they can inhibit each other by exchanging O_2^-. Otherwise, O_2^- is reduced to H_2O_2 by the antioxidant enzyme CuZn-SOD.

Example 9.2.1 (Multi-Coupling)
Damaged mitochondria produce and release reactive oxygen species (ROS), e. g., O_2^- (superoxide anion), which inhibit nearby mitochondria and potentiate the mitochondria-driven ROS propagation Park et al. [2011]. However, there are also antioxidant enzymes in the cell, such as CuZn-SOD or Gpx1, which can reduce ROS before they reach other mitochondria, especially when mitochondria are distant. Now we want to specify a distance-based coupling that connects mitochondria either with others if their distance is below a certain threshold or with the enzyme CuZn-SOD that reduces O_2^- to H_2O_2 (hydrogen peroxide). Such a coupling could look like the following:

$$\begin{aligned} mc(i^n) = \big\{((id_s, pn_s), (id_t, pn_t)) \mid \exists (id_s, (xy, s_a)), (id_t, (xy', s_a')) \in i^n \; \exists a_t, a_p \in dom(s_a) \\ \exists a_t', a_p' \in dom(s_a') \; \exists pn_s \in dom(xy) \; \exists pn_t \in dom(xy') \colon \; pn_s = pn_t = \text{“}O_2^-\text{”} \\ \wedge \big((a_p = a_p' = \text{“pos”} \wedge dist(s_a(a_p), s_a'(a_p')) \leqslant 0.1) \\ \vee (a_t = \text{“mito”} \wedge a_t' = \text{“sod”})\big)\big\}, \end{aligned}$$

where i^n is a subset of I^n of the coupled model “Cell” and the binary function $dist(\ldots)$ calculates the Euclidean distance between two two-dimensional coordinates. Figure 9.5 shows two concrete coupling schemes (left and right) that can be derived from the above multi-coupling. Note that a direct feedback loop between one and the same mitochondria is prevented when translating the multi-coupling into concrete couplings (see Section 9.3.2).

Although still rather “simple,” Example 9.2.1 gives us an impression about the potential and expressivity of the novel definition of multi-couplings. A single multi-coupling can encode an arbitrary number of n:m port-to-port couplings, while incorporating the structural variability and other information at the same time. In contrast to Uhrmacher et al. [2007] and Steiniger et al. [2012], where multi-couplings are only defined based upon port names, our revision of multi-couplings allows establishing exclusive couplings between specific components more easily, by using their identifiers (i. e., names) to discriminate between components. Furthermore, we can call special domain-dependent selection-functions as introduced by Uhrmacher et al. [2006] that select a subset from a set of potential target components. By doing so, we can model the exchange of consumable resources rather than the classic information broadcast in DEVS.

Up- and Downward Causation

In addition to a flexible definition of horizontal couplings between the components of a coupled model (by the means of multi-couplings), ML-DEVS allows expressing vertical interdependencies between different levels of behavior. In systems biology, these inter-level dependencies are typically referred to as *up- and downward causation*, e. g., by Campbell [1974]. ML-DEVS provides, on top of multi-couplings, mechanisms for modeling up- and

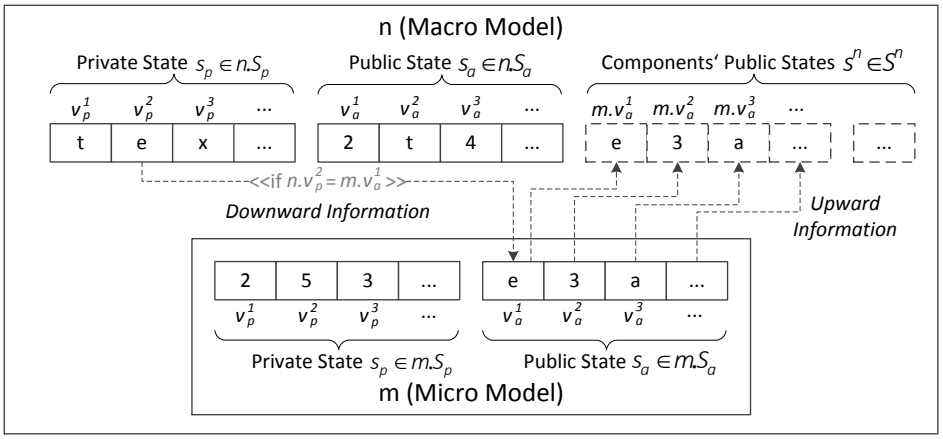

Figure 9.6: State sharing in ML-DEVS: The macro model can read the accessible states of its components (upward information) and assign values to some accessible state variables of its components via value coupling in return (downward information).

downward causation between a macro level and the components at its micro level explicitly. To the best of our knowledge, this kind of "vertical coupling" cannot be found in other DEVS variants and is, along with states and behavior at each level of the organizational hierarchy, a prerequisite for multi-level modeling, beyond traditional hierarchical modeling. However, due to the modularity of ML-DEVS, i. e., only the interfaces of ML-DEVS models are accessible, multi-level modeling is inevitably more restrictive than in other non-modular multi-level modeling languages, such as ML-Rules [Maus et al. 2011] and ML-Space [Bittig, Haack, Maus, & Uhrmacher 2011; Bittig, Matschegewski, Nebe, Stählke, & Uhrmacher 2014]. Therefore, we cannot model interdependencies across arbitrary levels, instead up- and downward causation is defined between a macro level and the associated micro level (cf. Figure 9.2).

Up- and downward causation are split into *information transfer* and *activation*, which leads to the following four schemes of inter-level interdependencies:

1. Upward information;
2. Upward activation;
3. Downward information;
4. Downward activation.

The two information schemes establish a kind of *state sharing*, where the accessible states of the components at the micro level are accessible by the corresponding Macro-DEVS model and private state variables of the Macro-DEVS model are readable by its components at the micro level (see Figure 9.6). In a more traditional sense, this state sharing can be viewed as a non-modular interaction between components and the Macro-DEVS model, as the state transitions of the components affect the overall state of the Macro-DEVS model and vice versa. This relation becomes more apparent in Section 9.4.

Upward Information is the ability of a Macro-DEVS model to read the accessible states of its components, which constitute the set S^n in Definition 9.2.2. The accessible state of a component is determined by its state transition function δ and part of the component's interface instance (see Section 9.2.2). The Macro-DEVS model makes use of the accessible states of its components in the upward activation function act_{up}, the structure change function sc, the state transition function δ, and the downward activation function λ_{down} (the accessible states of the components are included in the set I^n). The abstract simulator of ML-DEVS ensures that for each private state $s_p \in S_p$ of the Macro-DEVS model, only the available components are considered in $s^n \in S^n$.

Example 9.2.2 (Upward Information)
As indicated in Example 9.2.1, the distance between mitochondria is of interest, when, e. g., examine the mitochondria-driven ROS propagation. To determine the (Euclidean) distance between two mitochondria, their position (or location) need to be known to the superordinate coupled model. Instead of sending the coupled model its position whenever it changes, a mitochondrion can declare its position as an accessible state variable that can then be accessed by the corresponding coupled model. So we add an indexed set denoted by S_{pos} with

$$S_{pos} = \mathbb{R} \times \mathbb{R}$$

and $pos \in \mathcal{N}$ to the indexed family of sets based upon which the set S_a of a mitochondrion is defined. In addition, the name "pos" becomes a part of the set $\mathcal{V}_a$.

Upward Activation refers to the ability of the micro level to initiate changes at the macro level by changing accessible state variables; but not ports as in our previous work [Steiniger et al. 2012; Uhrmacher et al. 2007]. In a Macro-DEVS model, upward activation is realized by the *upward activation function* act_{up}. The function act_{up} guards the fulfillment of *invariants* defined on the components' accessible states. If one or more invariants are violated, the function act_{up} returns '$\top$' (*activation event*) and triggers as a result the Macro-DEVS model, i. e., its state transition function is called. An activation event can represent one or more incidents (requests) such as the request to add or remove a component, which requires a specific reaction of the Macro-DEVS model in return, e. g., a structure change. Such a structure change is carried out by the structure change function sc, which is called after the Macro-DEVS model has performed a state transition.

Example 9.2.3 (Upward Activation)
Based on the model described by Patel et al. [2013], suppose the macro model "Cell" shall become active whenever the health of more than the half of its mitochondria drops below a threshold. Each mitochondrion announces its current health status via an accessible state variable "health," which can take values between 0.0 and 1.0. A corresponding upward activation function can be specified by:

$$act_{up}(s_p, s^n) = \begin{cases} \top & \text{if } |D| \geqslant 0.5 * |dom(s^n)| \text{ with } D = \{id \mid \exists (id, s_a) \in s^n \\ & \exists v \in dom(s_a) \colon v = \text{“health”} \wedge s_a(v) \leqslant 0.5\} \\ \bot & \text{otherwise.} \end{cases}$$

Downward Information is the ability of a Macro-DEVS model to make certain information readable for its components, via a mechanism called *value coupling* [Uhrmacher et al. 2007]. For a given private state of the Macro-DEVS model, the *value coupling function* v_{down} determines private state variables of the Macro-DEVS model actual values of which should be readable as long as the Macro-DEVS model is in the state. If a component of the Macro-DEVS model has an accessible state variable whose name matches that of a readable private state variable of the Macro-DEVS model, the value of this private state variable is assigned to the matching accessible state variable of the component (see Figure 9.6). Therefore, the range of values that can be assigned to the component's accessible state variable has to be a subset of the value range of the corresponding private state variable of the Macro-DEVS model (see the respective constraint in Definition 9.2.2). Affected components can afterwards use this "global information" when determining their new states (cf. global attributes in John, Lhoussaine, and Niehren [2009]). However, value coupling itself does not trigger the components at the micro level, i. e., leads to state transitions in the components. In contrast

to Uhrmacher et al. [2007], value coupling is no longer defined on ports but on state variables. Thus it is not necessary to formulate the constraint that a port that is used for value coupling cannot be an input port. Additionally, value coupling can change during simulation in our revision of ML-DEVS. For instance, when a Macro-DEVS model changes its private state, a private state variable that was readable before the state change occurred may not be readable afterward.

Example 9.2.4 (Downward Information)
In the case mitochondria are modeled as coupled models themselves, comprising components of their own (e. g., DNA, ATP, or granules[a]), certain information about the mitochondrion may be of interest for its components. Suppose we want to establish the private state variables "health" and "volume" of a mitochondrion as global information that is accessible by the components of the mitochondrion, as long as it alive. We can achieve this by defining the function v_{down} of a mitochondrion as follows:

$$v_{down}(s_p) = \begin{cases} \{\text{“health”, “volume”}\} & \text{if } s_p(\text{phase}) = \text{“alive”} \\ \varnothing & \text{otherwise.} \end{cases}$$

Note that components of such a mitochondrion have only access to the global information, if they have accessible state variables with matching names and value ranges.

[a] small particles that are visible by a microscope

Downward Activation is the ability of a Macro-DEVS model to initiate state changes at its micro level based on the Macro-DEVS model's current private state and a set of interface instances of all the components of the Macro-DEVS model. To initiate state changes, the *downward output function* λ_{down} allows the Macro-DEVS model to create inputs for its components, denoted by XY^n in Definition 9.2.2, by directly accessing their ports. The result is a simultaneous activation of the corresponding components. The interface instance of each available component indicates which ports are selected and thus can be used for downward activation. As a set of the interface instances of all available components is an argument of the function λ_{down}, the modeler can directly access the information which ports are currently available when specifying λ_{down}.

9.2.3 Consistency of Model Specifications in ML-DEVS

In DEVS and most of its variants we find none or only few constraints regarding a consistent and correct model specifications, such as the subset-relation between coupled ports when models are defined at the level of structured systems [Zeigler et al. 2000, p. 130]. In contrast, when defining Micro-DEVS models (Definition 9.2.1) and Macro-DEVS models (Definition 9.2.2), we formulate a number of different constraints, mainly as a result of up- and downward causation and the variability of the composition and ports and inter-level couplings. Since it is somehow easy for a modeler to specify an inconsistent models (e. g., by using ports for generating outputs that are actually not available in the current state of the model), consistency plays a more prominent role in ML-DEVS than in other DEVS variants. However, it is impractical or not possible to check whether or not certain constraints hold before the actual model execution, in particular those regarding the domains of the characteristic functions. The constraints that can be checked before the model execution, should be monitored by the modeling environment that supports ML-DEVS to inform the modeler about potential design flaws.

As we will see in the next section, the abstract simulator of ML-DEVS takes care about the adherence of most of the constraints, formulated in this section, and the translation of

the intensional multi-couplings into concrete couplings. So even if a model has violated a constraint that does not necessarily mean that the model cannot be executed properly.

9.3 Abstract Simulator of Multi-Level DEVS

ML-DEVS describes discrete event systems[28], i. e., systems whose state only changes at discrete points in continuous time and remains constant in the meantime. As characteristic for DEVS variants, an *abstract simulator* (a simulation algorithm [Zeigler et al. 2000, p. 26]) specifies the *execution semantics* of ML-DEVS, in an operational manner[29]. Thereby the abstract simulator defines how a model specified in the formalism is executed, i. e., how (state and output) trajectories are produced according to a given model specification and an input trajectory resulting from this specification. The simulator is abstract because it provides information about what has to be done to execute a model, but not necessarily how it has to be done exactly [Zeigler et al. 2000, p. 176] on a certain target machine. This makes an abstract simulator technology and platform agnostic. In fact, an abstract simulator can be implemented rather differently (cf. Himmelspach and Uhrmacher [2006]).

Similar to other DEVS variants, the abstract simulator of ML-DEVS consists of three types of (so-called) processors[30]: Simulators, Coordinators, and a Root-Coordinator.

Remark. Please note that despite the ambiguous naming, a processor of type Simulator is only a part of the abstract simulator of ML-DEVS and not the abstract simulator itself.

Simulators are responsible for executing Micro-DEVS models, whereas Coordinators are responsible for executing Macro-DEVS models. The Root-Coordinator initiates and controls the simulation cycles (simulation steps) and keeps track of the time elapsed during simulation (simulation time). The processors are arranged in a tree: the *processor tree*, which reflects the compositional hierarchy of the model to be executed. From a given, hierarchical ML-DEVS model that is composed of atomic and coupled models we derive the corresponding processor tree by associating a Simulator with each atomic model and a Coordinator with each coupled model. Thereby, Coordinators form the inner nodes of the resulting processor tree, whereas Simulators form the leaves of the tree. The root of the processor tree is the Root Coordinator, which has only one child node: the processor associated with the topmost model component. As the model composition in ML-DEVS can change during simulation, the corresponding processor tree has to change accordingly. Figure 9.7 shows the mapping between two incarnations of a hierarchical ML-DEVS model and the associated processor trees (left and right side of the figure).

The processors of the abstract simulator communicate top-down and bottom-up throughout the processor tree by exchanging a predefined set of messages each simulation cycle. Table 9.1 lists the different kinds of messages that all together constitute the *communication protocol* (also called *simulator protocol*) of ML-DEVS. Figure 9.8 shows the communication protocol with its different messages and their content. The depicted protocol adapts that of P-DEVS as presented by Himmelspach and Uhrmacher [2006] and extends the `i-`, `y-`, `x-`, and

[28] This does not mean that we can use ML-DEVS only for modeling discrete event systems. We can also represent or approximate a continuous system by a discrete event system, e. g., as a *quantized state system* (QSS) [Cellier & Kofman 2006, pp. 542–8].

[29] The execution semantics is operational because the abstract simulator can be viewed as an *abstract machine* whose state and behavior is defined in terms of the transition and other functions of ML-DEVS (cf. *operational semantics* as described by Pierce [2002, pp. 32–3]). However, the abstract simulator is not described by using the *structural operational semantics*, which is a well-known and established approach to describe the operational semantics of programs or languages introduced by Gordon D. Plotkin (see, e. g., Plotkin [2004a] or Plotkin [2004b]).

[30] In this context, a processor can be viewed as a virtual processing unit, e. g., an algorithm, that is responsible for processing a well-defined task.

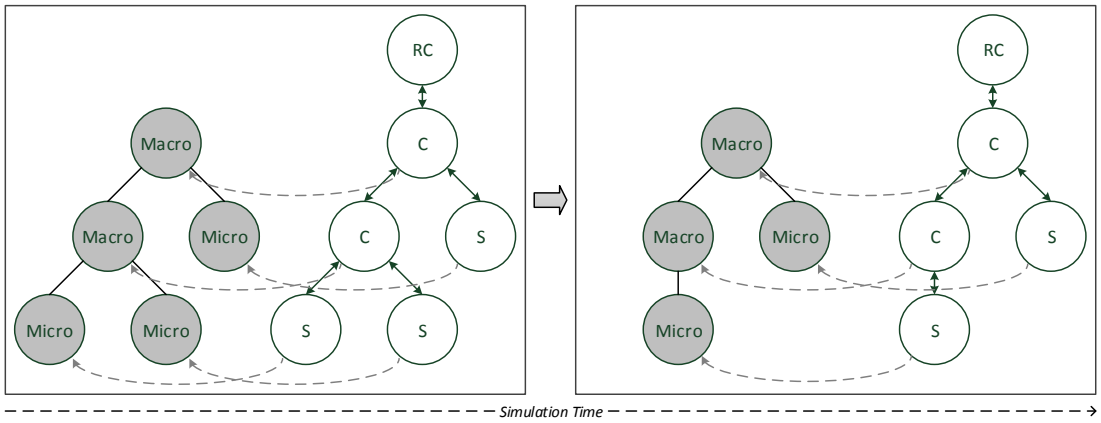

Figure 9.7: Mapping between two incarnations of a hierarchical ML-DEVS model and the corresponding processor tree responsible for executing the model. "RC," "C," and "S" denote a Root Coordinator, Coordinator, and Simulator, respectively.

Table 9.1: The messages the communication protocol of the abstract simulator of ML-DEVS consists of.

Message	Purpose
`i`-message or initialization message	initializes the receiver
`*`-message or star message	requests outputs from the receiver
`y`-message or output message	contains the outputs of the sender
`x`-message or input message	triggers a state transition in the receiver
`done`-message	signals that the sender finished the current simulation step

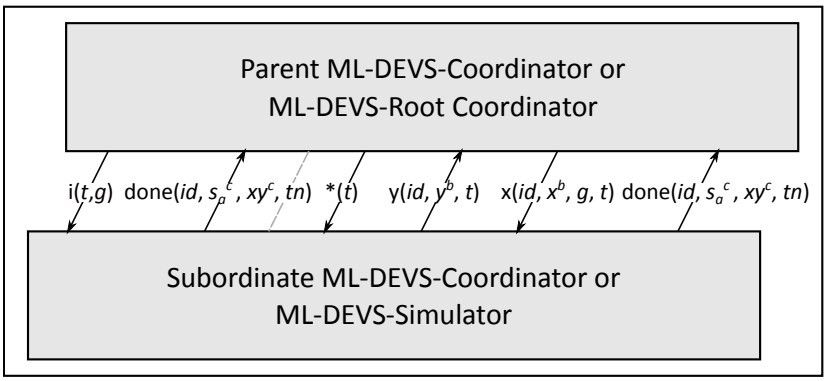

Figure 9.8: Communication protocol of the abstract simulator of ML-DEVS.

`done`-messages by additional information that is necessary for achieving up- and downward information. Thereby, the information that is propagated up- and downward—i. e., model identifiers (denoted by id), global information (denoted by g), accessible states (denoted by s_a^c), and available ports (denoted by xy^c)—is separated from regular input bags (denoted by x^b) and output bags (denoted by y^b) of models. A similar extension of messages can, e. g., be found in Uhrmacher [2001] and Uhrmacher et al. [2007]. In addition, the communication protocol of ML-DEVS incorporates an explicit initialization message, the `i`-message (cf. Zeigler et al. [2000, p. 176 ff.]).

The `i`-message is used to initialize processors with the current simulation time t and the currently valid global information g. Note that due to the modularity of ML-DEVS, global

information is defined between a macro level and its micro level[31] and is thus not accessible from arbitrary levels of a complex composition hierarchy without further ado. If information shall be made available throughout the entire composition hierarchy (in the proper meaning of the word "global"), the modeler has to take care about an appropriate propagation of the information from one level to another. Also note that due to the variable composition of ML-DEVS models and in contrast to P-DEVS, `i`-messages are not only sent at the beginning of a simulation run, but whenever new model components are added to a composition, during simulation. For this reason, the `i`-message plays a special role in Figure 9.8. After receiving an `i`-message, the corresponding processor sends a `done`-message back to its parent processor. The *done*-message (i) contains the identifier id, available ports xy^c, accessible state s_a^c, and the time of the next internal event tn of the sender and (ii) indicates that the sender finished the initialization. The `*`-message is sent from a Coordinator or Root Coordinator to its *imminent* child processors, i. e., those processors the associated models of which are about to perform an internal state transition at the current simulation time t. In return, each imminent processor sends a `y`-message to its parent containing the outputs of the associated model. The `x`-message is used to trigger all processors whose models are imminent and/or influenced (receive inputs). In addition to an input bag (which can be empty), an `x`-message contains global information, which may have changed in the previous simulation cycle. At the end of a simulation cycle, each active processor that received an `x`-message sends a `done`-message to its parent processor in return, just like when receiving an `i`-message. These `done`-messages contain updates of the accessible states and available ports of the associated models.

Next, the different parts of the abstract simulator of ML-DEVS, i. e., the different kinds of processors, are described in more detail. The description revises and refines the one given in Steiniger et al. [2012] and is based on Steiniger and Uhrmacher [2016]. The presented abstract simulator and a Java-based representation of the ML-DEVS model specification are implemented as plug-ins for the modeling and simulation framework JAMES II[32] [Himmelspach 2007; Himmelspach & Uhrmacher 2007].

9.3.1 Simulator

Algorithm 9.1 shows the Simulator of ML-DEVS, which is rather simple in comparison with the Coordinator of ML-DEVS and which is similar to the one of P-DEVS presented by Zeigler et al. [2000, p. 285]. A Simulator is responsible for executing a given specification of a Micro-DEVS model, denoted by m, associated with the Simulator. Where m is as in Definition 9.2.1, i. e.,

$$m = \langle id, XY, S_p, S_a, s_{init}, p, \delta, \lambda, ta \rangle.$$

During execution, the model m is characterized by its private state, accessible state, and outputs and by the time of the last event (denoted by tl) and the time of the next internal event (denoted by tn), which change according to the semantics described in the following. All these information form the runtime model of m. In addition, the Simulator holds certain variables that are necessary to determine subsequent states and outputs.

At first, a Simulator receives an `i`-message (initialization message) with the current simulation time t and information that is propagated downward (*global information*), denoted by g, from its parent (Algorithm 9.1, line 2). Afterward, the private state s_p and accessible state s_a of the model m are set to the initial private state and initial accessible state, respectively[33], which are part of the initial state s_{init} of the model m. As a result of *downward information*, the accessible state s_a is updated based upon the global information g sent from the *parent*

[31] for which the information is global

[32] `http://jamesii.org`; last accessed February 2018

[33] $\pi_i()$ refers to the i-th projection as defined in Appendix A.1.7

Algorithm 9.1: Simulator of ML-DEVS for MICRO-DEVS models.

```
variables:
    m                    // atomic model associated with this Simulator
    s_p, s_a             // the atomic model's current private and accessible state
    P                    // currently available ports with P ⊆ 𝒫
    s_a^m                // set of accessible state variable assignments
    xy^m                 // set of available ports
    tl, tn               // time of last event and time of next internal event

1  // initialization
2  when receive i-message(t, g) at time t with global information g
3      s_p ← π_1(m.s_init)
4      s_a ← π_2(m.s_init)
5      update s_a according to g
6      P ← m.p(s_p)
7      xy^m ← {(pn, v, X) | pn ∈ P ∧ X = range_pn(m.XY) ∧ v = ε}
8      s_a^m ← {(vn, v, X) | vn ∈ variables(m.S_a) ∧ X = range_vn(m.S_a) ∧ v = s_a(vn)}
9      tl ← t
10     tn ← tl + m.ta(s_p)
11     send done-message(m.id, xy^m, s_a^m, tn) to parent
12
13 // generate output bag
14 when receive *-message(t) at time t
15     y^b ← m.λ(s_p)
16     update y^b according to P
17     send y-message(m.id, y^b, t) to parent
18
19 // perform state transition
20 when receive x-message(id, x^b, g, t) at time t with input bag x^b and global
       information g
21     update x^b according to P
22     update s_a according to g
23     (s_p, s_a) ← m.δ((s_p, s_a, t − tl), x^b)
24     P ← m.p(s_p)
25     xy^m ← {(pn, v, X) | pn ∈ P ∧ X = range_pn(m.XY) ∧ v = ε}
26     s_a^m ← {(vn, v, X) | vn ∈ variables(m.S_a) ∧ X = range_vn(m.S_a) ∧ v = s_a(vn)}
27     tl ← t
28     tn ← tl + m.ta(s_p)
29     send done-message(m.id, xy^m, s_a^m, tn) to parent
```

(line 5). For this, the global information, which is a set of name-value pairs (see Section 9.2) with unique names, is examined. If there is an accessible state variable whose name also appears as a name in g, the associated value in g is used to update the value of the matching accessible state variable. Thus we obtain the updated accessible state $s_a' \in m.S_a$ based on the current accessible state s_a and the global information g as follows:

$$\forall vn \in variables(S_a) : \; s_a'(vn) = \begin{cases} v & \text{if } \exists\, (vn, v) \in g \wedge v \in range_{vn}(m.S_a) \\ s_a(vn) & \text{otherwise.} \end{cases} \tag{9.8}$$

Note that the value of an accessible state variable is only updated if the corresponding value from the global information is within the actual value range of the accessible state variable, which is a relaxation of the second constraint in Definition 9.2.2. In line 6, the port names that are available in the current private state s_p are determined by the port selection function of the model m. Then the Simulator initializes the two sets xy^m and s_a^m based on the currently available ports P and the current accessible state s_a, respectively (lines 7 and 8). When recalling the definitions of the sets XY^c and S_a^c from Definition 9.2.2 it becomes apparent that

$xy^m \in XY^c$ and $s_a^m \in S_a^c$ with $m = c$, if m is in some C of a MACRO-DEVS model for which XY^c and S_a^c shall be defined. In lines 9 and 10 the time of the last event tl and the time of the next internal event tn (also often referred to as *tonie*) are initialized. At the end of the initialization, the Simulator sends a done-message to its parent (line 11), which contains the identifier of the model m, the time of the next internal event tl, and the two sets s_a^c and xy^c. The latter two sets and the model identifier represent the current instance of the interface of the model m, which is made accessible to the parent.

The processing of a *-message (lines 14 to 17) is similar to the processing of a *-message in traditional P-DEVS (cf. Zeigler et al. [2000, p. 285]). However, as ML-DEVS supports variable ports, not all ports may be available in the current state. Therefore, the output bag y^b returned by the output function of the model m (line 15) is updated in a way that only ports that are available, i. e., whose names are element of P, appear in the elements of the output bag that is actually sent to the parent. We obtain the updated output bag $y^{b'}$ from a given output bag $y^b \in (m.XY)^b$ and the currently available port names P as follows:

$$y^{b'} = \biguplus_{y \in y^b} \left\{ y' \colon P \cap dom(y) \to ran(y);\ p \mapsto y(p) \right\}. \tag{9.9}$$

As $P \cap dom(y) \subseteq variables(m.XY)$ and the range of y' is the range of y with $y \in m.XY$, y' is an element of $m.XY$ as well[34]. After the update, the set y^b is overwritten by $y^{b'}$ (line 16). This update ensures that the second constraint from Definition 9.2.1 is enforced, even if the modeler has violated the constraint, intentionally or unintentionally (see Section 9.2.3). In addition, ML-DEVS makes no distinction between input ports and output ports. When creating an output bag all available ports that contain values are considered as output ports. After an output bag has been created and updated, the bag is sent within a y-message to the Simulator's parent (line 17).

Finally, the x-message, which contains an input bag x^b and global information g, triggers an actual state transitions (internal, external, or confluent). Sending the global information g via the x-message and not whenever the corresponding information changes at the macro level has the advantage that global information is only sent on demand, i. e., when the Simulator of model m becomes active[35]. After receiving an x-message, the input bag x^b is updated according to the currently available ports (line 21), similar to the update of an output bag as described above. We obtain the updated input bag $x^{b'}$ from a given input bag $x^b \in (m.XY)^b$ and the currently available port names P as follows:

$$x^{b'} = \biguplus_{x \in x^b} \left\{ x' \colon P \cap dom(x) \to ran(x);\ p \mapsto x(p) \right\}. \tag{9.10}$$

Each function x' is—by definition—an element of $m.XY$. Next, the accessible state s_a is updated according to the global information g sent within the x-message (line 22), just like during the initialization (see Equation 9.8). In line 23, the Simulator determines the new private and accessible state by invoking the state transition function of the model m. The new states does not necessarily have to be different from the previous states. Afterward, the port names available in the new public state are determined by the port selection function. As the accessible state and the available ports may have changed as a result of the state transition, the sets xy^m and s_a^m are updated (lines 25 and 26). Before the done-message is sent to the parent (line 29), the time of the last event tl and the time of the next internal event tn are updated based upon the current simulation time and the time advance function of the model m (lines 27 and 28), such that the simulation can advance.

[34] Within its domain, a partial function can be viewed as a total function (see Appendix A.1.6).

[35] Note that in P-DEVS the x-message is used to trigger state transitions in general and not only external state transitions as in the case of classic DEVS.

Algorithm 9.2: Coordinator of ML-DEVS for Macro-DEVS models.

```
variables:
    parent              // parent processor
    n                   // associated model (coupled model)
    s_p, s_a            // current private and accessible state of n
    C, MC               // currently available components and multi-couplings
    P                   // currently available port names with P ⊆ 𝒫
    y^b                 // output bag to parent
    s_a^n               // set of accessible state variable assignments
    xy^n                // set of available ports
    x_act               // downward activation message (with x_act ∈ n.XY^n)
    g_down              // information propagated downward to children
    msg                 // message container for outputs from components
    inp                 // input message bag for children
    tl, tn              // time of last event and next internal event
    {I_c}               // is the set of influencers for c, with c ∈ {n} ∪ n.C
    {P_i,c}             // is the set of port-to-port mappings, with i ∈ I_c and c ∈ C
    i^n                 // component interface instances with i^n ∈ I^n
    IMM, INF, ACT       // imminent, influenced, and activated children
    events              // queue of elements (id, tn_id) sorted by tn_id (ascending order)

1  when receive i-message(t, g) at time t with global information g
2      initialization(t, g)
3
4  when receive *-message(t) at time t
5      processStarMessage(t)
6
7  when receive x-message(id, x^b, g, t) at time t with input bag x^b and global
       information g
8      processXMessage(t, x^b, g)
```

9.3.2 Coordinator

Algorithm 9.2 shows the Coordinator of ML-DEVS, which revises and extends the one presented in Steiniger et al. [2012] and Steiniger and Uhrmacher [2016]. Due to its complexity and for the sake of clarity, we break down the Coordinator into special functions, which are called by the Coordinator and operate on its global variables, listed at the beginning of Algorithm 9.2.

In general, a Coordinator in ML-DEVS is responsible for executing the specification of an associated Macro-DEVS model, denoted by n, with n as in Definition 9.2.2, i. e.,

$$n = \langle id, XY, S_p, S_a, C, MC, s_{init}, \delta, \lambda, p, ta, sc, \lambda_{down}, v_{down}, act_{up} \rangle.$$

The Coordinator combines the functionality of other P-DEVS coordinators (cf. Zeigler et al. [2000, pp. 284–7]) and the Simulator of ML-DEVS, as Macro-DEVS models can comprise other ML-DEVS models and both Micro-DEVS and Macro-DEVS models have certain sets and functions in common. However, in contrast to coordinators of other DEVS variants, which are comparatively "simple," the Coordinator of ML-DEVS has, in addition to receiving and forwarding events, (i) to frequently translate the intensional couplings into concrete coupling schemes before events can be forwarded and (ii) to decide whether or not n or its components have to be triggered as a result of up- or downward causation.

After receiving an i-message with the current simulation time t (Algorithm 9.2, line 1), the Coordinator initialized its internal state[36] (Algorithm 9.3, lines 2 and 3) and internal structures that are required later (line 1). Of particular interest is the set of the available

[36] Which is the state of the associated Macro-DEVS model, denoted by n.

Algorithm 9.3: Initialization of the Coordinator.

```
method name: initialization
inputs:
   t                    // current simulation time
   g                    // information propagated downward from parent

1  events, i^n ← ∅
2  s_p ← π_1(n.s_init)
3  s_a ← π_2(n.s_init)
4  C ← π_3(n.s_init)
5  MC ← π_4(n.s_init)
6  P ← n.p(s_p)
7  update s_a according to g
8
9  // initialize components
10 g_down ← {(vn, v) | vn ∈ n.v_down(s_p) ∧ v = s_p(vn)}
11 send i-message(t, g_down) to all c ∈ C
12    wait for done-message(id^c, xy^c, s_c^a, tn_c) from each c
13       enqueue (id^c, tn_c) in events
14       add (id^c, (xy^c, s_c^a)) to i^n
15
16 xy^n ← {(pn, v, X) | pn ∈ P ∧ X = range_pn(n.XY) ∧ v = ε}
17 s_a^n ← {(vn, v, X) | vn ∈ variables(n.S_a) ∧ X = range_vn(n.S_a) ∧ v = s_a(vn)}
18 tl ← t
19 tn ← min(n.ta(s_p), min_tn(events))
20 send done-message(m.id, xy^n, s_a^n, tn) to parent
```

components' interface instances $i^n \in I^n$ with I^n as defined in Equation 9.5, which is used for translating the multi-couplings into concrete port-to-port couplings and from which s^n and xy^n are derived. The initially available components and multi-couplings are given as part of the initial state of the coupled model n (lines 4 and 5). After the state of the Coordinator is initialized, the available port names, denoted by P, are determined (line 6). In line 7, the accessible state s_a is updated according to Equation 9.8 based on the global information g sent within the i-message from the parent of the Coordinator. Before all initially available components of the coupled model n are initialized (i. e., their associated processors) the global information that shall be propagated downward, denoted by g_{down}, is determined based on the v_{down}-function of n and the initial public state (line 10)[37]. In general, the global information g_{down} is defined as a set of name-value pairs that refer to names and values of private state variables of n. For a given private state s_p, we define g_{down} as follows:

$$g_{down} = \{(vn, v) \mid vn \in n.v_{down}(s_p) \wedge v = s_p(vn)\} \tag{9.11}$$

Next, the Coordinator sends an i-message to each available component $c \in C$ and waits for a done-message from each component in return (lines 11 and 12). Each done-message contains, among other things, the identifier (denoted by id^c) and the time of the next internal event (denoted by tn_c) of the sender. The Coordinator uses both information to initialize its internal event queue[38], which schedules the internal events of the available components in C (line 13). In addition, each incoming done-message of a component c contains the available ports (denoted by xy^c) and accessible state (denoted by s_c^a) of c. Together with the identifier of c, the sets xy^c and s_c^a are used to initialize the set of interface instances i^n by adding the

[37] The function v_{down} returns the names of the private state variables whose values are made accessible.

[38] At any time during simulation, the event queue contains exactly one event for each component available at this time. This event indicates when—from the current simulation time on—the corresponding component performs an internal state transition, if the component does not receive any inputs in the meantime. Due to the variable composition of ML-DEVS, the event queue can grow and shrink during simulation.

Algorithm 9.4: Processing of an incoming *-message by the Coordinator.

```
method name: processStarMessage
inputs:
  t                    // current simulation time
local variables:
  y^b                  // output bag to parent of n

1  // initialize required variables
2  msg, y^b ← ∅
3  s^n ← {(id_c, s_{a,c}) | ∃(id_c, (xy_c, s_{a,c})) ∈ i^n}
4
5  // translate multi-couplings
6  ({I_c}, {P_{i,c}}) ← translateCouplings(i^n ∪ {(n.id, (xy^c, s_a^c))})
7
8  // trigger all imminents and forward their outputs
9  IMM ← {c ∈ C | ∃(id, tn_id) ∈ events: id = c.id ∧ tn_id = t}
10 send *-message(t) to all c ∈ IMM
11   wait for y-messages(id_c, y_c^b, t_c) from each c
12     // remove corresponding internal event from event queue
13     dequeue (id_c, t_c) from events
14     // buffer output bag of current child c
15     add (id_c, y_c^b) to msg
16
17 // check if n is imminent as well
18 if tn = t then
19   y^b ← n.λ(s_p)
20   x_act ← n.λ_down(s_p, i^n)
21   update y^b according to p
22   update x_act according to i^n
23
24 // add outputs of children directed to n to output bag
25 for each (id_i, y_i^b) ∈ msg with ∃i ∈ C: i.id = id_i ∧ i ∈ I_n do
26   for each y ∈ y_i^b do
27     for each (pn, v) ∈ y do
28       add (P_{i,n}(pn), v) to y^b
29
30 // send merged outputs to parent
31 send y-message(n.id, y^b, t) to parent
```

tuple $(id^c, (xy^c, s_c^a))$ to the set, which is empty at the beginning. As a done-message is only sent once per component, the constraint defined in Equation 9.6 cannot be violated. Similar to the Simulator, the Coordinator determines the information that shall be propagated upward (lines 16 and 17) and updates tl and tn (lines 18 and 19) at the end of the initialization phase. The initialization of the Coordinator ends with sending a done-message to the parent processor (line 20).

In DEVS variants such as ML-DEVS, a regular simulation cycle is initiated by a *-message. When the Coordinator receives such a message from its parent (either another Coordinator or a Root Coordinator), first, the available, intensional multi-couplings MC have to be evaluated and translated into a concrete coupling scheme consisting of consistent port-to-port couplings and based on the interface instances of the currently available components and the coupled model n (Algorithm 9.4, line 6). The Coordinator can then use the derived coupling scheme to forward in- and output events.

Algorithm 9.5 shows a naive reference implementation of the translation of intensional multi-couplings, which assures that the result of the translation are consistent couplings (*correctness by construction*). To sum-up Definition 9.2.4, a port-to-port coupling is consistent

Algorithm 9.5: Optimistic translation of all active multi-couplings into a concrete coupling scheme.

```
method name: translateCouplings
inputs:
  i^{n'}                  // interface instances of all available components and n
                          // with i^{n'} ⊂ I^n
outputs:
  {I_c}                   // set of sets of influencers of c with c ∈ C
  {P_{i,c}}               // set if port-to-port mappings with i ∈ I_c, c ∈ C,
                          // and where for each i and c P_{i,c}: P_i → P_c
variables:
  Cplg_ext                // set of potential port-to-port couplings
  id, id'                 // identifiers of the source and target component
  c, c'                   // source and target component
  pn_s, pn_t              // names of the source and target port
  X, X'                   // value ranges of the source and target port

1  // initialization
2  {I_c}, {P_{i,c}} ← ∅
3  Cplg_ext ← ⋃_{mc ∈ MC} {mc(i^{n'})}
4
5  // iterate all possible combinations of existing component pairs
6  for each id ∈ {id_s | id_s ∈ dom(i^{n'}) ∧ ((id_s, pn_s), (id_t, pn_t)) ∈ Cplg_ext} do
7    for each id' ∈ {id_t | id_t ∈ dom(i^{n'}) ∧ ((id_s, pn_s), (id_t, pn_t)) ∈ Cplg_ext} do
8      // check if there is no direct feedback loop
9      if id ≠ id' then
10       // get the corresponding components for additional consistency checks ||
11       c ← getComponent(id, C)
12       c' ← getComponent(id', C)
13       // iterate all potential port-to-port couplings to consider
14       for each ((id, pn_s), (id', pn_t)) ∈ ⋃_{mc∈MC} {mc(i^{n'})} do
15         // check availablility of respective ports
16         if pn_s ∈ dom(dom(xy^c)) with (id, (xy^c, s_a^c)) ∈ i^{n'} then
17           if pn_t ∈ dom(dom(xy^c)) with (id', (xy^c, s_a^c)) ∈ i^{n'} then
18             // check value ranges
19             X ← getRange(pn_s, xy^c) with (id, (xy^c, s_a^c)) ∈ i^{n'}
20             X' ← getRange(pn_t, xy^{c'}) with (id', (xy^{c'}, s_a^{c'})) ∈ i^{n'}
21             if X ⊆ X' then
22               // update corresponding influencer set
23               add c to I_{c'}
24               // initialize corresponding mapping if necessary
25               if P_{c,c'} = ∅ then ∀pn ∈ variables(c.XY): P_{c,c'}(pn) = ∅
26               // update corresponding mapping
27               add pn_s ↦ pn_t to P_{c,c'}
28
29 // return updated sets
30 return {I_c} and {P_{i,c}}
```

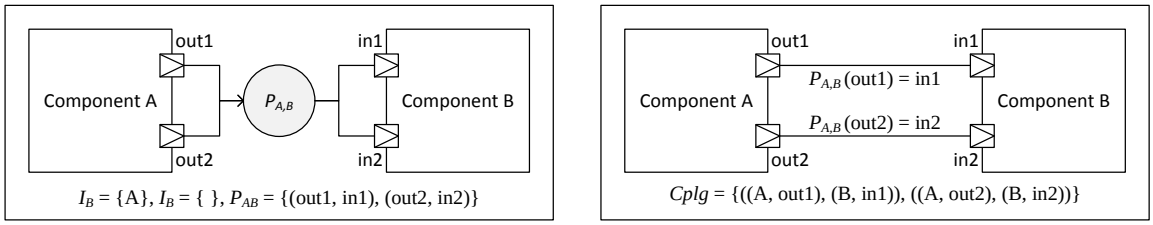

Figure 9.9: The relationship between a port-to-port map as used by the Coordinator (left-hand side) and port-to-port couplings as returned by the intensional multi-couplings of ML-DEVS (right-hand side).

if (i) the referenced components and ports are available in a given state, (ii) the value range of the source port is a subset of the value range of the target port, and (iii) the source component is not the target component (as direct feedback loops are not allowed like in other DEVS variants). Internally, the resulting concrete coupling scheme is represented by the two sets:

- $\{I_c \,|\, c \in C \cup \{n\}\}$ a set of sets of influencers of the coupled model n and its components, where for each $c \in C:\ I_c \subseteq C \cup \{n\}$ and for $c = n:\ I_c \subseteq C$;
- $\{P_{i,c} \,|\, c \in C \cup \{n\} \wedge i \in I_c \wedge c \notin I_c\}$ a set of port-to-port maps with for each i and p

$$P_{i,c}\colon \mathcal{P}_i \to \mathcal{P}_c \cup \{\varnothing\},$$

where $\mathcal{P}_i$ and $\mathcal{P}_c$ denote the sets of port names of the model components i and c, respectively, i. e., $\mathcal{P}_i = \mathit{variables}(i.XY)$ and $\mathcal{P}_c = \mathit{variables}(c.XY)$.

Figure 9.9 illustrates the relationship between such a coupling scheme that is used internally and the concrete port-to-port couplings as defined in Section 9.2.2.

At the beginning of the translation (Algorithm 9.5), each available multi-coupling in MC is applied to the current set of interface instances including the interface instance of the coupled model n. All port-to-port couplings, i. e., coupling candidates, returned by the different multi-couplings are collected in the set Cplg_{ext} (Algorithm 9.5, line 3). Whether these port-to-port couplings are part of the eventual coupling scheme that is returned by the translation algorithm depends on their consistency. Therefore, for each port-to-port coupling candidate, which consists of two model identifiers and two port names, the following questions are answered (lines 6 to 25):

1. Do the model identifiers reference currently available components including the coupled model (line 6 and 7)?
2. Are the referenced components different (line 10)?
3. Do the corresponding components have ports whose names match those of the coupling candidate and that are currently available (lines 16 and 17)?
4. Is the value range of the source port a subset of the value range of the target port (line 21)?

If all these questions are answered in the affirmative, the source component is added to the set of influencers of the target component (line 23) and a mapping from the source to the target port is added to the respective port-to-port map (line 27). For this, the auxiliary function $\mathit{getComponent}\colon \mathcal{N} \times 2^C \to C \cup \{\text{“undef”}\}$ is used (lines 11 and 12), which returns for a given model identifier the corresponding model specification or ‘undef’ if there is no model whose identifier matches the given one, i. e.,

$$\mathit{getComponent}(id, C) = \begin{cases} c & \text{if } \exists c \in C:\ id = c.id \\ \text{“undef”} & \text{otherwise.} \end{cases}$$

The proposed translation algorithm is optimistic as it assumes that the set $i^{n\prime}$ reflects the actual state of the composition, so that the function *getComponent* does not return 'undef' for c or c'. After all consistent port-to-port couplings have been identified, the derived coupling scheme is returned (line 30) and can be used by the Coordinator to forward events, in the current simulation cycle.

Next, the Coordinator sends a *-message to each of its imminent children (Algorithm 9.4, lines 9 and 10) and waits for y-messages in return (line 11). These messages contain the output bags of the imminents. The corresponding internal events will be removed (dequeued) from the event queue of the Coordinator and the output bag of each imminent component is buffered in the message container *msg* for later usage (line 15). As a coupled model n can have outputs of its own, the Coordinator checks whether n is imminent itself (line 18). If so, the output bag of n, denoted by y^b, is determined by the coupled model's output function λ and the current private state s_p (line 19). In addition, the downward activation events, denoted by x_{act} are determined by the downward activation function λ_{down} (line 20). Afterward, the output bag y^b is updated according to the currently available port names p (see Equation 9.9). Furthermore, the activation message x_{act} is updated according to the set of the interface instances of the currently available components. We obtain the updated downward activation message ${x_{act}}'$ from the set i^n as follows:

$$ {x_{act}}' = \{(id, xy) \in x_{act} \,|\, (id, (xy^c, s_a^c)) \in i^n \wedge dom(dom(xy)) \subseteq dom(dom(xy^c))\}\,. \qquad (9.12) $$

Next we overwrite the value of x_{act} with the value of ${x_{act}}'$. This update ensures that the first constraint in Definition 9.2.2 is not violated. Concluding the processing of the *-message from the parent of the Coordinator, all values of ports of components that are coupled to ports of the coupled model n are added to the output bag y^b (lines 25 to 28), which is finally sent to the parent of the Coordinator within a y-message (line 31).

At the end of each simulation cycle, x-messages are sent downward to all active processors, i. e., processors that are imminent, influenced, or activated. When a Coordinator receives such a message with an input bag x^b and the global information g from its parent (Algorithm 9.2, line 7), the input bag x^b is updated according to the currently available port names (see Equation 9.13) and the accessible state s_a is updated according to the global information sent via the x-message (see Equation 9.8) (Algorithm 9.6, lines 37 and 3). Before the Coordinator sends x-messages to its active children (line 12), the global information g_{down} that shall be propagated downward is determined based on the function v_{down} of the coupled model n and the current private state s_p. Which of the available child processors are active is determined based on (i) the current simulation time t and tn of the child processors[39], (ii) the input bag x^b of the coupled model n and the external input couplings in the derived coupling scheme (line 7), and (iii) the downward activation message x_{act} (line 8). The Coordinator has then to bundle all inputs that are directed to each of the active child processors (line 11). The input bag x_c^b of a child processor, i. e., of an active component c, includes: (i) inputs of the model n which are delegated to the model component c via external input couplings, (ii) inputs of other, imminent components of the model n that are forwarded to component c via internal couplings, and (iii) activation events of the model n as a result of the downward output function λ_{down}. The auxiliary function $inp(\ldots)$ undertakes the task of collecting all inputs and creating an input bag for each active component c as follows:

$$ \begin{aligned} inp(c, n, x^b, msg, x_{act}, I_c, \{P_{i,c} \,|\, i \in I_c\}) = & inp_{EIC}(n, x^b, I_c, P_{i,c}) \\ & \uplus\ inp_{IC}(n, msg, I_c, \{P_{i,c} \,|\, i \in I_c\}) \\ & \uplus\ inp_{ACT}(c, x_{act}) \end{aligned} \qquad (9.13) $$

[39] Notice that although the internal event queue is already cleaned up, the set *IMM* still contains all imminent children.

Algorithm 9.6: Processing of an x-message by the Coordinator.

```
method name: processXMessage
inputs:
  g                    // information propagated downward from parent
  t                    // current simulation time

1  // update accessible state and input bag
2  update s_a according to g
3  update x^b according to p
4
5  // execute all imminent, influenced, and activated children
6  g_down ← {(vn, v) | vn ∈ n.v_down(s_p) ∧ v = s_p(vn)}
7  INF ← {c ∈ C | ∃i ∈ C ∃(id_i, y_i^b) ∈ msg ∃y ∈ y_i^b ∃(pn, v) ∈ y: i.id = id_i ∧ i ∈ I_c ∧ P_i,c(pn) ≠ ∅}
8  ACT ← {c ∈ C | ∃(id, xy) ∈ x_act: c.id = id ∧ xy ≠ ∅}
9  for each c ∈ IMM ∪ INF ∪ ACT do
10     // merge all inputs for current component
11     x_c^b ← inp(c, n, x^b, msg, x_act, I_c, {P_i,c | i ∈ I_c})
12     send x-message(n.id, x_c^b, g_down, t) to c
13        wait for done-message(id_c, xy_c, s_a,c, tn_c) from each c
14           enqueue (id_c, tn_c) in events
15           update i^n with (id_c, (xy_c, s_a,c))
16
17 // check if macro model has to be executed
18 // update network state
19 s^n ← {(id_c, s_a,c) | ∃(id_c, xy_c, s_a,c) ∈ i^n}
20 if n.act_up(s_p, s^n) or x^b ≠ ∅ or tn = t then
21     // change model state and update global information and port names
22     (s_p, s_a) ← n.δ((s_p, s_a, t − tl), s^n, x^b, a^b)
23     g_down ← {(vn, v) | vn ∈ n.v_down(s_p) ∧ v = s_p(vn)}
24     p ← n.p(s_p)
25     // change model structure
26     C* ← C
27     (C, MC) ← n.sc(s_p, s^n)
28     // initialize new components
29     send i-message(t, g_down) to all c ∈ C\C*
30        wait for done-message(id_c, xy_c, s_a,c, tn_c) from each c
31           enqueue (id_c, tn_c) in events
32           add (id_c, (xy_c, s_a,c)) to i^n
33     // update event queue and interface instance set
34     remove all c ∈ C*\C from i^n and events
35
36 xy^c ← {(n, v, X) | n ∈ p ∧ X = range_n(m.XY) ∧ v = ε}
37 s_a^c ← {(n, v, X) | n ∈ variables(m.S_a) ∧ X = range_n(m.S_a) ∧ v = s_a(n)}
38 tl ← t
39 tn ← t + min(n.ta(s), min_tn(events))
40 send done-message(m.id, xy^c, s_a^c, tn) to parent
```

where

$$inp_{EIC}(n, x^b, I_c, P_{n,c}) = \begin{cases} \biguplus\limits_{x \in x^b} \left\{ \left(P_{n,c}(pn), x(pn) \right) \mid pn \in dom(x) \wedge P_{n,c}(pn) \neq \varnothing \right\} & \text{if } n \in I_c \\ \varnothing & \text{else} \end{cases}$$

and where

$$inp_{IC}(n, msg, I_c, \{P_{i,c} \mid i \in I_c\}) = \begin{cases} \biguplus\limits_{i \in I_c} \{inp_{IC\,i}(out_i(i, msg), P_{i,c})\} & \text{if } I_c \backslash \{n\} \neq \varnothing \\ \varnothing & \text{else} \end{cases}$$

with

$$out_i(i,\ msg) = \begin{cases} y_c^b & \text{if } \exists!\left(id_c,\ y_c^b\right) \in msg\colon\ id_c = i.id \\ \varnothing & \text{else} \end{cases}$$

and

$$inp_{IC\,i}(y_i^b, P_{i,c}) = \biguplus_{y \in y_i^b} \left\{ \left(P_{i,c}(pn), y(pn) \right) \mid pn \in dom(y) \wedge P_{n,c}(pn) \neq \varnothing \right\}$$

and where

$$inp_{ACT}(c, x_{act}) = \begin{cases} xy & \text{if } \exists!(id, xy) \in x_{act}\colon\ id = c.id \\ \varnothing & \text{else.} \end{cases}$$

On closer inspection it becomes apparent, that

$$inp(c, n, x^b, msg, x_{act}, I_c, \{P_{i,c} \mid i \in I_c\}) \in c.XY^b.$$

In the case that an active component c is only imminent but not influenced or activated, the input bag of c will be empty, i. e., $x_c^b = \varnothing$. After the Coordinator has sent an x-message to each of its active components (line 12), it waits for done-messages of the active components in return. Each done-message is used to update the internal event queue and the set of current interface instances i^n (lines 14 and 15). We obtain the updated set of interface instances $i^{n\prime}$ from the current set of interface instances i^n and the available ports xy_c and the accessible state $s_{a,c}$ of a component c with the identifier id_c as follows:

$$i^{n\prime} = \left\{ (id_c{}', (xy_c{}', s_{a,c}{}')) \in i^n \mid id_c{}' \neq id_c \right\} \cup \left\{ (id_c, (xy_c, s_{a,c})) \right\}. \tag{9.14}$$

Afterward the network state s^n is updated according to the changed set of interface instances (line 19). Hence, the Coordinator operates on the updated interface instances and network state in the remainder of the current simulation cycle. As the coupled model can change it's state as well, the Coordinator checks whether or not the state transition function δ has to be invoked, i. e., whether the coupled model is activated upwards, the input bag x^b is not empty, or the lifespan defined for the current private state is exceeded (line 20). If so, the private and accessible state are updated by calling the state transition function of n (line 22). As a result of the state transition, the global information that shall be propagated downward g_{act} and the available port names P have to be updated as well (lines 23 and 24). Then, the new model structure is changed according to the structure change function sc. Thereby, the Coordinator initializes all newly available components by sending them i-messages (line 29). In return, the internal event queue is updated by adding internal events for the new components to it and the interface instances of the new components are added to the set of interface instances i^n

Algorithm 9.7: The Root Coordinator of ML-DEVS.

```
variables:
   child                // subordinate processor
   c                    // model associated with subordinate processor
   t                    // simulation time
parameters:
   t_0                  // start time of the simulation

1  // initialization
2  t ← t_0
3  send i-message(t, ∅) to child
4     wait for done-message(id_c, xy_c, s_a,c, tn_c) from child
5        t ← tn_c
6
7  // simulation loop
8  loop
9     // start current simulation cycle
10    send *-message(t) to child
11       wait for y-message(id_c, xy_c, t_c) from child
12
13    send x-message(id, ∅, ∅, t) to child
14       wait for done-message(id_c, xy_c, s_a,c, tn_c) from child
15          t ← tn_c
16 until *end* *of* simulation
```

(lines 30 to 32). The adaptation of the model structure is completed by removing the interface instances of all recently deleted components from i^n (line 34). At the end of the processing of a x-message updates the information that is propagated upwards (lines 36 and 37) and the internal times. Finally, the Coordinator sends a done-message to its parents, which contains its available ports and public state as well as the time of the next internal event of the coupled model n.

9.3.3 Root-Coordinator

The last type of processor the abstract simulator of ML-DEVS consists of is the Root Coordinator. This processor is special in the way that, in contrast to a Simulator or Coordinator, their is no model associated with the Root Coordinator. Any processor tree that is derived from a given ML-DEVS model has only one Root Coordinator, which forms the root of the respective processor tree that is responsible for executing the model.

Algorithm 9.7 shows the Root Coordinator of ML-DEVS, which is rather simple in comparison to the other two types of processors. The Root Coordinator is responsible for initiating and controlling the *simulation loop* consisting of *cycles* (simulation steps[40]) while advancing the global simulation time, by jumping to the time of the next event.

Before an actual simulation (run) is started, i. e., during initialization (Algorithm 9.7, lines 2 to 5), the Root Coordinator sets the current simulation time t to the given start time t_0 (line 2). This start time t_0 is a parameter of the Root Coordinator and set externally. How this is exactly done is not of interest at this point, instead it is subject to a concrete implementation of the abstract simulator. Next, the Root Coordinator sends an i-message to its child processor with the current simulation time t and without global information (i. e., an empty set), as the child processor is already associated with the topmost model component

[40] In the literature (e. g., Himmelspach and Uhrmacher [2006]), also the term "pulse" is used to denote a single simulation cycle, which is initiated by sending and propagating *-messages downward the processor tree.

in a hierarchical ML-DEVS model (line 3). In return, the Root Coordinator waits for a done-message from its child processor (line 4). Afterward, the simulation time is set to the time of the next internal event sent within the done-message (line 5). Notice that whereas a Root Coordinator sends an i-message only once at the very beginning of a simulation run, Coordinators at lower levels of the processor tree can also send i-messages during a simulation run, as Coordinators can change their composition due to the variable composition in ML-DEVS.

As long as the current simulation run is not ended (lines 8 and 16), a simulation cycle (lines 10 to 15) is initiated by sending a *-message with the current simulation time t to the child processor of the Root Coordinator (line 10). The information propagated upward by the child processor within the y-message (line 11), which is sent in response to the *-message, is ignored, as there is no processor, i. e., model, above the Root Coordinator. Before a simulation cycle ends, the Root Coordinator sends an empty x-message to its child processor (line 13). The identifier *id* that is sent in the incoming y-message is afterward used as identifier in the x-message sent back to the child processor. That way we avoid the necessity to reserve a special, unique identifier for the Root Coordinator itself. Finally, at the end of each simulation cycle, the current simulation time t is set to the time of the next internal event (line 15) sent in the done-message (line 14). The Root Coordinator stops the simulation run, when its "end" is reached. However, how the end is defined can differ. For instance, a simulation run can end, if the simulation time exceeds a certain threshold (simulation stop time) or is set to infinity (i. e., all models become passive). Other simulation stop criteria are conceivable and subject of the implementation of the abstract simulator of ML-DEVS and the simulation system it is embedded into.

Although ML-DEVS supports a variable composition, the child of the Root Coordinator remains available during the entire simulation, as the Root Coordinator is not associated with any model that can carry out structure changes. However, the same applies for other variable structure variants of DEVS, such as DSDEVS, DSDE, dynDEVS, or DynPDEVS. If the modeler wants to specify a hierarchical ML-DEVS model the topmost component of which can change as well, the modeler has to introduce an additional Macro-DEVS model that takes care of the desired structure changes and becomes the new root of the hierarchical model. This additional component will, however, again remain unchanged during simulation.

9.4 Closure under Coupling of Multi-Level DEVS

If a coupled system or network of systems (i. e., system components) specified in a certain formalism can be specified as a basic or atomic system in the same formalism, then the formalism is *closed under coupling* [Zeigler et al. 2000, p. 149]. Recollect that *system specification* is a synonym for the term "model." So in other words: A modeling formalism is closed under coupling, if we can express any coupled model of the formalism by an *behaviorally equivalent* atomic model in that formalism (see Figure 9.10). This property allows us to use networks of systems (i. e., coupled models) as components in larger coupled systems (coupled models) and thus leads to a modular, hierarchical model construction in the first place.

As the abstract simulator of ML-DEVS already indicates and in contrast to claims of other authors such as Li et al. [2011], ML-DEVS is, in fact, closed under coupling.

Theorem 1 (Closure under coupling of ML-DEVS). *ML-DEVS is closed under coupling. Any Macro-DEVS model that couples model components specified as basic Micro-DEVS models can itself be specified as a Micro-DEVS model in ML-DEVS.*

So although the components of a Macro-DEVS model are—by Definition 9.2.2—Micro-DEVS models, we can also use Macro-DEVS models as components of other Macro-DEVS models in ML-DEVS. Notice that defining a Macro-DEVS model in the way that it can

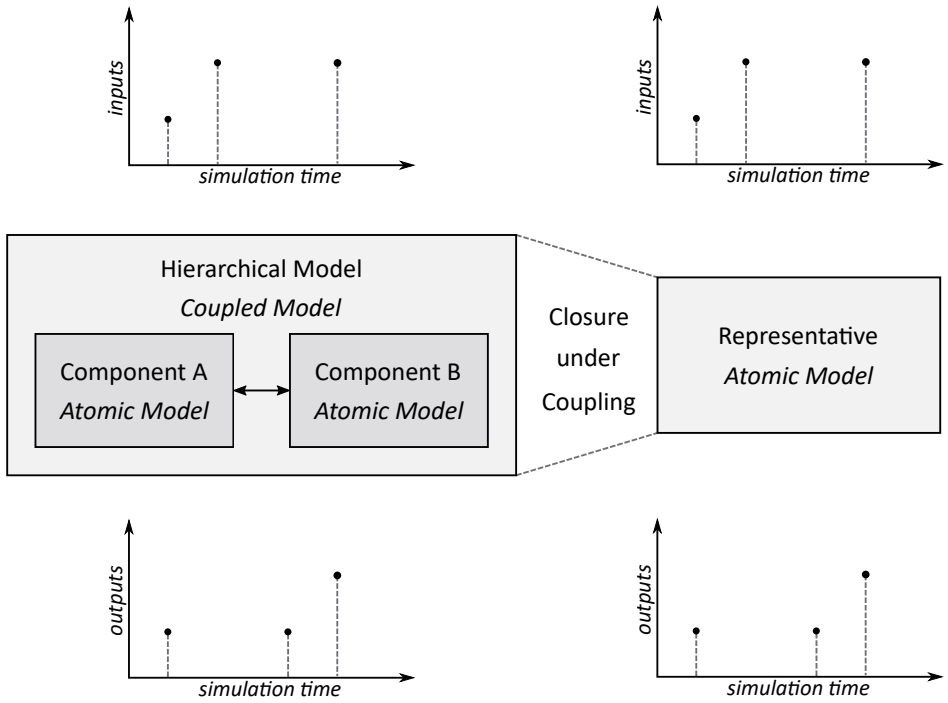

Figure 9.10: Closure under coupling allows us to replace an arbitrary hierarchical model by an behaviorally equivalent atomic model (here the model "Representative"). For a given input trajectory the respective atomic model produces the same output trajectory as the hierarchical model would do. In other words, we cannot distinguish between the atomic and original coupled model just based upon their output trajectories. Closure under coupling is a prerequisite for *flattening* hierarchical models and executing them more efficiently.

consist of Micro-DEVS and Macro-DEVS models at the same time would yield fundamental circularities in the definition of Macro-DEVS (cf. *circular definitions* in Section 3.1). For instance, we have to define a Macro-DEVS model in terms of its components in the set C, however, if the Macro-DEVS model can consist of other Macro-DEVS models, we have to define the set C in terms of Macro-DEVS models, which we cannot do, as we have to define C first. This is another, more subtle reason, why *closure under coupling* is important for a sound and rigorous definition of a modeling formalism that allows hierarchical coupling.

When we contemplate the closure under coupling of ML-DEVS, the couplings and influence schemes of Macro-DEVS models are of particular interest. In addition to the horizontal couplings (defined by multi-couplings), we have to consider the vertical couplings (up- and downward causation) between the micro and macro level as well. To formally prove the closure under coupling of ML-DEVS, we have to construct a Micro-DEVS model that is behaviorally equivalent to a given and arbitrary Macro-DEVS model or, in other terms, to demonstrate that any Macro-DEVS model can be expressed as a well-defined Micro-DEVS model (cf. Chow and Zeigler [1994]). Therefore, in the tradition of other DEVS variants, we systematically define all characteristic sets and functions of a given Macro-DEVS model in terms of a Micro-DEVS model, which can replace the given Macro-DEVS model.

Note that although a Macro-DEVS model can represent a Micro-DEVS model, i. e., as a stunted Macro-DEVS model without any components and up- and downward causation, we still have to show that we can construct a behaviorally equivalent Macro-DEVS model without components for any Macro-DEVS model with components (see also Section 9.2.2 for a motivation of the distinction between Micro-DEVS and Macro-DEVS models). In contrast to closure under coupling proofs of other DEVS variants, such as presented by Zeigler

et al. [2000, pp. 149–55], Chow and Zeigler [1994], Barros [1995a], or Uhrmacher [2001], the proof for ML-DEVS is rather verbose and complex[41]. The reason for this is twofold: First, we define ML-DEVS at the level of structured systems and with variable ports, thus we have to deal with structured sets and their variables in the proof. Second, Macro-DEVS models have a variable composition, whereas Micro-DEVS models are static except of their ports.

Proof. Given an arbitrary Macro-DEVS model n with

$$n = \langle id, XY, S_p, S_a, C, MC, s_{init}, \delta, \lambda, p, ta, sc, \lambda_{down}, v_{down}, act_{up} \rangle$$

with components that are Micro-DEVS models, we associate a Micro-DEVS model with n, called *resultant*:

$$resultant = \langle id, XY, S_p, S_a, s_{init}, p, \delta, \lambda, ta \rangle,$$

where

$$id = n.id$$

and

$$XY = n.XY.$$

So the identifier and the in- and output interface of the *resultant* are equivalent to those of the Macro-DEVS model n.

Remark. Please note that in the following we write id, XY, S_p, S_a, s_{init}, δ, λ, p, and ta rather than $resultant.id$, $resultant.XY$, $resultant.S_p$, $resultant.S_a$, and so on to refer to the corresponding components of the *resultant*'s defining tuple.

In contrast to coupled models of other DEVS variants, Macro-DEVS models have a state and behavior of their own, which have to be considered when constructing the *resultant*. Furthermore, the ability of a Macro-DEVS model to access the *interfaces* of its components has also to be taken into account. Hence, this information and the overall state of n become a part of the *resultant*'s private state. In addition, the set of the currently available components are part of the state, as they are necessary to determine the network state (see below). So the private state space S_p of the *resultant* is defined by

$$S_p = Q_C \times Q_n \times 2^{n.C},$$

where $Q_C = \prod_{c \in n.C} Q_c$ and for each c, Q_c is defined as follows:

$$Q_c = \{(s, a, xy, e) \mid s \in c.S_p,\ a \in c.S_a,\ xy \in c.XY, 0 \leqslant e \leqslant c.ta(s_p)\}$$

and with

$$Q_n = \{(s, a, xy, e) \mid s \in n.S_p,\ a \in n.S_a,\ xy \in n.XY, 0 \leqslant e \leqslant n.ta(s_p)\}.$$

Reflecting the absence of an intrinsic order in C, the set Q_C is defined as a *generalized Cartesian product* of the family of sets $\{Q_c\}_{c \in C}$ indexed by the set C, so as a set of functions

[41] Please also notice that, if given at all, we often find short outlines of closure under coupling proofs for different DEVS variants, e. g., Li et al. [2011] or Muzy and Zeigler [2014], rather than fully elaborated proofs. However, if a sound abstract simulator can be defined for a DEVS variant and concrete implementations of that simulator exist, we can assume that the respective variant of DEVS is closed under coupling, even though a formal proof is not given explicitly.

(see Appendix A.2.2). By specifying a function as a set of ordered pairs (see Notation A.2.2), for $q_c \in Q_C$ we can, for notational convenience, also write

$$q_c = \{(c, (s, a, xy, e)) \,|\, c \in C \wedge q_c(c) = (s, a, xy, e) \wedge q_c(c) \in Q_C\}$$

or simply

$$q_c = \{\ldots, (s_c, a_c, xy_c, e_c), \ldots\}.$$

For an element $q_n \in Q_n$ we also write

$$q_n = (s_n, a_n, xy_n, e_n).$$

Then, a private state $s_p \in S_p$ of the *resultant* can be written as follows:

$$s_p = \big(\underbrace{\{\ldots, (s_c, a_c, xy_c, e_c), \ldots\}}_{q_c \in Q_C}, \underbrace{(s_n, a_n, xy_n, e_n)}_{q_n \in Q_n}, \underbrace{comp}_{\subseteq n.C}\big). \tag{9.15}$$

Remark. For notational convenience and clarity, we write, in the following, s and a (instead of s_p and s_a) for private and accessible states, respectively. Furthermore, given a private state $s \in S_p$ of the *resultant* with $s = \big(\{\ldots, (s_c, a_c, xy_c, e_c), \ldots\}, (s_n, a_n, xy_n, e_n), comp\big)$ we simply write s_c, a_c, xy_c, e_c, and so on to access the different components. Accordingly, we use also q_c to refer to the set of all component-related tuples.

The accessible state space S_a of the *resultant* is equivalent to the accessible state space of the macro model:

$$S_a = n.S_a.$$

As a result the private state of the *resultant* contains also its accessible state, which is for keeping the proof as simple as possible. The initial state s_{init} of the *resultant* is $(s_{i,p}, s_{i,a})$, where the initial private state $s_{i,p} \in S_p$ is defined by

$$s_{i,p} = (q_{i,c}, q_{i,n}, comp_i)$$

with

$$\begin{aligned} q_{i,c} &= \big\{(c, (\pi_1(c.s_{init}), \pi_2(c.s_{init}), ports(c.p(\pi_1(c.s_{init}))), 0)) \,|\, c \in C\big\} \\ q_{i,n} &= (\pi_1(n.s_{init}), \pi_2(n.s_{init}), ports(\pi_1(n.s_{init}))), 0) \\ comp_i &= \pi_3(n.s_{init}) \end{aligned}$$

and $\pi_i(\ldots)$ being a projection on the i-th coordinate of an n-fold Cartesian product with $1 \leqslant i \leqslant n$ (see Appendix A.1.7) and where the initial accessible state $s_{i,a} \in S_a$ is defined by

$$s_{i,a} = \pi_2(n.s_{init}).$$

The function *ports* returns a set of key-value pairs for a given component c (incl. the macro model n) based on the component's port selection function, such that only available ports appear as keys in the set and the symbol ε is the value in each pair, i. e.,

$$ports(\mathcal{P}') = \big\{(pn, \varepsilon) \,|\, pn \in \mathcal{P}'\big\} \tag{9.16}$$

with $\mathcal{P}'$ being a set of port names that is returned by the port selection function of the corresponding component, i. e., $\mathcal{P}' = c.p(s_c)$.

Although ML-DEVS allows a variable composition, the set $n.C$ contains all potential components of n regardless of their actual availability during simulation. However, at each time in the simulation, only a finite number of available components exists. The availability of components is determined by the structure change function sc of n based upon its current private state. Only those can act and thus change their state and influence the macro model. Now given a *resultant*'s private state $s \in S_p$, as defined in Equation 9.15, we define the set of active (available) components $ACT(s) \subseteq n.C$ in state s by

$$ACT(s) = comp.$$

As we will see later, *comp* contains the currently available components.

Both MICRO-DEVS model and MACRO-DEVS model have variable ports whose availability during simulation is determined by their port selection functions p. We define $p\colon S_p \to 2^{\mathcal{P}}$ of the *resultant* by

$$p(s) = n.p(s_n).$$

So the available ports of n are the available ports of the *resultant* . The time advance function $ta\colon S_p \to \mathbb{R}_0^+ \cup \{\infty\}$ is defined by

$$ta(s) = min\{\sigma_c \,|\, c \in \{n\} \cup ACT(s)\} \text{ with } \sigma_c = c.ta(s_c) - e_c, \tag{9.17}$$

where σ_c is the time remaining to the next internal event in c.

For defining the remaining functions of the *resultant*, up- and downward activation are of particular interest, as they can lead to the activation of n and its components. Thus, these activation schemes have to be considered when determining how the *resultant*'s overall state is evolving. Also, the intensional multi-couplings of n have to be translated into a concrete coupling scheme to identify influencing and influenced components for a given private state s of the *resultant*. For this, the current interface instances of the available components have to be determined given a *resultant*'s private state s, which contains all the required information (see above):

$$ifaces(s) = \bigcup_{c \in \{n\} \cup ACT(s)} \big\{(c.id, (xy_c, a_c))\big\}, \tag{9.18}$$

where $ifaces(s) \subseteq I^n$ (see Equation 9.5) of the MACRO-DEVS model. Algorithm 9.5 shows how multi-couplings are then resolved. The result is, for each available component c, a set of influencers $I_c \subseteq \{n\} \cup ACT(s)$ with $c \in \{n\} \cup ACT(s)$ and for each combination of influencing component i and influenced components c a port-to-port mapping $P_{i,c}\colon \mathcal{P}_i \to \mathcal{P}_c \cup \{\varnothing\}$ with $c \notin I_c$. The latter determines which port of an influencing component is coupled to which port of the influenced component or is not coupled (indicated by $\varnothing$, i. e., $P_{i,c}(pn) = \varnothing$) according to the multi-couplings. In contrast to other DEVS variants with extensional couplings, the derived concrete coupling scheme in ML-DEVS is not invariant and has to be determined for each private state s of the *resultant*.

In ML-DEVS exists only one state transition function that is invoked when an external, internal, or confluent transition has to be performed. Before we construct the state transition function $\delta\colon Q \times XY^b \to S_p \times S_a$ of the *resultant* with $Q = \{(s, a, e) \,|\, s \in S_p, a \in S_a, 0 \leqslant e \leqslant ta(s)\}$, we partition the available components into further subsets based on the given private

state s of the *resultant* and an input bag $x^b \in XY^b$:

$$
\begin{aligned}
&IMM(s) = \{c \in \{n\} \cup ACT(s) \,|\, \sigma_c = ta(s)\} && \text{(imminent components incl. n)}\\
&DOWN(s) = \{c \in ACT(s) \,|\, \sigma_n = ta(s) \wedge x_{c,n} \neq \varnothing\} && \text{(components activated by n)}\\
&INF(s, x^b) = \left\{c \in ACT(s) \,|\, x_c^b \neq \varnothing\right\} \cup DOWN(s) && \text{(influenced components)}\\
&INT(s) = IMM(s) \backslash INF(s, x^b) && \text{(imminent, but not influenced components)}\\
&EXT(s) = INF(s, x^b) \backslash IMM(s) && \text{(influenced, but not imminent components)}\\
&CONF(s) = IMM(s) \cap INF(s, x^b) && \text{(imminent and influenced components)}\\
&UN(s) = ACT(s) \backslash IMM(s) \backslash INF(s, x^b) && \text{(not imminent, not influenced components)}
\end{aligned}
$$

where σ_c as defined in Equation 9.17 and

$$
\text{with } x_{c,n} = \begin{cases} xy & \text{if } \exists!(id, xy) \in n.\lambda_{down}(s_n, s^n(q_c, comp)) \colon c.id = id \\ \varnothing & \text{otherwise} \end{cases} \tag{9.19}
$$

where $s^n(q_c, comp)$ returns the current network state and is defined by

$$
s^n(q_c, comp) = \bigcup\nolimits_{c \in comp} \left\{(c.id, a_c)\right\} \tag{9.20}
$$

and where the input bag x_c^b of a component c is defined by

$$
x_c^b = \left(\biguplus\nolimits_{i \in IMM(s) \cap Infl_c} \biguplus\nolimits_{x \in i.\lambda(s_i)} \bigcup\nolimits_{(pn,v) \in x} \left\{(P_{i,c}(pn), v) \,|\, P_{i,c}(pn) \neq \varnothing\right\}\right) \uplus x_{c,n}^b \tag{9.21}
$$

with

$$
x_{c,n}^b = \begin{cases} \biguplus_{x \in x^b} \bigcup_{(pn,v) \in x} \left\{(P_{n,c}(pn), v) \,|\, P_{n,c} \neq \varnothing\right\} & \text{if } n \in Infl_c \\ \varnothing & \text{otherwise} \end{cases} \tag{9.22}
$$

and where $\uplus$ refers to the sum of two bags (see Appendix A.1.4). Components that are activated downward by the downward output function λ_{down} of n are treated as influenced models, complementing those components that receive regular inputs. Thus, downward activation is considered as classic external event (cf. Section 9.3.2).

Now, given a private state $s = \left(\{\ldots, (s_c, a_c, xy_c, e_c), \ldots\}, (s_n, a_n, xy_n, e_n), comp\right)$ and $c \in n.C$, an accessible state $a \in S_a$, an elapsed time e with $0 \leqslant e \leqslant ta(s)$, and an input bag $x^b \in XY^b$, we can define the *resultant*'s state transition function as follows:

$$
\delta(\underbrace{(s, a, e)}_{q \in Q}, x^b) = \underbrace{\Big(\big(\underbrace{\{\ldots, (s_c{}', a_c{}', xy_c{}', e_c{}'), \ldots\}}_{q_c{}' \in Q_C}, \underbrace{(s_n{}', a_n{}', xy_n{}', e_n{}')}_{q_n{}' \in Q_n}, comp'\big), a'\Big)}_{s' \in S_p},
$$

where for each $c \in C$:

$$
(s_c{}', a_c{}', xy_c{}', e_c{}') = \begin{cases} (c.\delta((s_c, a_c^*, e_c + e), x_c^b \uplus x_{c,n}), xy_c^*, e_c + e) & \text{if } c \in EXT(s) \\ (c.\delta((s_c, a_c^*, e_c + ta(s)), \varnothing), xy_c^*, 0) & \text{if } c \in INT(s) \\ (c.\delta((s_c, a_c^*, e_c + ta(s)), x_c^b \uplus x_{c,n}), xy_c^*, 0) & \text{if } c \in CONF(s) \\ (s_c, a_c^*, xy_c, e_c + e) & \text{if } c \in UN(s) \\ (s_c, a_c, xy_c, e_c) & \text{otherwise} \end{cases}
$$

with x_c^b as defined in Equation 9.21, $x_{c,n}$ as defined in Equation 9.19, and $a_c^* \in c.S_a$ being the accessible state of component c merged with the global information announced by the macro model n (see value coupling in Section 9.2.2) defined by

$$\forall v \in c.\mathcal{V}_a\colon\ a_c^*(v) = \begin{cases} s_n(v) & \text{if } \exists v' \in n.v_{down}(s_n)\colon\ v = v' \\ a_c(v) & \text{otherwise} \end{cases}$$

and with xy_c^* being the information about the ports available in the new state defined by

$$xy_c^* = ports(c.p(\pi_1(s_c{}'))), \tag{9.23}$$

where the function *ports* is defined as in Equation 9.16. The updated port information of each active component xy_c^* is used to determine $ifaces(s')$. The last case—labeled with otherwise—refers to all components that are not available in the current state, i.e., $c \in n.C \backslash ACT(s)$. For n we define:

$$(s_n{}', a_n{}', xy_n{}', e_n{}') = \begin{cases} (n.\delta((s_n, a_n, e_n + e), x^b, s^n(q_c{}', comp)), xy_n^*, 0) & \text{if } n \notin IMM(s) \wedge (x^b \neq \varnothing) \\ & \quad \vee n.act_{up}(s_n, s^n(q_c{}', comp)) = \top) \\ (n.\delta((s_n, a_n, e_n + ta(s)), x^b, s^n(q_c{}', comp)), xy_n^*, 0) & \text{if } n \in IMM(s) \\ (s_n, a_n, xy_n^*, e_n + e) & \text{otherwise.} \end{cases}$$

Please note that for brevity we directly use the new private state $s_c{}'$ of component c and the new $q_c{}'$ instead of invoking the respective state transition functions again. For $q_c{}'$ and *comp* we define $s^n(q_c{}', comp)$ as in Equation 9.20 and xy_n^* is defined as in Equation 9.23. Please remember that the macro model is accessing the public state of its components, which are part of the *resultant*'s state. The new composition $comp'$ is defined by

$$comp' = \pi_1(n.sc(s_n{}', s^n(q_c{}', comp))).$$

Finally, the new accessible state of the *resultant* is the new accessible state of the macro model n, i. e.,:

$$a' = a_n{}'.$$

We obtain the output of the *resultant* for a given private state s by collecting all the external outputs (those that are routed to the original macro model) of the imminent components and the output of the Macro-DEVS model itself in a bag:

$$\lambda(s) = \left(\biguplus_{i \in (IMM(s) \backslash \{n\}) \cap Infl_n} \ \biguplus_{y \in i.\lambda(s_i)} \ \bigcup_{(pn,v) \in y} \left\{ (P_{i,n}(pn), v) \mid P_{i,n}(pn) \neq \varnothing \right\} \right) \uplus n.\lambda(s_n).$$

As shown above, we can construct a well-defined Micro-DEVS model for a given Macro-DEVS model, so ML-DEVS is closed under coupling.

9.5 Systems Specified by Multi-Level-DEVS

As done for DEVS (and P-DEVS) by Zeigler et al. [2000, pp. 139–44], we illustrate the meaning of ML-DEVS as a formalism for systems specification by describing the systems that is specified by ML-DEVS from a more general, system-theoretic point of view. However, as we define ML-DEVS at the level of structured systems, we assume that the in- and outputs of the general system, specified by ML-DEVS, are structured in similar way like in

ML-DEVS (e. g., by using Cartesian products) or that there exists a meaningful mapping from the unstructured ("flat") in- and outputs of the general system to the structured in- and outputs of ML-DEVS. Please note that the same applies for other DEVS variants that are defined at structured system level, even if the ports (in- and output variables) of these variants are not variable.

Now let m be a MICRO-DEVS model—the basic model of ML-DEVS—with

$$m = \langle id, XY, S_p, S_a, s_{init}, p, \delta, \lambda, ta \rangle.$$

Then m describes a general input/output system (see Zeigler et al. [2000, pp. 99–132] for more details on I/O systems and the hierarchy of system specifications), denoted by S, with

$$S = \langle T, X, \Omega, Y, Q, \Delta, \Lambda \rangle$$

that is time invariant, has a Moore-like output function (see Appendix B.2), and that has the following characteristics (cf. Zeigler et al. [2000, pp. 139–44]):

1. The time base T is the set of real numbers $\mathbb{R}$ (or a subset of $\mathbb{R}$).
2. The input set X is
$$X = m.XY^b \cup \{\varnothing\},$$
where $m.XY^b$ is a set of bags over the elements in $m.XY$, $\varnothing \notin m.XY$ and $\varnothing$ denotes the nonevent.
3. The output set Y equals the input set X.
4. The state set Q is a set of total states with
$$Q = \{(s_p, s_a, e) \mid s_p \in m.S_p, s_a \in m.S_a, 0 \leqslant 0 \leqslant m.ta(s_p)\}.$$
5. The set Ω of admissible input segments is the set of bags of all discrete event segments over $m.XY$ and T.
6. The state trajectories are piecewise constant segments over $m.S_p \times m.S_a$ and T.
7. The output trajectories are bags of discrete event segments over $m.XY$ and T.
8. The state trajectory is defined as follows: Let
$$\omega \colon \langle t_i, t_f] \to X$$
with $\omega \in \Omega$ and $X = m.XY^b \cup \{\varnothing\}$ be a discrete event segment and let
$$q = (s_p, s_a, e)$$
with $q \in Q$ be the state of the system S at time t_i. Then we define
$$\Delta \colon Q \times \Omega \to Q$$
by
$$\Delta(q, \omega_{\langle t_i, t_f]}) = \begin{cases} (s_p, s_a, e + t_f + t_i) \\ \qquad \text{if } e + t_f - t_i < m.ta(s_p) \wedge (\nexists t \in \langle t_i, t_f] \colon \omega(t) \neq \varnothing) \\ \Delta((m.\delta((s_p, s_a, e + m.ta(s_p)), \varnothing), 0), \omega_{[t_i + m.ta(s_p) - e, t_f]}) \\ \qquad \text{if } e + t_f - t_i \geqslant m.ta(s_p) \wedge (\forall t \in \langle t_i, t_i + m.ta(s_p) - e] \colon \omega(t) = \varnothing) \\ \Delta((m.\delta((s_p, s_a, e + t - t_i), p(\omega(t), q)), 0), \varnothing_{[t,t]} \bullet \omega_{\langle t, t_f]}) \\ \text{if } (\exists t \in \langle t_i, \min(t_f, t_i + m.ta(s_p) - e)) \, \forall t' \in \langle t_i, t) \colon \omega(t') = \varnothing) \wedge \omega(t) \neq \varnothing \\ \Delta((m.\delta((s_p, s_a, e + m.ta(s_p)), p(\omega(t), q)), 0), \varnothing_{[t,t]} \bullet \omega_{\langle t, t_f]}) \\ \qquad \text{if } e + t_f - t_i \geqslant m.ta(s_p) \wedge (\forall t' \in \langle t_i, t] \colon \omega(t) = \varnothing) \wedge \omega(t) = \varnothing \end{cases}$$

with $t = t_i + m.ta(s_p) - e$ and where the function

$$p\colon X \times Q \to X$$

returns a bag of inputs taking the availability of ports into account with

$$p(\omega(t), q) = \biguplus_{x \in \omega(t)} \bigcup_{(pn,v) \in x} \{(pn, v) \mid \exists pn \in m.p(s_p)\}. \tag{9.24}$$

The first case refers to the absence of any event, where simply the elapsed time is advanced. The second, third, and fourth case refer to an internal, external, and confluent (internal and external event at the same time) event, respectively. Note that in all of the three latter cases the state transition function of the model m is invoked, however different arguments are passed to the function. In addition, global information—if used—plays a special role, as it is set by the superordinate Macro-DEVS model. Alternatively, the Micro-DEVS model can be adapted in a way that global information is provided as an additional input by the environment, similar to the procedure of translating nonmodular multicomponent DEVS models into modular DEVS models as described by Zeigler et al. [2000, pp. 161–2].

9. The output function Λ with $\Lambda:\ Q \to Y$ of the system S is defined as follows: Let $q = (s_p, s_a, e)$ be a state of the system S with $q \in Q$, then we define

$$\Lambda(q) = \begin{cases} p^*(m.\lambda(s_p), q) & \text{if } e = m.ta(s_p) \\ \varnothing & \text{otherwise,} \end{cases}$$

where the function $p^*:\ Y \times Q \to Y$ returns a bag of outputs taking the availability of ports into account and is defined just like the function p in Equation 9.24, because $X = Y$.

Due to the closure under coupling of ML-DEVS, the general system described by a Macro-DEVS model can be specified as above, if we specify the Macro-DEVS model by the means of a behaviorally equivalent Micro-DEVS model, as shown in Section 9.4.

9.6 Summary

This chapter presents our substantial revision of the modeling formalism ML-DEVS for multi-level and variable structure modeling and its rigorous formal foundation. For this, the chapter first describes how models are specified in the formalism and explains informally how the models are executed. Then the execution semantics of ML-DEVS is described formally by the means of an abstract simulator, as characteristic of a DEVS-based modeling formalisms. Finally, the closure under of coupling of ML-DEVS is proven and the general I/O systems specified by ML-DEVS are briefly outlined.

In my doctoral studies, ML-DEVS served, among other things, as a vehicle for the implementation of the concept of intensional interface couplings as introduced in Section 8.7 in the form of multi-couplings. We show that intensional coupling definitions and interfaces can be exploited for variable structure modeling and that they are an expressive and powerful tool, particularly when it comes to maintaining structural consistency in variable structure models. In addition to multi-couplings, ML-DEVS has further vertical coupling schemes that allow a communication between different levels by the means of up- and downward causation (multi-level modeling). These "vertical couplings" also have an intensional nature, since they are evaluated and translated during simulation. However, the intensional coupling definitions are not restricted to ML-DEVS. They can also be used in other modeling formalism, especially when they are defined in a similar way (system theoretically) or make use of the reactive systems metaphor.

Part III

Conclusion and Future Work

10 Conclusion

> We simply don't have enough data to form a conclusion.
>
> Mike A. Lancaster

This chapter concludes the thesis by summarizing and discussing the results. By doing so, we come full circle.

10.1 Conclusion and Discussion

During my doctoral studies, we have investigated and evaluated possibilities as well as limitations of combining traditional model composition[1] —which assumes static model structures— and a composition over time, as it occurs in variable structure models. Traditional model composition, i. e., the construction of correct, executable simulation models from predefined, exchangeable, retrievable, and often customizable model components via a composition methodology or formalism, takes place at configuration time [Petty & Weisel 2003a], before the execution of the respective model composition (simulation). Variable model structures and interfaces, on the other hand, are a runtime (or execution time) phenomenon, i. e., structure changes can only occur during the execution of variable structure models [Hu et al. 2005]. Although the modeler specifies the concrete circumstances under which a variable structure model changes its structure, i. e., makes a transition from one structure incarnation to another, the modeler does not know beforehand when these model transition exactly take place during simulation[2]; How and when a model changes its structure in a simulation is determined by the initial state and structure[3] as well as its behavior, i. e., its state, input, and output trajectory. The latter of which (the model behavior) we want to keep separate from compositional descriptions in traditional model composition (cf. Röhl and Uhrmacher [2008]). When thinking about components encapsulating models with a variable structure or interface as well as time-variant composition and communication patterns, among others, the following questions come to mind:

- How and to which extent can we reflect structure variability by a composition methodology or formalism?
- What impact has structure variability on the assurance of correctness and structural consistency of a model composition beyond the initial model state and configuration? Or in other words: Can we still make statements on the correctness and structural consistency of a model composition at configuration time, when the model structure can change during simulation?

Another focus of my doctoral studies was the development, implementation, and evaluation of a flexible, yet expressive coupling mechanism that allows specifying couplings in variable structure and interface models concisely, without taking care about structural consistency during model execution.

As starting points for our studies, i. e., the combination of traditional model composition and models with time-variant structures and interfaces, we used the following two approaches:

1. The composition framework COMO (Component-based Modeling) and its underlying formalism/language for specifying and analyzing components, component interfaces, and compositions [Röhl & Uhrmacher 2008; Steiniger & Uhrmacher 2013].
2. Ideas about the modular-hierarchical modeling formalism ML-DEVS for (parallel) discrete event, multi-level, and variable structure simulation [Steiniger et al. 2012; Steiniger & Uhrmacher 2016; Uhrmacher et al. 2007].

The former approach does not only allow a platform- and modeling-formalism-independent[4] specification of model compositions [Röhl 2006], in a compact, set-theoretical notation, it

[1] That is a model composition as described by Verbraeck [2004] and others.

[2] Please note that one of the main reasons for using simulation is that the corresponding model cannot be examined analytically (cf. Law and Kelton [2000, p. 5]).

[3] The ports, components, and couplings that are available at the beginning of a simulation.

[4] The composition methodology shipped with COMO is virtually independent of a concrete modeling formalism. However, the actual component models, i. e., the models that are encapsulated by the components, need to be implemented in suitable source formalisms, so that an executable simulation model in a target formalism can be derived from the compositional descriptions and the model implementations. These source formalism has at least to support the concept of ports.

also incorporates component interfaces as first-class abstractions, emphasizing the role of interfaces in modeling complex systems. The explicit specification of such interfaces allows us to (i) compose components that adhere to their interface specification regardless of their actual implementation following Verbraeck [2004] and (ii) check the correctness of a model composition before an executable simulation model is derived and executed [Röhl & Uhrmacher 2008]. However, the approach assumes a static model structure and time-invariant interfaces (i. e., static ports) [Röhl 2008, p. 113]. The author notices that all checks for analyzing the (syntactic) correctness of a composition are carried out before the actual model execution. In the case of a variable composition and dynamic interfaces, this would mean to analyze all possible incarnations of the model structure and interfaces.

The modeling formalism ML-DEVS, on the other hand, incorporates variable structures and interfaces as well as multi-level modeling into parallel discrete event simulation, in the tradition of P-DEVS [Chow & Zeigler 1994]. Both structure variability and multi-level modeling are promising for modeling complex adaptive systems, such as smart environments, eukaryotic cells, or entire populations.

In Chapter 7 and Steiniger and Uhrmacher [2013] we show that, if we exclude the opportunity to create new, arbitrary ports and components during model execution, the superset of potential ports and components that can become available during execution is know beforehand and thus can be specified at that time. Therefore, we can specify the interface of a component with a variable interface by defining all its ports regardless of their availability during model execution. Making use of a superset of ports in an interface specification does not conflict with the idea of a well-defined interface, since all possible communication capabilities of a component are part of its interface specification and thus can be analyzed before and when deriving executable simulation models. We pursue the same approach when specifying the potentially available subcomponents of a composite component, the internal composition of which can change during simulation. In addition, the initially available components and ports need to become a part of the compositional description to allow us deriving executable simulation models from these descriptions and the implementations of the corresponding component models[5]. Based on the initially available components and ports, we can derive the initial model structure, which is necessary for executing the derived model properly, i. e., running a simulation.

Another important aspect of compositions with a variable structure are the "communication channels" between the components, as these channels (i. e., couplings or connections) can change as well; either as a direct result of the variable composition (a model component that serves as a communication partner for another model component may exist at a certain time but not at another) or by the desire of the modeler (who may want to model non-permanent couplings or connections). To reflect this structure variability in the specification of a communication structure, we introduced the novel concept of an intensional coupling definition, which is based on port names [Steiniger & Uhrmacher 2013] and inspired by multi-couplings as used in ρ-DEVS [Uhrmacher et al. 2006] and the original ML-DEVS [Uhrmacher et al. 2007]. These intensional coupling definitions do not describe explicit, concrete couplings, but serve as templates from which concrete couplings can be derived during model execution. The availability of ports with matching names and value ranges implies the existence of concrete couplings, into which intensional couplings are eventually transformed. This transformation of intensional couplings need to be done by the simulation algorithm whenever the structure of the model compositions changes during simulation, so whenever events occur; which is and cannot be the responsibility of COMO. Hence, the target formalism for the transformation of compositional descriptions and component

[5] This implies that there is a transformation between each source formalism and the target formalism, which creates a behaviorally equivalent model. Vangheluwe [2000], Feng, Zia, and Vangheluwe [2007], Syriani and Vangheluwe [2010], or Syriani and Vangheluwe [2013] show that transformations between various modeling formalisms exist and DEVS variants are particularly suitable as a common target formalism.

implementations needs to incorporate such an intensional coupling mechanism as well. At this point, the original version of ML-DEVS came into play, because it served as a suitable target formalism that already made use of an intensional couplings. Since only consistent couplings are derived and established during simulation (see below), structural consistency is guaranteed, already at configuration time (correctness by construction)[6], as long as the implementations of the involved components adhere to the corresponding compositional descriptions[7]. The fact that the superset of ports and the superset of components are known at configuration time, allows us to analyze the consistency of potential coupling candidates regardless of their actual existence during execution. This is useful to indicate potential modeling errors, which will be otherwise unnoticed, since inconsistent couplings are not established.

Another, crucial contribution of my work is the fundamental revision and extension of the multi-level modeling formalism ML-DEVS and its formal foundation, as described in Chapter 9. As mentioned earlier in this section, ML-DEVS served as a target for the model transformations and synthesis carried out by the also revised composition framework COMO (see Section 7.4), when creating executable simulation models. The formalism was also used as a suitable source formalism for "implementing" the components, which were eventually composed. In addition, we also used ML-DEVS detached from COMO to directly create multi-level, discrete event, and variable structure models based upon which simulation studies were conducted (see Krüger et al. [2012] and Steiniger et al. [2012]). The extension of ML-DEVS primarily involved the elevation of interfaces and the introduction of a more expressive intensional coupling definition, i. e., multi-couplings, which is based on runtime instances of interfaces and which reflects and surpasses the aforementioned coupling mechanism introduced in COMO. Although this mechanism is based on ideas from ρ-DEVS [Uhrmacher et al. 2006], the original ML-DEVS [Uhrmacher et al. 2007], and our revision of COMO [Steiniger & Uhrmacher 2013], it extends these ideas considerably. In the revision of COMO, we merely use port names for defining couplings, whereas in the revised ML-DEVS we can make use of identifiers and public states of model components to define and constrain couplings. Moreover, we can incorporate arithmetic and other functions in the coupling definition (see the examples in Section 9.2.2). This allows the modeler to specify variable structure models more naturally and concisely, without the need to introduce auxiliary "marker ports" or encode semantic information about the model state into port names on the hand and taking care about structural consistency at the other hand, which is otherwise hard to achieve and assure (cf. Muzy and Zeigler [2014]). As described earlier, the concrete coupling scheme is derived from the intensional coupling definition by the simulation algorithm during mode execution, whenever the structure of the composed model changes. For this, only model components and ports that are available after a structure change are considered when deriving the new concrete coupling scheme from the intensional coupling definition. The consistency of a derived concrete coupling scheme is guaranteed by the corresponding transformation, which takes all constraints and conditions into account. A coupling that would violate any of these constraints and conditions is not created in the first place (e. g., if the value ranges of ports are not matching). Although the coupling mechanism described above was implemented in ML-DEVS, it can be used by any other system-theoretic model approach following the metaphor of reactive systems, such as SysML. In addition, an intensional coupling definition can also be used in static structure models. Here, the transformation of an intensional coupling definition into a concrete coupling scheme has to be performed only once, at the beginning of the model execution.

[6] For instance, a concrete coupling between ports whose names match is only established if all other coupling conditions are fulfilled, i. e., the ports are compatible and their directions are correct.

[7] This adherence refers to the refinement relationships described by Röhl and Uhrmacher [2008].

11 Future Work

> The limiting surface of one thing is the beginning of another.
>
> Leonardo da Vinci

This chapter mentions and discusses interesting topics and questions that arose during my doctoral studies, which, however, I was not able to follow-up and address due to the focus of the thesis and the limited time.

Some of these topics, such as the usability and intelligibility of modeling approaches, are research topics on their own, worth of further investigations and studies, e. g., comprehensive user studies. In addition, the usability and intelligibility of a modeling approach concerns other disciplines such as behavioral science just as much as computer science, if not more, asking for an interdisciplinary cooperation of experts from the respective disciplines.

11.1 Usability Evaluation of Modeling Approaches

When developing a novel modeling approach, the creators face the challenge of motivating and evaluating this novel approach, properly and unbiasedly, to show that it offers benefits with respect to the existing modeling methodology (*scientific principles* and *research practices*). This task is anything but easy or convenient.

From a rather theoretical or technical perspective, we can analyze the "expressive power" of a modeling approach; in the sense whether phenomena of interest can in principle be expressed in a certain modeling approach or methodology. Since modeling can be considered as a computational problem, we can determine the underlying formal model of computation, for a given modeling approach (modeling formalism). The *Church–Turing thesis* basically asserts that any function that can be computed by a computer can also be computed by a *Turing machine* (cf. Kleene [1967, p. 232] or Copeland [2015])[1]. For instance, the π-calculus, a general-purpose modeling formalism for modeling concurrent processes, is *Turing complete* [Milner 1992]. However, "beware of the Turing tar-pit in which everything is possible but nothing of interest is easy" [Perlis 1982]. Hence, assessing a modeling approach by its mere expressive power, in terms of *computability*, is often not useful, especially when the modeling approach is (already) Turing complete.

As modelers we are more interested in the *practical expressivity* as defined by Farmer [2007], i. e., "the measure of how readily ideas can be expressed in the logic [modeling approach]." In other words, practical expressivity allows us to express the significant model hypotheses easily and concisely and is closely related to the notion of conciseness or succinctness (see also Warnke et al. [2015]).

Often modeling methodologists argue that their modeling approaches are more intuitive or easier to use[2], cf. Huhns and Singh [1998], Kasputis and Ng [2000], Silverman et al. [2001], Barros [2003], or Maus et al. [2011], without, however, elaborating on these personal assessments. This does not necessarily mean that these assessments are incorrect or incomprehensible from the perspective of the methodologists, it simply makes it hard to verify these claims impartially. To evaluate the intuitiveness, usability, or ineligibility of a modeling approach unbiasedly, extensive user studies are required, which are costly. Designing such user studies is not trivial and requires additional psychological expertise on top of knowledge about the modeling approach at hand. Aggravating this situation, not just any users can be asked to participate in such user studies, but users with a background in modeling. Moreover, some users may already have worked with similar modeling approaches, whereas other may have worked with entirely different modeling metaphors. For instance, modeling in the π-calculus or its variants (*concurrent processes*) is quite different from modeling in DEVS or its variants (*reactive systems*). Regardless of the users' background, there will also always be a learning curve when using a novel modeling approach for the first time. However, the steepness of this learning curve may differ from modeling approach to modeling approach.

We think that the aforementioned challenges and considerations are the reason why we do not find much work dedicated to an empirical evaluation of the usability or ineligibility of modeling approaches, especially in the realm of modeling and simulations. Figl, Mendling, and Strembeck [2009] or Schalles [2013] are some exceptions, which, however, focus on visual modeling approaches. We think that these visual modeling approaches as well as domain-specific modeling languages are easier to learn and understand, also by novice modelers, than more formal or general-purpose modeling approaches (cf. Steiniger et al. [2014] or Warnke et al. [2015]); where ML-DEVS has to be counted as a representative of the latter group.

[1] Even more, Kaye, Laflamme, and Mosca [2007, p.6̃] state that "a quantum Turing machine can efficiently simulate any realistic model of computation."

[2] than existing or established modeling approaches

11.2 Intensional Definitions

In Chapter 9, we have briefly discussed that an extensional definition of a set of potential components of a coupled model in a hierarchical modeling formalism, such as ML-DEVS, does not cope well with the desired flexibility by "creating model components on demand," when we are dealing with variable structures. So instead of enumerating the defining tuples of all potential components of a coupled model explicitly and in advance, we propose an intensional definition of a set of (sub-)components, denoted by C in the following. This means, by defining properties all admissible components (elements of the set C) must satisfy; in terms of a *set-builder notation* in set theory (cf. Rosen [2007, pp. 111–2]) and similar to the intensional definition of couplings in hierarchical models as introduced in Section 9.2.2. In the most general case, the set C contains all conceivable and valid model definitions in a certain modeling formalism, such as ML-DEVS. Hence, C can be considered as a universe as known from set theory; a universe of models. From a modeler's perspective, however, it is desirable to constrain the kind of (sub-)models that can be created and work with a subset of this universe rather than the universe itself. For instance, mitochondria—eukaryotic cell organelles—do not contain technical components such as lithium-ion batteries. This limitation can be achieved by using predicates (Boolean-valued functions) that take the "types" or other properties of models into account and indicate which model belongs to the set C. By types we refer to a different concept than the traditional distinction between atomic and coupled models in the realm of DEVS. Here, types shall refer to classes of models the members of which share common properties, such as the availability of certain ports, or similar behavior patterns. Moreover, such types are specific to the application domain; in a cell we find different types of submodels than in a smart environment. In ML-DEVS, we can make use of the runtime interfaces and accessible states of models for defining such types or predicates.

Example 11.2.1
Suppose we want to model a mitochondrial network (such as described in Chapter 9) of an eukaryotic cell, while ignoring all other cell components (i. e., other organelles of this cell that are not mitochondria). In this case, the set of components, denoted by C, of the coupled model representing the cell should only consists of mitochondria and nothing else. To achieve this, we can define the set C as follows:

$$C = \{c \in C^* \mid mito(c)\}, \tag{11.1}$$

where C^* is the superset of all conceivable and consistent Micro-DEVS models (and Macro-DEVS models due to the closure under coupling of ML-DEVS) and

$$mito : C^* \to \{\top, \bot\} \tag{11.2}$$

is a unary predicate determining whether or not a Micro-DEVS *model is a mitochondrion and thus a member of the set C. The actual definition of the predicate, i. e., the mapping, is not easy and thus omitted here intentionally.*

Here, a higher-level constraint language, in the style of OCL[3], could be helpful for modelers to easily define such predicates.

As Section 7.4.1 indicates, intensional definition techniques can also be employed for defining the superset of ports, i. e., the interface of model components.

[3] Object Constraint Language

11.3 Improvements on Multi-Level DEVS

When introducing ML-DEVS and an intensional coupling mechanism in Chapter 9, we were focusing on a sound and rigorous formal definition[4] rather than on providing a particularly intuitive modeling formalism. Although ML-DEVS is powerful and allows us to explicitly and concisely express phenomena such as variable structures and emergent behavior, we acknowledge that the learning curve of creating executable simulation models by using the formalism can be rather steep, due to the formalism's complexity, especially for modelers without a background in DEVS. In addition to the idea outlined in Section 11.2, we briefly discuss, in the following, two further ideas that may can help to decrease the steepness of the learning curve.

11.3.1 Activation Events

In ML-DEVS, upward activation is achieved by the act_{up}-function of Macro-DEVS models. This function checks the fulfillment of invariants that are defined on the accessible states of the components of a Macro-DEVS model and the Macro-DEVS model's private state (see Section 9.2.2). When such invariants are violated the upward activation function act_{up} returns "$\top$" (i. e., true) and the state transition function of the respective Macro-DEVS model is triggered (i. e., the Macro-DEVS model is activated). However, the information about which invariants are violated is not passed to the state transition function. If necessary, the modeler needs to re-evaluate the invariants in the state transition function to determine which of the invariants were violated. This ultimately leads to a duplication of the evaluation logic (in the upward activation and state transition function), which makes the definition of the state transition function more verbose than necessary and can lead to inconsistencies.

To ease the modeling, we can introduce another type of events, i. e., *activation events*, which can occur in Macro-DEVS models. The upward activation function can then be defined in the way that it maps to bags of these activation events instead of truth values, i. e.,

$$act_{up}\colon\ S_p \times S^n \to A^b,$$

where A is a set of activation events. If invariants are violated the bag of activation events is nonempty and the Macro-DEVS model is triggered (i. e., the empty bag is returned if the Macro-DEVS model is not activated upward). At the same time, the signature of the state transition function δ of a Macro-DEVS model can be extended by these activation events:

$$\delta\colon\ Q \times S^n \times XY^b \times A^b \rightsquigarrow S_p \times S_a.$$

Each activation event represents one invariant that is violated, i. e., the event that causes the upward activation of the Macro-DEVS model. Thus, the modeler does not need to re-evaluate invariants. The information about violated invariants, in terms of activation events, is directly accessible in the state transition function, making upward activation even more explicit in ML-DEVS than it already is.

Such an introduction of activation events entails a corresponding adaptation of the abstract simulator of ML-DEVS or the demonstration of how to transform a ML-DEVS model with activation events into one without (as introduced in Section 9.2).

11.3.2 Model Specification

As Section 9.3 mentions, there exists a Java-based representation of the ML-DEVS model specification, which is implemented as a plug-in for the modeling and simulation framework

[4] In the tradition of other DEVS variants or DEVS-based modeling formalism.

JAMES II [Ewald, Himmelspach, Jeschke, Leye, & Uhrmacher 2010; Himmelspach & Uhrmacher 2009]. Hence, creating executable simulation models in ML-DEVS means to implement and extend specific interfaces and abstract classes in Java, respectively. This may be "easier" than using the set-theoretic notation described in Section 9.2, but still requires a certain skill set and some training.

In our opinion, there are two promising approaches that could ease the model creation and improve the overall modeling experience, particularly for novices in the DEVS realm:

- Enabling modelers to create models through a sophisticated graphical model editor (i. e., *visual modeling*).
- Incorporating more natural and intuitive modeling languages to create models that are eventually transformed into the actual modeling formalism; here ML-DEVS.

Over the last decades a lot of work was dedicated to enabling the graphical specification of models, using some sort of GUI. Some of this work directly focuses on DEVS-based modeling [Fard & Sarjoughian 2015; Ighoroje, Maïga, & Traoré 2012; Praehofer & Pree 1993; Risco-Martín, de la Cruz, Mittal, & Zeigler 2009; Sarjoughian & Elamvazhuthi 2009; Wainer 2002; Wainer & Liu 2009], whereas other work is more general [Buss 1996; Buss & Blais 2007; Rivera, Duran, & Vallecillo 2009]. Most of these approaches use a metaphor of components[5] that can be interconnected with each other through connections [6] to express the structure of a complex model This is rather straightforward and especially when defining the model structure at the level of coupled systems. However, the challenging part is enabling the modeler to specify the actual behavior of the individual model components by using suitable visual metaphors. Especially since we have to keep in mind, that DEVS and its variants only provide the theoretical frame for manipulating the state or state variables of a model, but do not provide specific instructions on how to do so. In fact, the modeler can incorporate arbitrary functions to change state variables within state transitions.

The second general approach to ease the model creation is to use a more intuitive modeling language on top of the set-theoretic model specification of ML-DEVS. For Cell-DEVS [Wainer, Frydman, & Giambiasi 1997], a DEVS-variant that allows spatial modeling, such a specialized, high-level modeling language exists [Ameghino, Troccoli, & Wainer 2001; Rodriguez & Wainer 1999] allowing the modeler to specify Cell-DEVS models more conveniently. Such a modeling language can be more general or specific to a certain domain. The latter leads us to *domain-specific modeling languages*, which are gaining more and more popularity in recent years. A domain-specific (modeling) languages make use of idioms and abstractions used and established in the corresponding domain, for a better understanding and easier modeling in the domain [Walter, Parreiras, & Staab 2014]. Examples of such domain-specific modeling languages are ML3 [Warnke et al. 2015] from computational demography and ML-Rules Maus et al. [2011] from systems biology.

However, both approaches, visual modeling and specialized modeling languages, need to be thoroughly evaluated before we can draw objective conclusions on their intuitiveness and convenience (see Section 11.1).

[5] depicted as rectangles
[6] depicted as connection lines

Part IV

Appendices

A Mathematical Notations and Concepts

As noted in Chapter 2, the subject of modeling are systems, which can be real or imaginary. "The general formal properties of systems, closed and open systems, etc., can be axiomatized in terms of set theory" [von Bertalanffy 1969, p. 21]. Therefore, "set theory provides the means to construct [modeling] formalisms" Zeigler [1984, p. 22] and, thus, modeling formalisms, such as the one presented in this thesis, are often defined by using concepts from set theory and algebra (*algebraic specification*). So in the words of Wymore [1967, p. 3]:

> Only if mathematical rigor is adhered to, can systems problems be dealt with effectively, and so it is that the system engineer must, at least, develop an appreciation for mathematical rigor if not also considerable mathematical competence.

For this reason, we describe and define, in the following, the most important mathematical concepts, especially those whose meanings may not always be clear or vary in the literature. Additionally, we introduce the notation that we use in the remainder of the thesis (particularly in the Chapters 8 and 9). For more details about fundamental mathematical concepts, e. g., sets, relations, or functions, refer to standard textbooks such as Devlin [1993] or Jech [1997].

A.1 Set- and Function-Theoretic Concepts

In the following section, we introduce set- and function-theoretic concepts that play a crucial role for defining DEVS-based modeling formalisms, especially those that are based on P-DEVS or defined at the level of structured system.

A.1.1 Supersets

In set theory, a *superset* is the antonym of a *subset*. According to Hrbacek and Jech [1999, p. 6], a set X is a subset of Y, denoted by $X \subseteq Y$, if and only if every element of X is an element of Y, i. e.,

$$\forall z\colon\ z \in X \Rightarrow z \in Y.$$

At the same time, Y is the superset of X, denoted by $Y \supseteq X$, meaning that Y contains at least all elements of X. If $X \neq Y$, X is a proper subset of Y and Y is a proper superset of X denoted by $X \subset Y$ and $Y \supset X$, respectively. Figure A.1 illustrates the relation between a subset and superset in terms of an Euler diagram.

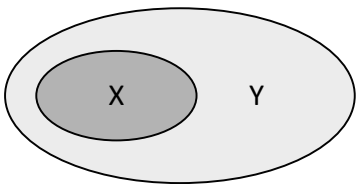

Figure A.1: An Euler diagram showing that X is a proper subset of Y, i. e., $X \subset Y$ and conversely Y is a proper superset of X, i. e., $Y \supset X$.

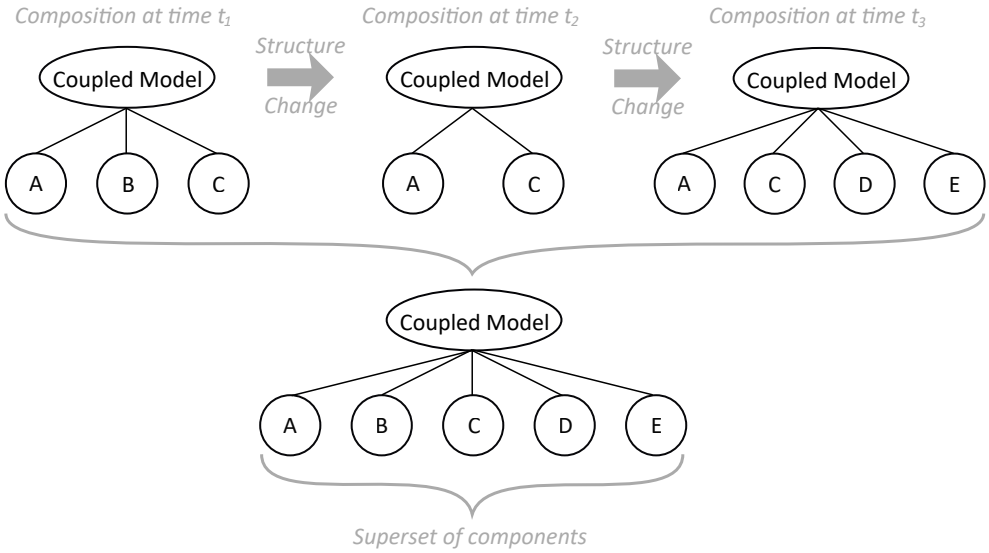

Figure A.2: Three different possible compositions of a coupled model with a variable composition at three different instants (top). The superset of all possible components for the three depicted compositions (bottom).

From a certain point of view, supersets are quite similar to *universal sets* (or *universes*), which are sets containing all elements under consideration[1]. These universal sets are the supersets of more specific sets that are of interest in a certain domain and which are often defined explicitly.

In the context of time-variant model structures, supersets prove to be useful. For instance, even if the composition of a composed model (or coupled model) changes during time, we often know the set of all possible components that may exist at some point beforehand. In such cases, we can define a superset of components regardless of their availability during simulation. The actual availability of the different components then needs to be determined for each possible situation, i. e., model state, by a dedicated function. Figure A.2 shows the relation between the availability of components and the superset of components.

Even if components shall be created more dynamically or in great numbers, there are often certain constraints that can be used for defining or refining a superset of components, e. g., by using intensional definition techniques as described in Section 3.3. For instance, eukaryotic cells do not contain other cells. However, the question is, how these constraints can be defined conveniently and concisely.

We can also use supersets when defining models with variable ports. Similar to a variable composition, we often know beforehand which ports can, in principle, become available. Hence, we can define the superset of all possible ports regardless their availability during simulation.

A.1.2 Families of Sets and Indexed Families of Sets

A set whose elements are sets themselves, i. e., a set of sets, is often called a *family of sets*. So, in set theory, a family is simply a set of sets, such as a *power set*. Given an arbitrary set A, the power set of A, denoted by $P(A)$ or 2^A, is the family of all subsets of A (including the empty set, denoted by $\varnothing$ or $\{\}$).

[1] A universal set does not need to contain all elements, which leads to *Russell's paradox* [Russell 1903, §106] implying that a universe cannot exist in naive set theory.

Example A.1.1 (Power Set)
Let A be the set $\{a, b\}$. The power set of A, 2^A, is defined by the following family of sets:

$$\big\{\{\,\}, \{a\}, \{b\}, \{a, b\}\big\}.$$

Based on this description, a family of sets is a *proper set*, in which each element occurs exactly once. However, sometimes families of sets are described as *collections* of sets (rather than sets of sets), as they may contain certain elements (also called members) more than once, which violates the notion of a set. This brings us to the idea of *indexed families of sets*. Adapting Sundstrom [2013, p. 268], we define an indexed family of sets as follows:

Definition A.1.1 (Indexed Family of Sets)
Let I be a nonempty set and suppose that for each $i \in I$ there exists a corresponding set A_i. The family of sets

$$\{A_i\}_{i \in I} \text{ or } \{A_i \mid i \in I\}$$

is called an **indexed family of sets** indexed by I. Each $i \in I$ is called **index** and I is the **indexing set** (or index set).

In contrast to the previous notion of a family of sets, an indexed family of sets does not exclude the possibility that for two different indexes i and j, with $i \neq j$, the corresponding sets A_i and A_j are equal. However, the following has to hold:

$$\forall i, j \in I\colon\ i = j \Rightarrow A_i = A_j.$$

Remark. The idea of attaching indices to elements of a set (which can be sets) allowing these elements to occur more than once is similar to the underlying idea of *disjoint unions* (cf. Appendix A.1.3), but different from the idea of *bags* (cf. Appendix A.1.4).

Since there is a mapping between indices and sets, i. e., each index of the indexing set is associated with a member of a regular family of sets, we consider an indexed set of families as a function that maps indices to corresponding sets (cf. Halmos [1960, p. 34]). This view leads to the following, alternative definition, which is based on Goodfriend [2005, pp. 58–9]:

Definition A.1.2 (Indexed Family of Sets as a Function)
Let I be a nonempty set and let $\mathcal{A}$ be a nonempty set of sets, i. e., a regular, non-indexed family of sets. An **indexed family of sets** indexed by the set I is a function A with

$$A\colon\ I \to \mathcal{A}.$$

This means that, for each index i in I, $A(i) \in \mathcal{A}$. For notational convenience, we also write A_i instead of $A(i)$. Similar to Definition A.1.1, we also write $\{A_i\}_{i \in I}$ to denote the indexed family of sets A indexed by the set I.

Definition A.1.2 allows us to map different indices to the same member of the family $\mathcal{A}$. However, an indexed family of sets can still describe an injective function such that:

$$\forall i, j \in I\colon\ i \neq j \Leftrightarrow A_i \neq A_j.$$

Remark. Some authors do not explicitly distinguish between families of sets and indexed families of sets and often refer to the latter when talking about families of sets. Herein, we distinguish between both concepts explicitly.

A.1.3 Disjoint Unions or Disjoint Sums

A *disjoint union* (or *disjoint sum*) is a binary operator, denoted by $\oplus$, that takes two arbitrary sets and combines all elements of both sets while retaining the original set membership as a distinguishing characteristic (usually some sort of index) of the resulting union set (cf. Uhrmacher [2001]).

Definition A.1.3 (Disjoint Union)
Let A and B be two arbitrary sets. The **disjoint union** of A and B, denoted by $A \oplus B$ is defined as follows:

$$A \oplus B \equiv (A \times \{1\}) \cup (B \times \{2\}) .$$

Instead of $A \oplus B$, we also find the notations $A \sqcup B$ or $A \cup^* B$ to denote the disjoint union of A and B in the literature. Here, we adhere to the $\oplus$-notation.

Please note that we can also choose other, possibly non-numerical, values (e. g., strings) as distinguishing characteristic, i. e., to discriminate between the memberships to the underlying sets.

Example A.1.2
Given the sets $A = \{a, b, c, d\}$ and $B = \{a, b, e, f\}$, the disjoint union of A and B is then, according to Definition A.1.3, defined as follows:

$$A \oplus B = \big\{(a,1), (b,1), (c,1), (d,1), (a,2), (b,2), (e,2), (f,2)\big\}.$$

Based on indexed families of arbitrary sets (see Section A.1.2), we can generalize a disjoint union to an n-ary operation.

Definition A.1.4 (Disjoint Union of an Indexed Family of Sets)
Let $\{A_i\}_{i \in I}$ be an indexed family of arbitrary sets indexed by the set I, then the **disjoint union of this indexed family**, denoted by $\bigoplus_{i \in I} A_i$, is defined by

$$\bigoplus_{i \in I} A_i = \big\{(a, i) \mid a \in A_i\big\}.$$

A.1.4 Bags and Bag Sets

A *bag* or *multiset* is a generalization of the concept of a set, in which elements can occur multiple, but finite times [Syropoulos 2001]. According to Syropoulos, a bag is defined as follows:

Definition A.1.5 (Bag)
A **bag** $\mathcal{A}$ over a set A is a pair

$$\langle A, f \rangle,$$

where A is the set of elements that can occur in the bag and $f\colon A \to \mathbb{N}_0$ is a function that indicates the multiplicity of the elements of A in the bag $\mathcal{A}$, i. e., the number of occurrences of the elements, where $\mathbb{N}_0$ denotes the set of natural numbers including 0.

Let $\mathcal{A} = \langle A, f \rangle$ be a bag; we will call the set A also the *base set* of the bag $\mathcal{A}$. A set $B \subseteq A$ is called the *support* of the bag $\mathcal{A}$, if $\forall a \in A\colon\ f(a) > 0 \Rightarrow a \in B$ and $\forall a \in A\colon\ f(a) = 0 \Rightarrow a \notin B$.

Notation A.1.1 (Bags). Like Syropoulos [2001], we also use the square bracket notation to write down bags and distinguish them from ordinary sets. For instance, we write

$\mathcal{A} = [a, a, a, b, c, c]$ for the bag $\mathcal{A} = \langle \{a, b, c\}, \{(a, 3), (b, 2), (c, 1)\} \rangle$, where the set $\{a, b, c\}$ is the support of $\mathcal{A}$. Please note that the above notation is ambiguous in a way that we can only derive the support of the respective bag but not the base set. The bag $\mathcal{A} = [a, a, a, b, c, c]$ could also represent the bag $\langle \{a, b, c, d\}, \{(a, 3), (b, 2), (c, 1), (d, 0)\} \rangle$, whose support is also the set $\{a, b, c\}$.

If the multiplicity of all elements of the base set A of bag $\mathcal{A}$ is 0, then $\mathcal{A}$ refers to the empty bag, denoted by $\varnothing$ or $[\,]$. By definition, the support of an empty bag is the empty set.

In addition to bags, Syropoulos [2001] defines a number of operations on bags. Such an operation is the sum of two bags, which we will use later.

Definition A.1.6 (Sum of Two Bags)
Let $\mathcal{A} = \langle A, f \rangle$ and $\mathcal{B} = \langle A, f \rangle$ be two bags with the same base set (A). The **sum** of $\mathcal{A}$ and $\mathcal{B}$, denoted by $\mathcal{A} \uplus \mathcal{B}$, is another bag

$$\mathcal{C} = \langle A, h \rangle,$$

where

$$\forall a \in A\colon\ h(a) = f(a) + g(a).$$

The sum operation has the following properties:

- Commutative, i. e., $\mathcal{A} \uplus \mathcal{B} = \mathcal{B} \uplus \mathcal{A}$;
- Associative, i. e., $(\mathcal{A} \uplus \mathcal{B}) \uplus \mathcal{C} = \mathcal{A} \uplus (\mathcal{B} \uplus \mathcal{C})$;
- Their exists a bag, the empty bag $\varnothing$, such that $\mathcal{A} \uplus \varnothing = \mathcal{A}$.

In addition to the sum of two bags, we define the sum of n bags with $n > 2$.

Definition A.1.7 (Sum of n Bags with $n > 2$)
Let $\mathcal{A}_1 = \langle A, f_1 \rangle$, $\mathcal{A}_2 = \langle A, f_2 \rangle$, …, $\mathcal{A}_n = \langle A, f_n \rangle$ be n bags, with $n \in \mathbb{N}$, that have the same base set A. The **sum of these bags**, denoted by $\biguplus_{i=1}^{n} \mathcal{A}_i$, is the bag

$$\mathcal{A} = \langle A, f \rangle,$$

where

$$\forall a \in A\colon\ f(a) = \sum_{i=1}^{n} f_i(a).$$

Let A be an arbitrary, nonempty set. We define A^b as a set of all possible bags (*bag set*)—including the empty bag—that have A as base set. Thus, A^b formally defines a proper set whose elements are bags.

Notation A.1.2. Instead of using calligraphic letters, such as $\mathcal{A}$, to denote elements of a bag set, we also write $a^b \in A^b$.

Note that the aforementioned bag set A^b is different from the set $\mathcal{P}^A$ defined by Syropoulos [2001] (which can be viewed as a power set of a bag), as the latter is restricted to bags whose support is the set A, i. e., $\mathcal{P}^A \subset A^b$.

Remark. Bag sets were introduced in the DEVS realm by the need to express the inputs of models in P-DEVS variants (cf. Chow and Zeigler [1994] or Zeigler et al. [2000, pp. 142–3]), which can contain the same input several times (see Section 4.2). However, the respective literature presents a rather informal definition of a bag set (set of bags). In addition, X^b is sometimes erroneously described as a bag over the elements in the set X and not as a set of bags.

A.1.5 Domains, Ranges, Co-Domains, and Images

Domains and *ranges* of relations and functions play a crucial role in the following chapters, especially in Chapters 8 and 9. Relations and their domains and ranges can be used to give a formal, set-theoretic definition of a function (cf. Devlin [1993, p. 12]).

Definition A.1.8 (Domain of a Binary Relation)
Let R be a binary relation with $R \subseteq A \times B$ and A and B being arbitrary sets. We define the **domain** of R, denoted by $dom(R)$, by

$$dom(R) = \{a \in A \mid \exists\, b \in B : (a, b) \in R\}.$$

The domain of a relation R is sometimes also denoted by $domain(R)$ or $Domain(f)$.

Similarly, we can define the range of a binary relation:

Definition A.1.9 (Range of a Binary Relation)
Let R be a binary relation with $R \subseteq A \times B$ and A and B being arbitrary sets. We define the **range** of R, denoted by $ran(R)$, by

$$ran(R) = \{b \in B \mid \exists\, a \in A : (a, b) \in R\}.$$

The range of a relation R is sometimes also denoted by $range(R)$ or $Range(R)$.

The above definitions of the domain and range of a binary relation can be generalized for $(n+1)$-ary relations with $n > 1$, as done by Devlin [1993, pp. 12–3].

Definition A.1.10 (Domain of a n-ary Relation with $n > 2$)
Let $R \subseteq \bigtimes_{i=1}^{n} A_i$ be an n-ary relation on the n-fold Cartesian product of the arbitrary sets $A_1, A_2, \ldots, A_n$ with $n > 2$ and $n \in \mathbb{N}$. The **domain** of R is defined by

$$dom(R) := \left\{ (a_1, a_2, \ldots, a_{n-1}) \in \bigtimes_{i=1}^{n-1} A_i \;\middle|\; \exists a_n \in A_n : (a_1, a_2, \ldots, a_{n-1}, a_n) \in R \right\}.$$

Hence, we can still think of an n-ary relation (with $n > 2$) such as above as a set of ordered pairs, where the first component is a $(n-1)$-tuple and the second component a single value, i. e.,

$$R = \{((a_1, a_2, \ldots, a_{n-1}), a_n) \mid \forall i \in [1, n] : a_i \in A_i\}.$$

Notation A.1.3 (Finite Cartesian Products). In case we are talking about a finite, n-fold Cartesian product of the sets $A_1, A_2, \ldots, A_n$ with $n \in \mathbb{N}$, such as in Definition A.1.10, we also write

$$\bigtimes_{i=1}^{n} A_i$$

instead of

$$A_1 \times A_2 \times \ldots \times A_n.$$

The range of an n-ary relation is defined correspondingly:

Definition A.1.11 (Range of a n-ary Relation with $n > 2$)
Let $R \subseteq \times_{i=1}^{n} A_i$ be an n-ary relation on the n-fold Cartesian product of the arbitrary sets A_1, A_2, ..., A_n with $n > 2$ and $n \in \mathbb{N}$. The **range** of R is defined by

$$ran(R) := \left\{ a_n \in A_n \;\middle|\; \exists (a_1, a_2, \ldots, a_{n-1}) \in \bigtimes_{i=1}^{n-1} A_i \colon\ (a_1, a_2, \ldots, a_{n-1}, a_n) \in R \right\}.$$

In addition to relations, functions are of particular interest in the remainder of the thesis. Functions are relations with certain characteristics, more specifically: "An n-ary function on a set x is an $(n+1)$-ary relation, R, on x [x is an $(n+1)$-fold Cartesian product] such that for every $a \in dom(R)$ there exits exactly one $b \in ran(R)$ such that $(a, b) \in R$" [Devlin 1993, p. 13]. Instead of $(a, b) \in R$, we write $R(a) = b$. We can define the domain and range of a function similarly as done above.

Definition A.1.12 (Domain of a Function)
Let f be a (total) function with

$$f \colon\ A \to B,$$

then the set A is the **domain of the function** f, denoted by $dom(f)$, i. e.,

$$dom(f) = A.$$

Note that the set A in the above definition could also refer to an n-fold Cartesian product, then the domain of the function would be this product. According to Khoussainov and Nerode [2001, p. 7], we define the range of a function as follows:

Definition A.1.13 (Range of a Function)
Let f be a (total) function with

$$f \colon\ A \to B.$$

The **range of the function** f, denoted by $ran(f)$, is a subset of the set B, i. e., $ran(f) \subseteq B$, that is defined by

$$ran(f) = \big\{ f(a) \mid a \in A \big\}.$$

So the range of function f in the above definition does not have to equal set B (however it can). We call the set B also the *co-domain* of the function f, which is often distinguished from the function's range.

Definition A.1.14 (Co-Domain of a Function)
Let f be a (total) function with

$$f \colon\ A \to B,$$

then the set B is the **co-domain of the function** f, denoted by $cdm(f)$, i. e.,

$$cdm(f) = B.$$

Please note that the range of a function sometimes refers the function's co-domain (i. e., $ran(f) = B$). Herein, we will use the range in the more common meaning of the *image* of a function (cf. Halmos [1960, p. 31]), which is another function-theoretic concept.

Definition A.1.15 (Image of a Function)
Let f be a (total) function with

$$f\colon A \to B.$$

If $U \subseteq A$, we define the **image** of U under f, denoted by $f[U]$, by

$$f[U] = \{f(a) \mid a \in U\}.$$

When we compare the definition of a function's range with that of its image, it becomes apparent that, for a function f with $f\colon A \to B$:

$$ran(f) = f[A],$$

thus the image of A under f is the range of f [Halmos 1960, p. 31].

A.1.6 Partial Functions

In a function, each argument, i. e., element of the function's domain, has to be mapped to a value of the function's co-domain (in other words: a function describes total mapping). Hence, we call such a function also a *total function*.

Partial functions generalizing the concept of total functions by relaxing the totality of their definition (i. e., a total function is defined for all elements of the function's domain). Based on Pierce [2002, p. 16], we define a partial function as follows:

Definition A.1.16 (Partial Function)
A **partial function** f from A to B is, denoted by

$$f\colon A \rightsquigarrow B,$$

is said to be defined on an argument $a \in dom(f)$ with $dom(f) \subseteq A$ and $f(a) \in B$, otherwise the function is undefined, denoted by

$$f(a') = \text{undef}$$

with $a' \in A \backslash dom(f)$.

So at a first glance, both kinds of functions seem to be similar, as they are both defined for all elements of their respective domains. However, the subtle difference between total and partial functions lies in their domains.

Definition A.1.17 (Domain of a Partial Function)
Let f be a partial function with

$$f\colon A \rightsquigarrow B.$$

The **domain of the partial function** f, denoted $dom(f)$, is some subset of the set A, i. e.,

$$dom(f) \subseteq A.$$

If $dom(f) = A$, then the partial function f is a total function.

The range and co-domain of a partial function are defined just like for total functions (see Section A.1.5).

Since a partial function is a total function regarding its domain, we can turn any partial function f with $f\colon A \rightsquigarrow B$ into a total function f' that is defined by

$$f'\colon A' \to B;\ f'(a) \mapsto f(a)$$

with $A' = dom(f)$. Alternatively, we can define the partial function f by the total, piecewise function

$$f''\colon A \to B^{\text{undef}}$$

with

$$f''(a) = \begin{cases} f(s) & \text{if } s \in dom(f) \\ \text{undef} & \text{otherwise,} \end{cases}$$

where $B^{\text{undef}} = B \cup \{\text{undef}\}$ and undef is a special element that is not in B. So all elements of A that are not in the domain of f are mapped to the special element undef. Note that instead of undef any other element that is not in S can be used.

A.1.7 Projections and Projection Functions

In general, a projection refers to a mapping of a set or structure into a subset or a substructure. Jech [2002, p. 34] writes that

> [...] a set B is a projection of a set A if there is a mapping of A onto B. Note that B is a projection of A if and only if there is a partition P of A such that $|P| = |B|$. If $|A| \geqslant |B| > 0$, then B is a projection of A.

Cartesian products are one of this structures on which we can define a projection.

Definition A.1.18 (Projection on Cartesian Product)
Let X be a Cartesian product with

$$X = X_1 \times X_2 \times \ldots \times X_n$$

with $n \in \mathbb{N}$, then we define the i-th **projection** on elements of X, denoted by $\pi_i(x)$ with $x \in X$, as an unary function, where

$$\forall (x_1, x_2, \ldots, x_n) \in X\ \forall i \in \{i \mid 0 < i \leqslant n\}\colon\ \pi_i((x_1, x_2, \ldots, x_n)) = x_i.$$

In a nutshell, the projection allows us to access elements of an ordered n-tuple individually. In the literature, we also find the notation $\pi_i(x)$ rather than $\pi_i(x)$.

A.2 Structuring Sets According to Variables

Often the states of systems of study are not opaque or flat, but can be described as a collection of different variables, i. e., state variables (cf. Law and Kelton [2000, p. 3]). Similarly, we may describe the inputs and outputs of a system as vectors of input variables and output variables, respectively. Zeigler et al. [2000, p. 123] call such systems *multivariable* or *structured systems*.

When creating models of such systems, especially *mechanistic models*, it is natural that we want to capture our additional knowledge about the systems' states, inputs, and outputs in the process. This leads us to *structured system specifications* [Zeigler 1976, pp. 247–55]. One possibility to model structured systems is to *structure* the states, inputs, and outputs of the corresponding models according to certain variables—state, input, and output variables.

A simple, straightforward approach is to define the sets of states, inputs, and outputs of a model as n-fold Cartesian products, where n corresponds to the number of the respective descriptive variables. For each of the three sets a certain number in the interval $[1, n]$ refers to a certain state, input, or output variable. An element of such an n-fold Cartesian product (i. e., state, input, or output) is an n-tuple, where the i-th projection of the tuple (with $i \in [1, n]$) returns a concrete value of a state, input, or output variable (see Appendix A.1.7). Hence, the sets upon the Cartesian products are defined correspond to ranges of values that can be assigned to the respective variables.

Example A.2.1 (State Set as Cartesian Product)
Suppose the state of a simple model shall consists of a physical phase, a real number that indicates the time elapsed since the last state transition (in seconds), and a natural number that represents some abstract information that the model can store. The state set S of such a model can now be defined as follows:

$$S = \underbrace{\{\text{“on”}, \text{“off”}\}}_{phase} \times \underbrace{\mathbb{R}_0^+}_{\sigma} \times \underbrace{\mathbb{N}_0}_{store},$$

where the variable names (phase, σ, or store) are just displayed for illustration purposes Then, the triple

$$(\text{“on”}, 2.5, 1)$$

represents the state when the model is in phase on*,* 2.5 *seconds have passed since the last state transition, and the model stores the information* 1*.*

Note that the state space described by S is infinite, since S contains all positive real numbers.

The above approach of structuring sets is pursued, e. g., by Zeigler et al. [2000, pp. 77–84] for defining the state sets of simple DEVS models.

A drawback of using regular Cartesian products is the implicit relation between variable names and values; variable names are not part of the definition. So if a modeler defining such Cartesian products does not provide information about the meaning of the sets in the Cartesian products (implicit knowledge), it may hinder other modelers in understanding the corresponding model.

In the following, we briefly describe two other approaches to structure sets according to certain variables while making the relation between variable names and values more explicit than in the approach above.

A.2.1 Multivariable Sets

Zeigler et al. [2000, pp. 123] introduce *multivariable sets* (or *structured sets*[2]) as a “system theoretic mechanism for representing the use of variables in modeling and simulation practice,” especially in the DEVS realm[3]. Multivariable sets can be used to model the states, inputs, and outputs of a system in a structured manner.

[2] Since we present other approaches to structure sets, we use the term “multivariable sets” rather than structured sets to refer to the approach introduced by Zeigler et al.

[3] Please note that the structured sets defined by Zeigler et al. [2000, pp. 124–5] look quite different from the structured sets as originally introduced in Zeigler [1976, pp. 248–51] and Zeigler [1984, pp. 40–1]. However, the more recent variant of structured sets has prevailed.

Definition A.2.1 (Multivariable Sets)
Let V be an ordered set of n variables (variable names) with

$$V = (v_1, v_2, \ldots, v_n)$$

and let

$$S_1, S_2, \ldots, S_n$$

be n arbitrary sets. A **multivariable set** (or structured set) of the ordered set V and the sets $S_1, S_2, \ldots, S_n$ is defined as the pair

$$S = (V, S_1 \times S_2 \times \ldots \times S_n).$$

Each coordinate $i \in [1, n]$ is denoted by the variable $v_1 \in V$, i. e., for an element $s \in S$ with $s = (s_1, s_2, \ldots, s_n)$ the value of v_i equals s_i [Zeigler et al. 2000, p. 124][a].

According to Zeigler et al. the multivariable set S can also be written as follows:

$$S = \big\{(v_1, v_2, \ldots, v_n) \mid v_1 \in S_1, v_2 \in S_2, \ldots, v_n \in S_n\big\}.$$

[a] Note that from a set-theoretical point of view, $s \in S$ is not defined since S is an ordered pair, not a set.

The latter notation is mathematically more sound and corresponds to simple n-fold Cartesian products, because

$$\big\{(v_1, v_2, \ldots, v_n) \mid v_1 \in S_1, v_2 \in S_2, \ldots, v_n \in S_n\big\} \equiv S_1 \times S_2 \times \ldots \times S_n.$$

However, it is important to note that in this simplified notation the tuple $(v_1, v_2, \ldots, v_n)$ does not correspond to the actual ordered set of variables V.

Remark. The type of ordering on the set of variables V is not further defined by Zeigler et al. [2000]. However, as the variables are given as a *sequence* (or *chain*), we assume a *linear* or *total ordering* on V. For instance, we can define V as the total ordered set $(V^*, \leqslant)$, where

$$V^* = \{v_1, v_2, \ldots, v_3\}$$

is a proper set that contains all elements of V and $\leqslant \subseteq V^* \times V^*$ is a total ordering (binary relation) that orders the elements of V^* as they occur in V:

$$\begin{aligned} \leqslant = \big\{&(v_1, v_1),\ (v_1, v_2),\ \ldots,\ (v_1, v_n),\\ &(v_2, v_2),\ \ldots,\ (v_2, v_n),\\ &\qquad\vdots\\ &(v_n, v_n)\big\} \end{aligned}$$

Given a multivariable set $S = (V, S_1 \times S_2 \times \ldots \times S_n)$, S_i denotes the *range* of v_i, i. e., the values that can be assigned to the variable v_i. We also write $range_{v_i}(S) = S_i$. To make use of multivariable sets, Zeigler et al. [2000, pp. 124–5] define a few operations on these sets. The following operations are of particular interest for this thesis:

- The operation *variables* returns the ordered set of variables of a given multivariable set. So given a multivariable set S as defined in Definition A.2.1, $variables(S) = (v_1, v_2, \ldots, v_2)$ or $variables(S) = V$. Note that this operation formally does not return a proper set but an ordered n-tuple.

- The (cross) product of two multivariable sets is, again, a multivariable set with all the coordinates ordered in a sequence. Let $A = \{(a_1, \ldots, a_n) \,|\, a_1 \in A_1, \ldots, a_n \in A_n\}$ and $B = \{(b_1, \ldots, b_m) \,|\, b_1 \in B_1 \ldots b_m \in B_m\}$ be two multivariable sets and let $\mathit{variables}(A) = V_A$ and $\mathit{variables}(B) = V_B$. The product of A and B is then defined by

$$A \times B = \big\{(a_1, \ldots, a_n, b_1, \ldots b_m) \,|\, a_1 \in A_1, \ldots, a_n \in A_n, b_1 \in B_1, \ldots, b_m \in B_m\big\}.$$

  If we specify V_A and V_B as the totally ordered sets $V_A = (V_A^*, \leqslant_A)$ and $V_B = (V_B^*, \leqslant_B)$, respectively, and assume that $V_A^* \cap V_B^* = \varnothing$, then the product of A and B boils down to the sum of the totally ordered sets V_A and V_B, denoted by $V_A + V_B$. According to Khoussainov and Nerode [2001, p. 16], the sum of the two totally (linearly) ordered sets V_A and V_B can be obtained as follows:

  – The set of all elements of $V_A + V_B$ is $V_A^* \cup V_B^*$;

  – The order $\leqslant$ on the sum is defined as the union:

$$\leqslant_A \cup \leqslant_B \cup \big\{(a, b) \mid a \in V_A^*, b \in V_B^*\big\}.$$

- Given a multivariable set S as defined in Definition A.2.1, the projection operation $\cdot : S \times V \to \bigcup_{i=1}^{n} S_i$ allows accessing a given coordinate—referred to by a variable (name)—of an element of the multivariable set S with

$$s \cdot v_i = s_i,$$

  where $s \in S$, $v_i \in V$, and $s_i \in S_i$.

Notation A.2.1. For the projections, we will also write $s.v_i$ instead of $s \cdot v_i$ to keep consistency with the notation used in this thesis.

For more details on multivariable sets refer to Zeigler et al. [2000][pp. 124–5].

A.2.2 Generalized Cartesian Products

As the previous section already indicates, a multivariable set, as defined by Zeigler et al. can be mapped to a traditional n-fold or *finite Cartesian product* (also called cross product) of n sets $S_1 \times S_2 \times \ldots \times S_n$, where (i) n equals the number of variables that are used to structure the multivariable set and (ii) for each n, the set S_n refers to the range of values that can be assigned to the n-th variable. Thereby, the order of the variables dictates the order of the corresponding sets in the Cartesian product and the relation between variables and value ranges is established only implicitly, via the coordinates in the ordered set of variables and the Cartesian product. *Generalized Cartesian products*, on the other hand, provide a more sophisticated mean to structure sets while establishing an explicit, unambiguously relation between variable names and values.

Remark. In the literature, different names exists that refer to the concept described below. Other terms are arbitrary Cartesian product, infinite (Cartesian) product [Jech 1997, pp. 43–6], indexed Cartesian product [Pirotte 1982], or Cartesian product of an indexed family of sets [Devlin 1993, p. 15]. Although the term generalized Cartesian product as used by Pirotte [1982] or Borzyszkowski, Kubiak, Leszczylowski, and Sokolowski [1988] is not established, we stick to the term in this thesis.

A generalized Cartesian product is "general definition of a Cartesian product of an arbitrary (possibly infinite) family of sets" [Devlin 1993, p. 15], i. e., a generalized Cartesian product is a generalization of a traditional n-fold Cartesian product.

Definition A.2.2 (Generalized Cartesian Product)
Let $\{X_i\}_{i\in I}$ be an indexed family of sets (indexed by the set I) and let I be a nonempty index set. The Cartesian product of the indexed family $\{X_i\}_{i\in I}$, called **generalized Cartesian product** and denoted by

$$\prod_{i\in I} X_i,$$

is defined as follows:

$$\prod_{i\in I} X_i = \left\{ f \;\middle|\; \left(f\colon\ I \to \bigcup_{i\in I} X_i \right) \wedge \left(\forall i \in I\colon\ f(i) \in X_i \right) \right\}.$$

Hence, a generalized Cartesian product is a set of functions and each element of the product is a (total) function that maps an index i to an element of the family member (i. e., set) that is indexed by i.

In the case that the index set I is finite, the above generalized Cartesian products offers a quite different definition of a traditional n-fold Cartesian product, which is however closely related to the original definition [Devlin 1993, p. 15]

When we think of the index set I as a set of variables (or variable names), like the set V of multivariable sets, the elements of a generalized Cartesian product can be considered as variable assignments, where each variable a value is assigned to. However, in contrast to the set V, the index set I is a proper set. In other words, using generalized Cartesian products instead of multivariable sets allows us to get rid of the need to implicitly or explicitly define an order on the variables. The members of the underlying indexed family of sets can be considered as value ranges of the associated variables.

Let, in the following, f be an element of the generalized Cartesian product $\prod_{i\in I} X_i$. For each index $i \in I$ we get its assigned value by applying f to i, i. e., $f(i)$. By Definition A.2.2, the domain of f equals the index set I:

$$dom(f) = I,$$

and the co-domain of f is the union of the underlying indexed family of sets, i. e.,

$$ran(f) = \bigcup_{i\in I} X_i.$$

Notation A.2.2. For notational convenience, we also write an element of a generalized Cartesian product $\prod_{i\in I} X_i$ as a set of ordered pairs:

$$\left\{ (i, f(i)) \;\middle|\; i \in I \right\}.$$

For each $i \in I$ the projection function (i-th projection), denoted by $\pi_i(f)$,

$$\pi_i\colon \prod_{j\in I} X_j \to X_i$$

is defined by

$$\pi_i(f) = f(i).$$

Like Zeigler et al. [2000, p. 124], we define two auxiliary functions on generalized Cartesian products: *variables*(...) and $range_i((\ldots))$. The function *variables* returns the index set of an arbitrary generalized Cartesian product $\prod_{i\in I} X_i$, i. e.,

$$variables\left(\prod_{i\in I} X_i\right) = I.$$

Let $i \in I$, where I is the index set of the arbitrary generalized Cartesian product $\prod_{j \in I} X_j$, then the function $range_i$ returns the set X_i (the value range of i), i. e.,

$$range_i \left(\prod\nolimits_{j \in I} X_j \right) = X_i.$$

Moreover, we define the cross product of two generalized Cartesian products $\prod_{i \in I} X_i$ and $\prod_{j \in J} Y_j$ with $I \cap J = \varnothing$ as follows:

$$\prod_{i \in I} X_i \times \prod_{j \in J} Y_j = \prod_{k \in K} Z_k$$

with

$$K = I \cup J$$

and

$$\{Z_k\}_{k \in K} = \{X_i\}_{i \in I} \cup \{Y_j\}_{j \in J}.$$

A.2.3 Partial Cartesian Products

The in- and outputs of models with ports and a static structure (incl. the interface) can readily be specified by using generalized Cartesian products (as defined in the previous section). Each index of the product corresponds to a port (name) and each element of the products corresponds to a value assignment for each port. However, since one focus of the thesis is on models whose interfaces are variable, we have to consider the consequent variability of ports when specifying the in- and outputs of such models. As generalized Cartesian products define total functions, i. e., each port has to map to a value from its value range, we need a further generalization of these Cartesian products that allow us to capture the variability of ports more explicitly[4]. In the case of variable ports, we actually want to specify partial functions instead of total functions, in which only a subset of all potential ports is assigned to values (namely the currently available ports).

For this reason, we introduce *partial Cartesian products.*

Definition A.2.3 (Partial Cartesian Product)
Let $\{X_i\}_{i \in I}$ be an indexed family of sets (indexed by the set I) and let I be a nonempty index set. The **partial Cartesian product** of the indexed family $\{X_i\}_{i \in I}$, denoted by

$$\widetilde{\prod_{i \in I}} X_i,$$

is defined as follows:

$$\widetilde{\prod_{i \in I}} X_i = \bigcup_{I' \subseteq I} \left\{ f \;\middle|\; \left(f : I \rightsquigarrow \bigcup_{j \in I} X_j \right) \wedge \left(dom(f) = I' \right) \wedge \left(\forall j \in I' : \; f(j) \in X_j \right) \right\}.$$

Hence, each element of a partial Cartesian product is a partial function that maps a subset I' of the index set I (incl. I) to elements of the corresponding family members, i. e., the sets indexed by I'.

Let f be a partial function from the partial Cartesian product $\widetilde{\prod}_{i \in I} X_i$ then we define, similarly to a generalized Cartesian product, the projection $\pi_j(f)$ by

$$\pi_j(f) = f(j).$$

[4] Without the necessity of introducing special elements—not part of any value range—to which a port will map in case it shall not be available.

B Finite State Automata

Finite state automata are closely related to the modeling formalism DEVS and its variants, as discussed in Section 4.2. In simple terms, a "finite automaton has a [finite] set of states, and its control moves from state to state in response to external inputs" [Hopcroft et al. 2001].

B.1 Basic Automata

Definition B.1.1 (Deterministic Finite Automaton)
A **deterministic finite automaton** is a5-tuple

$$A = (Q, \Sigma, \delta, q_0, F),$$

where:

1. A is the *name* of the automaton.
2. Q is a *finite set of states*.
3. Σ is a *finite set of input symbols*, i. e., the *input alphabet*.
4. $\delta : Q \times \Sigma \to Q$ is the *(state) transition function*.
5. q_0 is the start (or initial) state with $q_0 \in Q$.
6. F is a set of final or accepting states with F being a subset of Q.

In the literature, variations of the above notation can be found (e. g., Khoussainov and Nerode [2001]). The state transition function δ is often given as a graph or transition table, illustrating the transition between states.

In a deterministic finite automaton (DFA), each state has exactly one subsequent state, except final states (they have no subsequent states). A nondeterministic finite automaton (NFA) loosens this restriction. Khoussainov and Nerode [2001, p. 48] give the following definition of a nondeterministic finite automaton:

Definition B.1.2 (Nondeterministic Finite Automaton)
A **nondeterministic finite automaton** over the alphabet Σ is a quadruple

$$\mathcal{A} = (S, I, T, F),$$

where

1. S is a finite nonempty set called the **set of states**.
2. I is a subset of S called the **set of initial states**.
3. $T \subset S \times \Sigma \times S$ is a nonempty set called the **transition table** or **transition digram**.q
4. F is a subset of S called the *set of final states*.

Often the input alphabet Σ becomes part of the actual definition of the automaton, i. e., $\mathcal{A} = (\Sigma, S, I, T, F)$.

By using the powerset or *powerset* or *subset construction algorithm*, we can show that, despite different definition of NFA and DFA, for any NFA a DFA can be constructed that

recognizes the same formal language (cf. J. C. Martin [2010, p. 108]). In other words, NFA are unabble to recognize a language that cannot be recognized by some DFA.

B.2 Moore Machine

A *Moore machine* is a finite automaton, i. e., a model of a sequential machine, whose output "at a given time depends only on the current state of the machine" [Moore 1956]. More concretely, a Moore machine is a certain kind of a deterministic transducer, as does not allow random elements, the transition of the state is defined by a function, and the machine has an output. The first idea of such a machine was proposed by Edward F. Moore, after which the machine is named, in 1956 Moore [1956]. From the rather informal characterization of Moore we can derive the following formal definition of a Moore machine:

Definition B.2.1 (Moore Machine)
A **Moore machine** $\mathcal{A}_{Moore}$ is defined by the 6-tuple

$$\mathcal{A}_{Moore} = (Q, \Sigma, \Omega, q_0, F, \delta, \lambda),$$

where:

1. Q is a finite nonempty *set of states*.
2. Σ is the *input alphabet*.
3. Ω is the *output alphabet*.
4. $q_0 \in Q$ is the *initial state*.
5. F is a subset of Q called the *set of final states*.
6. $\delta : \ Q \times \Sigma \to Q$ is the state transition function.
7. $\lambda : \ Q \to \Omega$ is the output function.

Please note that the notation follows the one of general systems as used by Zeigler et al. [2000].

C Abstract Simulator of Parallel DEVS

In the tradition of DEVS, the execution semantics of P-DEVS is formally described by means of an abstract simulator. The abstract simulator consists of processors that can be of the following three types: (i) Simulator, (ii) Coordinator, and (iii) Root-Coordinator.

C.1 Simulator

Algorithm C.2 shows the Simulator of P-DEVS that is responsible for executing an atomic P-DEVS model.

Algorithm C.1: The Simulator of P-DEVS, which adapts the one presented by Zeigler et al. [2000, p. 285].

```
variables:
    m                   // the atomic model associated with the Simulator
    s                   // the atomic model's current state
    tl                  // time of the last event
    tn                  // time of the next internal event

1   // initialization
2   when receive i-message (i, t) with time t then
3       s ← s_init
4       tl ← t
5       tn ← tl + m.ta(s)
6       send done-message (d, tn) to parent
7
8   when receive *-message (*, t) with time t then
9       if tn = t then
10          y^b ← m.λ(s)
11          send y-message (y^b, t) to parent
12
13  // update the state
14  when receive x-message (x^b, t) with input bag x^b then
15      if x^b = ∅ and tn = t then
16          // internal state transition
17          s ← m.δ_int(s)
18      else if x^b ≠ ∅ and tn = t then
19          // confluent state transition
20          s ← m.δ_con(s, x^b)
21      else if x^b ≠ ∅ and tl ⩽ t < tn
22          // external state transition
23          e ← t − tl
24          s ← δ_ext((s, e), x^b)
25      else
26          error: illegal state
27
28     // update times
29     tl ← t
30     tn ← tl + m.ta(s)
31     send done-message (d, tn) to parent
```

C.2 Coordinator

Algorithm C.2 shows the Coordinator of P-DEVS that is responsible for executing coupled P-DEVS models. For each coupled model one Coordinator is instantiated.

C.3 Root-Coordinator

The Root Coordinator of P-DEVS is shown in Algorithm C.3. In contrast to a Simulator or Coordinator, the Root Coordinator is not associated with a model. Instead, the Root Coordinator is in control of the general simulation loop and keeps track of the time elapsed in the simulation.

Algorithm C.2: The Coordinator of P-DEVS, which adapts the one presented by Zeigler et al. [2000, pp. 285–7] and neglects checks for synchronization problems.

```
variables:
    n                   // the coupled model associated with the Coordinator
    tl                  // time of the last event
    tn                  // time of the next internal event
    parent              // parent processor
    events              // queue of elements (d, tn_d) sorted by tn_d (ascending order)
    msg                 // message container for outputs from components

1   // initialization
2   when receive i-message (t) with time t then
3       // initialize all components
4       send i-message (t) to each d ∈ n.D
5           wait for done-message (tn_d) from each d ∈ D
6               enqueue (d, tn_d) in events
7       // update times
8       tl ← t
9       tn ← min_tn(events)
10      send done-message (tn) to parent
11
12  // collect outputs from all imminents
13  when receive *-message (t) with time t then
14      msg, y^b ← ∅
15      IMM ← {d ∈ D | (d, t) ∈ events}
16      send *-message (t) to each d ∈ IMM
17          wait for y-message (y^b_d, t_d) from each d ∈ D
18              // remove internal event from event queue
19              dequeue (d, t) from events
20              // buffer output bag of current child d
21              add (d, y^b_d) to msg
22      // create y-message for parent and send it
23      for each (d, y^b_d) ∈ msg with d ∈ n.I_n do
24          for each y ∈ y^b_d with n.Z_{d,n}(y) ≠ ∅ do
25              add n.Z_{d,n}(y) to y^b
26      send y-message (y^b, t) to parent
27
28  // forward external events and execute the imminent and influenced children
29  when receive x-message (x^b, t) with input bag x^b and time t
30      INF ← {r ∈ D | ∃(d, y^b_d) ∈ msg ∃y ∈ y^b_d: d ∈ n.I_r ∧ y ≠ ∅}
31      INF ← INF ∪ {r ∈ D | n ∈ n.I_r ∧ ∃x ∈ x^b: n.Z_{n,d}(x) ≠ ∅}
32      for each r ∈ INF ∪ IMM do
33          // determine potential inputs
34          x^b_r ← ∅
35          for each (d, y^b_d) ∈ msg with d ∈ n.I_r do
36              for each y ∈ y^b_d with n.Z_{d,r}(y) ≠ ∅
37                  add n.Z_{d,r}(y) to x^b_r
38          if n ∈ n.I_r then
39              for each x ∈ x^b with n.Z_{n,r}(x) ≠ ∅ do
40                  add n.Z_{n,r}(x) to x^b_r
41          send x-message (x^b_r, t) to r
42              wait for done-message (tn_r) from r
43                  // update event queue
44                  enqueue (r, tn) in events
45      // update times
46      tl ← t
47      tn ← min_tn(events)
48      send done-message (tn) to parent
```

Algorithm C.3: The Root-Coordinator of P-DEVS.

```
variables:
    child                // subordinate processor
    t_0                  // simulation start time
    t                    // current simulation time
    tn_c                 // time of the next internal event of the child processor

1   // initialization
2   t ← t_0
3   send i-message (i, t) to child
4      wait until done-message (d, tn_c) received from child
5         t ← tn_c       // update simulation time (jump to first internal event)
6
7   // simulation loop
8   repeat
9      send *-message (*, t) to child
10        wait until y-message (y^b, t) received from child
11     send x-message (∅, t) to child
12        wait until done-message (d, tn_c) received from child
13           t ← tn_c    // update simulation time (jump to next internal event)
14  until end of simulation
```

Acronyms

ADP Adenosine diphosphate

ATP Adenosine triphosphate

CBD Component-Based Development

CBSE Component-Based Software Engineering

CODES COmposable Discrete-Event scalable Simulation

COMO Component-based Modeling or Component Models

COP Component-Oriented Programming

CoSMoS Component-based System Modeler and Simulator

DES Discrete Event Simulation

DEVS Discrete Event System Specification

DFA Deterministic finite automaton

DNA Deoxyribonucleic acid

FSM Finite State Machine

HLA High-Level Architecture

JAMES II Java-based *Multipurpose Environment for Simulation*; before that *Java-based Agent Modeling Environment for Simulation*

ML-DEVS Multi-Level Discrete Event System Specification

NFA Nondeterministic finite automaton

OCL Object Constraint Language

PDES Parallel Discrete Event Simulation

P-DEVS Parallel Discrete Event System Specification

QSS Quantized State System

SES System Entity Structure

SysML Systems Modeling Language

UML Unified Modeling Language

VHDL Very High Speed Integrated Circuit Hardware Description Language

XML Extensible Markup Language

References

ACIMS. (2009). *DEVSJAVA.* Retrieved 2005-07-20, from `https://acims.asu.edu/software/devsjava/`

Alur, R., & Dill, D. L. (1994, April). A Theory of Timed Automata. *Theoretical Computer Science*, *126*(2), 183–235. doi: 10.1016/0304-3975(94)90010-8

Ameghino, J., Troccoli, A., & Wainer, G. A. (2001). Models of Complex Physical Systems Using Cell-DEVS. In *Proceedings of the 34th Annual Simulation Symposium* (pp. 266–273). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc.

Bae, J. W., Bae, S. W., Moon, I.-C., & Kim, T. G. (2016, February). Efficient Flattening Algorithm for Hierarchical and Dynamic Structure Discrete Event Models. *ACM Transactions on Modeling and Computer Simulation*, *26*(4), 1–25. doi: 10.1145/2875356

Baeten, J. C. M. (2005, May). A Brief History of Process Algebra. *Theoretical Computer Science*, *335*(2-3), 131–146. doi: 10.1016/j.tcs.2004.07.036

Balci, O. (1997, December). Verification, Validation and Accreditation of Simulation Models. In S. Andradóttir, K. J. Healy, D. H. Withers, & B. L. Nelson (Eds.), *Proceedings of the 1997 Winter Simulation Conference* (pp. 135–141). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1145/268437.268462

Banks, J. (1998, September). Principles of Simulation. In J. Banks (Ed.), *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice* (1st ed., pp. 3–30). Hoboken, NJ, USA: John Wiley & Sons, Inc.

Banks, J., Carson II, J. S., Nelson, B. L., & Nicol, D. M. (2000). *Discrete-Event System Simulation* (3rd ed.). Upper Saddle River, NJ, USA: Prentice Hall.

Barros, F. J. (1995a, December). Dynamic Structure Discrete Event System Specification: A New Formalism for Dynamic Structure Modeling and Simulation. In C. Alexopoulos, K. Kang, W. R. Lilegdon, & D. Goldsman (Eds.), *Proceedings of the 1995 Winter Simulation Conference* (pp. 781–785). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1145/224401.224731

Barros, F. J. (1995b). *Dynamic Structure Discrete Event System Specification: Formalism and Abstract Simulators* (Technical Report). Coimbra, Portugal: University of Coimbra.

Barros, F. J. (1996, March). The Dynamic Structure Discrete Event System Specification Formalism. *Transactions of the Society for Computer Simulation International*, *13*(1), 35–46.

Barros, F. J. (1997, October). Modeling Formalisms for Dynamic Structure Systems. *ACM Transactions on Modeling and Computer Simulation*, *7*(4), 501–515. doi: 10.1145/268403.268423

Barros, F. J. (1998, December). Abstract Simulators for the DSDE Formalism. In D. J. Medeiros, E. F. Watson, J. S. Carson, & M. S. Manivannan (Eds.), *Proceedings of the 1998 Winter Simulation Conference* (pp. 407–412). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.1998.745015

Barros, F. J. (2002). Modeling and Simulation of Dynamic Structure Heterogeneous Flow Systems. *SIMULATION*, *78*(1), 18–27. doi: 10.1177/003754970207800119

Barros, F. J. (2003). Dynamic Structure Multiparadigm Modeling and Simulation. *ACM Transactions on Modeling and Computer Simulation*, *13*(3), 259–275. doi: 10.1145/937332.937335

Barros, F. J. (2004). Describing the HLA Using the DFSS Formalism. In T. G. Kim (Ed.), *Artificial Intelligence and Simulation, 13th International Conference on AI, Simulation,*

and Planning in High Autonomy Systems, AIS 2004, Jeju Island, Korea, October 4-6, 2004, Revised Selected Papers (Vol. 3397, pp. 117–127). Berlin, Germany: Springer-Verlag Berlin. doi: 10.1007/978-3-540-30583-5_13

Barros, F. J. (2012, December). A Compositional Approach for Modeling and Simulation of Bio-molecular Systems. In C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, & A. M. Uhrmacher (Eds.), *Proceedings of the 2012 Winter Simulation Conference.* Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. (Article No. 235) doi: 10.1109/WSC.2012.6465260

Barros, F. J. (2014, April). On the Representation of Dynamic Topologies: The Case for Centralized and Modular Approaches. In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative (DEVS '14).* San Diego, CA, USA: Society for Computer Simulation International. (Article No. 40)

Barros, F. J., Mendes, M. T., & Zeigler, B. P. (1994). Variable DEVS - Variable Structure Modeling Formalism: An Adaptive Computer Architecture Application. In *Proceedings of the 5th Annual Conference on AI, and Planning in High Autonomy Systems (AIS 1994)* (pp. 185–191). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/AIHAS.1994.390474

Barton, J. J., & Vijayaraghavan, V. (2002). *UBIWISE, A Ubiquitous Wireless Infrastructure Simulation Environment* (Tech. Rep.). Palo Alto, CA, USA: HP Laboratories. Retrieved from `http://www.hpl.hp.com/techreports/2002/HPL-2002-303.pdf`

Bergero, F., & Kofman, E. (2014, May). A Vectorial DEVS Extension for Large Scale System Modeling and Parallel Simulation. *SIMULATION*, *90*(5), 522–546. doi: 10.1177/0037549714529833

Bisgambiglia, P.-A., de Gentili, E., & Santucci, J.-F. (2009, August). iDEVS: New Method to Study Inaccurate Systems. In *Proceedings of the 2009 IEEE International Conference on Fuzzy Systems* (pp. 300–307). Piscataway, NJ, USA: IEEE. doi: 10.1109/FUZZY.2009.5277046

Bittig, A. T., Haack, F., Maus, C., & Uhrmacher, A. M. (2011). Adapting Rule-Based Model Descriptions for Simulating in Continuous and Hybrid Space. In F. Fages (Ed.), *Proceedings of the 9th International Conference on Computational Methods in Systems Biology* (pp. 161–170). New York, NY, USA: ACM. doi: 10.1145/2037509.2037533

Bittig, A. T., Matschegewski, C., Nebe, J. B., Stählke, S., & Uhrmacher, A. M. (2014). Membrane Related Dynamics and the Formation of Actin in Cells Growing on Micro-topographies: A Spatial Computational Model. *BMC Systems Biology*, *8*(106), 19. doi: 10.1186/s12918-014-0106-2

Borzyszkowski, A. M., Kubiak, R., Leszczylowski, J., & Sokolowski, S. (1988). *Towards a Set-theoretic Type Theory* (Tech. Rep.). Gdańsk, Poland: Polish Academy of Sciences.

Brim, L., Černá, I., Vařeková, P., & Zimmerova, B. (2005). Component-interaction Automata as a Verification-oriented Component-based System Specification. In *Proceedings of the 2005 Conference on Specification and Verification of Component-Based Systems - SAVCBS '05.* New York, New York, USA: ACM Press. (Article No. 4) doi: 10.1145/1123058.1123063

Brooks Jr., F. P. (1987, April). No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, *20*(4), 10–19. doi: 10.1109/MC.1987.1663532

Buss, A. H. (1996, December). Modeling with Event Graphs. In J. M. Charnes, D. J. Morrice, D. T. Brunner, & J. J. Swain (Eds.), *Proceedings of the 1996 Winter Simulation Conference* (pp. 153–160). Washington, DC, USA: IEEE Computer Society. doi: 10.1145/256562.256597

Buss, A. H., & Blais, C. (2007, December). Composability and Component-based Discrete Event Simulation. In S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, & J. J. Barton (Eds.), *Proceedings of the 2007 Winter Simulation Conference* (pp.

694–702). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2007.4419663

Bylund, M., & Espinoza, F. (2001). Using Quake III Arena to Simulate Sensors and Actuators when Evaluating and Testing Mobile Services. *CHI 2001 Extended Abstracts on Human Factors in Computing Systems*, 241–242. doi: 10.1145/634067.634210

Bylund, M., & Espinoza, F. (2002). Testing and Demonstrating Context-aware Services with Quake III Arena. *Communications of the ACM*, *45*(1), 46–48. doi: http://doi.acm.org/10.1145/502269.502294

Campbell, D. T. (1974). Downward Causation in Hierarchically Organised Biological Systems. In F. J. Ayala & T. G. Dobzhansky (Eds.), *Studies in the philosophy of biology: Reduction and related problems* (pp. 179–186). Berkeley: University of California Press.

Campuzano, F., Garcia-Valverde, T., Garcia-Sola, A., & Botia, J. A. (2011, April). Flexible Simulation of Ubiquitous Computing Environments. In P. Novais, D. Preuveneers, & J. M. Corchado (Eds.), *Ambient Intelligence - Software and Applications: 2nd International Symposium on Ambient Intelligence (ISAmI 2011)* (pp. 189–196). Berlin, Germany: Springer-Verlag Berlin Heidelberg. doi: 10.1007/978-3-642-19937-0_24

Cassandras, C. G., & Lafortune, S. (2008). *Introduction to Discrete Event Systems* (2nd ed.). Springer-Verlag Berlin.

Castro, R., Kofman, E., & Wainer, G. (2008, April). A Formal Framework for Stochastic DEVS Modeling and Simulation. In H. Rajaei (Ed.), *Proceedings of the 2008 Spring Simulation Multiconference* (pp. 421–428). San Diego, CA, USA: Society for Computer Simulation International.

Castro, R., Kofman, E., & Wainer, G. A. (2010, June). A Formal Framework for Stochastic Discrete Event System Specification Modeling and Simulation. *SIMULATION*, *86*(10), 587–611. doi: 10.1177/0037549709104482

Cellier, F. E. (1991). *Continuous System Modeling*. New York, NY, USA: Springer-Verlag New York, Inc. doi: 10.1007/978-1-4757-3922-0

Cellier, F. E., & Kofman, E. (2006). *Continuous System Simulation*. Springer.

Chen, G., & Szymanski, B. K. (2002, December). COST: A Component-oriented Discrete Event Simulator. In E. Yücesan, C.-H. Chen, J. L. Snowdon, & J. M. Charnes (Eds.), *Proceedings of the 2002 Winter Simulation Conference* (pp. 776–782). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2002.1172960

Chow, A. C. (1996). Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism and Its Distributed Simulator. *Transactions of the Society for Computer Simulation International*, *13*(2), 55–67.

Chow, A. C., & Zeigler, B. P. (1994, December). Parallel DEVS: A Parallel, Hierarchical, Modular Modeling Formalism. In J. D. Tew, M. S., D. A. Sadowski, & A. Seila (Eds.), *Proceedings of the 1994 Winter Simulation Conference* (pp. 716–722). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers. doi: 10.1109/WSC.1994.717419

Chow, A. C., Zeigler, B. P., & Kim, D. H. (1994, December). Abstract Simulator for the Parallel DEVS Formalism. In *Proceedings of the 5th Annual Conference on AI, and Planning in High Autonomy Systems (AIS 1994)* (pp. 157–163). IEEE Computer Society Press. doi: 10.1109/AIHAS.1994.390488

Cook, D. J., & Das, S. K. (Eds.). (2004). *Smart Environments: Technology, Protocols, and Applications* (1st ed.). Hoboken, NJ, USA: John Wiley & Sons, Inc.

Copeland, B. J. (2015). The Church-Turing Thesis. In E. N. Zalta (Ed.), *The stanford encyclopedia of philosophy* (2015th ed.).

Copi, I. M., Cohen, C., & McMahon, K. (2014). *Introduction to Logic* (14th ed.). Harlow, ESS, UK: Pearson Education Limited.

Dalle, O. (2007). The OSA Project: An Example of Component Based Software Engineering

Techniques Applied to Simulation. In *Proceedings of the 2007 Summer Computer Simulation Conference* (pp. 1155–1162). San Diego, CA, USA: Society for Computer Simulation International. doi: 10.1145/1357910.1358090

Dalle, O., Zeigler, B. P., & Wainer, G. A. (2008, December). Extending DEVS to Support Multiple Occurrence in Component-based Simulation. In S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, & J. W. Fowler (Eds.), *Proceedings of the 2008 Winter Simulation Conference* (pp. 933–941). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2008.4736159

Davis, P. C., Fishwick, P. A., Overstreet, C. M., & Pedgen, C. D. (2000, December). Model Composability as a Research Investment: Responses to the Featured Paper. In J. A. Joines, R. R. Barton, K. Kang, & P. A. Fishwick (Eds.), *Proceedings of the 2000 Winter Simulation Conference* (pp. 1585–1591). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2000.899143

Davis, P. K., & Anderson, R. H. (2004, April). Improving the Composability of DoD Models and Simulations. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, *1*(1), 5–17. doi: 10.1177/154851290400100101

Davis, P. K., & Tolk, A. (2007, December). Observations on New Developments in Composability and Multi-resolution Modeling. In S. G. Henderson, B. Biller, M.-H. Hsieh, J. F. Shortle, J. D. Tew, & J. J. Barton (Eds.), *Proceedings of the 2007 Winter Simulation Conference* (pp. 859–870). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2007.4419682

de Alfaro, L., & Henzinger, T. A. (2001). Interface Theories for Component-based Design. In T. A. Henzinger & C. M. Kirsch (Eds.), *Embedded Software. EMSOFT 2001* (pp. 148–165). Springer, Berlin, Heidelberg. doi: 10.1007/3-540-45449-7_11

Deniz, F. (2010). *Variable Structure and Dynamism Extensions to a DEVS Based Modeling and Simulation Framework* (Unpublished master's thesis). Middle East Technical University.

Deniz, F., Alpdemir, M. N., Kara, A., & Oğuztüzün, H. (2012, June). Supporting Dynamic Simulations with Simulation Modeling Architecture (SiMA): A Discrete Event System Specification-based Modeling and Simulation Framework. *SIMULATION*, *88*(6), 707–730. doi: 10.1177/0037549711428233

Deniz, F., Kara, A., Alpdemir, M. N., & Oğuztüzün, H. (2009, July). Variable Structure and Dynamism Extensions to SiMA, A DEVS Based Modeling and Simulation Framework. In O. Balci, X. Hu, M. Sierhuis, & L. Yilmaz (Eds.), *Proceedings of the 2009 Summer Computer Simulation Conference* (pp. 117–124). Vista, CA, USA: Society for Modeling \& Simulation International. Retrieved from `http://www.proceedings.com/12872.html`

Department of Defense. (1998). *DoD Modeling and Simulation (M&S) Glossary (DoD 5000.59-M).* Washington, DC, USA: Retrieved from http://www.dtic.mil/whs/directives/corres/pdf/500059m.pdf.

de Roever, W.-P., Langmaack, H., & Pnueli, A. (Eds.). (1998). *Compositionality: The Significant Difference* (Vol. 1536). Berlin, Heidelberg, Germany: Springer Berlin Heidelberg. doi: 10.1007/3-540-49213-5

Devlin, K. J. (1993). *The Joy of Sets: Fundamentals of Contemporary Set Theory* (2nd ed.; J. H. Ewing, F. W. Gehring, & P. R. Halmos, Eds.). New York, NY, USA: Springer-Verlag.

Dijkstra, E. W. (1982). *Selected Writings on Computing: A Personal Perspective* (D. Gries, Ed.). New York, NY, USA: Springer-Verlag.

Djitog, I., Aliyu, H. O., & Traoré, M. K. (2017, July). Multi-perspective Modeling of Healthcare Systems. *International Journal of Privacy and Health Information Management*, *5*(2), 1–20. doi: 10.4018/IJPHIM.2017070101

Elmqvist, H. (1978). *A Structured Model Language for Large Continuous Systems* (Doctoral dissertation, Lund University, Lund, Sweden). Retrieved from `http://lup.lub.lu.se/`

record/8524888

Elmqvist, H., Mattsson, S. E., & Otter, M. (2001). Object-oriented and Hybrid Modeling in Modelica. *Journal Européen des Systèmes Automatisés*, *35*(1), 1–10.

Ewald, R., Himmelspach, J., Jeschke, M., Leye, S., & Uhrmacher, A. M. (2010). Flexible Experimentation in the Modeling and Simulation Framework JAMES II—Implications for Computational Systems Biology. *Briefings in Bioinformatics*, *11*(3), 290–300. doi: https://doi.org/10.1093/bib/bbp067

Fard, M. D., & Sarjoughian, H. S. (2015). Visual and Persistence Behavior Modeling for DEVS in CoSMoS. In *DEVS '15 Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium* (pp. 227–234). San Diego, CA, USA: Society for Computer Simulation International.

Farmer, W. M. (2007). Chiron: A Multi-paradigm Logic. *Studies in Logic, Grammar and Rhetoric: The Journal of University of Bialystok*, *10*(23), 1–19.

Feng, T. H., Zia, M., & Vangheluwe, H. L. M. (2007, July). Multi-formalism Modelling and Model Transformation for the Design of Reactive Systems. In G. A. Wainer (Ed.), *Proceedings of the 2007 Summer Computer Simulation Conference (SCSC'07)* (pp. 505–5012). San Diego, CA, USA: Society for Computer Simulation International.

Figl, K., Mendling, J., & Strembeck, M. (2009). Towards a Usability Assessment of Process Modeling Languages. In *8th GI-Workshop Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (EPK), CEUR-WS* (pp. 138–156). Berlin, Germany: CEUR-WS.

Fujimoto, R. M. (2000). *Parallel and Distributed Simulation Systems* (1st ed.). New York, NY, USA: John Wiley & Sons, Inc.

Gaines, B. R. (1979). General Systems Research: Quo Vadis? *General Systems: Yearbook of the Society for General Systems Research*, *24*, 1–9.

Gallagher, R., & Appenzeller, T. (1999, April). Beyond Reductionism. *Science*, *284*(5411), 79. doi: 10.1126/science.284.5411.79

Gao, J., Li, Y., Wang, Y. G., & Chen, G. (2012, November). Micro-macro Modeling for Systems Biology with MR-DEVS. *Applied Mechanics and Materials*, *220-223*, 2975–2982. doi: 10.4028/www.scientific.net/AMM.220-223.2975

Garredu, S., Vittori, E., Santucci, J.-F., & Bisgambiglia, P.-A. (2013, July). From State-transition Models to DEVS Models - Improving DEVS External Interoperability using MetaDEVS - A MDE Approach. In T. Ören, J. Kacprzyk, L. Leifsson, M. S. Obaidat, & S. Koziel (Eds.), *Proceedings of the 3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2013)* (pp. 186–196). SciTePress - Science and and Technology Publications. doi: 10.5220/0004494401860196

Giambiasi, N., & Carmona, J. C. (2006, January). Generalized Discrete Event Abstraction of Continuous Systems: GDEVS Formalism. *Simulation Modelling Practice and Theory*, *14*(1), 47–70. doi: 10.1016/j.simpat.2005.02.009

Giambiasi, N., Escude, B., & Ghosh, S. (2001). GDEVS: A Generalized Discrete Event Specification for Accurate Modeling of Dynamic Systems. In *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems* (pp. 464–469). Piscataway, NJ, USA: The Institute of Electrical and Electronics Engineer, Inc. doi: 10.1109/ISADS.2001.917452

Goodfriend, J. H. (2005). *A Gateway to Higher Mathematics*. Sudbury, MA, USA: Jones and Bartlett Publishers.

Hagendorf, O., Pawletta, T., & Deatcu, C. (2009, September). Extended Dynamic Structure DEVS. In R. Aguilar, A. Bruzzone, & M. Piera (Eds.), *Proceedings of the 21st European Modeling and Simulation Symposium* (Vol. 1, pp. 36–45).

Halmos, P. R. (1960). *Naive Set Theory*. Princeton, NJ, USA: Van Nostrand Reinhold Company.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer*

Programming, *8*(3), 231–274. doi: 10.1016/0167-6423(87)90035-9

Heider, T., & Kirste, T. (2005). Multimodal Appliance Cooperation based on Explicit Goals: Concepts & Potentials. In *Proceedings of the 2005 Joint Conference on Smart Objects and Ambient Intelligence Innovative Context-Aware Services: Usages and Technologies (sOc-EUSAI '05)* (pp. 271–276). New York, NY, USA: ACM Press. doi: 10.1145/1107548.1107614

Hein, A., Burghardt, C., Giersich, M., & Kirste, T. (2009, September). Model-based Inference Techniques for Detecting High-level Team Intentions. In B. Gottfried & H. Aghajan (Eds.), *Behaviour monitoring and interpretation - bmi* (1st ed., pp. 257–288). Amsterdam, The Netherlands: IOS Press. doi: 10.3233/978-1-60750-048-3-257

Heineman, G. T., & Councill, W. T. (Eds.). (2001). *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley.

Helal, S., Lee, J. W., Hossain, S., Kim, E., Hagras, H., & Cook, D. J. (2011). Persim - Simulator for Human Activities in Pervasive Spaces. In *Proceedings of the Seventh International Conference on Intelligent Environments* (pp. 192–199). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/IE.2011.34

Helms, T., Ewald, R., Rybacki, S., & Uhrmacher, A. M. (2013). A Generic Adaptive Simulation Algorithm for Component-based Simulation Systems. *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation - SIGSIM-PADS '13*, 11–22. doi: 10.1145/2486092.2486095

Henzinger, T. A., Jobstmann, B., & Wolf, V. (2009, September). Formalisms for Specifying Markovian Population Models. In O. Bournez & I. Potapov (Eds.), *Reachability problems* (pp. 3–23). Springer-Verlag. doi: 10.1007/978-3-642-04420-5_2

Himmelspach, J. (2007). *Konzeption, Realisierung und Verwendung eines allgemeinen Modellierungs-, Simulations- und Experimentiersystems - Entwicklung und Evaluation effizienter Simulationsalgorithmen* (Dissertation). University of Rostock, Göttingen, Germany.

Himmelspach, J. (2012, December). Tutorial on Building M&S Software Based on Reuse. In C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, & A. M. Uhrmacher (Eds.), *Proceedings of the 2012 Winter Simulation Conference*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. (Article No. 167) doi: 10.1109/WSC.2012.6465306

Himmelspach, J., & Röhl, M. (2009, June). JAMES II - Experiences and Interpretation. In A. M. Uhrmacher & D. Weyns (Eds.), *Multi-agent systems: Simulation and application* (1st ed., pp. 509–533). Boca Raton, FL, USA: CRC Press.

Himmelspach, J., Röhl, M., & Uhrmacher, A. M. (2010, May). Component-based Models and Simulations for Supporting Valid Multi-agent System Simulations. *Applied Artificial Intelligence*, *24*(5), 414–442. doi: 10.1080/08839514.2010.481492

Himmelspach, J., & Uhrmacher, A. M. (2006). Sequential Processing of PDEVS Models. In A. G. Bruzzone, A. Guasch, M. A. Piera, & J. Rozenblit (Eds.), *Proceedings of the 3rd European Modeling & Simulation Symposium (EMSS)* (pp. 239–244). Barcelona, Spain: LogiSim. Retrieved from `http://wwwmosi.informatik.uni-rostock.de/mosi/veroeffentlichungen/inproceedingsreference.2006-06-27.0695819468`

Himmelspach, J., & Uhrmacher, A. M. (2007). Plug'n Simulate. In *Proceedings of the 40th Annual Simulation Symposium* (pp. 137–143). Washington, DC, USA: IEEE Computer Society. doi: 10.1109/ANSS.2007.34

Himmelspach, J., & Uhrmacher, A. M. (2009, October). The JAMES II Framework for Modeling and Simulation. In *2009 International Workshop on High Performance Computational Systems Biology* (pp. 101–102). IEEE. doi: 10.1109/HiBi.2009.20

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice Hall.

Hoekstra, A. G., Kroc, J., & Sloot, P. M. A. (2010). Introduction to Modeling of Complex Systems Using Cellular Automata. In A. G. Hoekstra, J. Kroc, & P. M. A. Sloot (Eds.), *Simulating Complex Systems by Cellular Automata* (pp. 1–16). Springer-Verlag Berlin Heidelberg. doi: 10.1007/978-3-642-12203-3

Hollmann, D. A., Cristiá, M., & Frydman, C. (2015, September). CML-DEVS: A Specification Language for DEVS Conceptual Models. *Simulation Modelling Practice and Theory*, *57*, 100–117. doi: 10.1016/j.simpat.2015.06.007

Hong, J. S., Song, H.-S., Kim, T. G., & Park, K. H. (1997, October). A Real-time Discrete Event System Specification Formalism for Seamless Real-time Software Development. *Discrete Event Dynamic Systems: Theory and Applications*, *7*(4), 355–375. doi: 10.1023/A:1008262409521

Honig, H. J., & Seck, M. D. (2012). ΦDEVS: Phase Based Discrete Event Modeling. In G. A. Wainer & P. J. Mosterman (Eds.), *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium.* San Diego, CA, USA: Society for Computer Simulation International. (Article No.: 39)

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Addison-Wesley.

Hrbacek, K., & Jech, T. (1999). *Introduction to Set Theory: Third Edition, Revised and Expanded* (3rd ed.; E. J. Taft & Z. Nashed, Eds.). Marcel Dekker, Inc.

Hu, X., Zeigler, B. P., & Mittal, S. (2005, February). Variable Structure in DEVS Component-based Modeling and Simulation. *SIMULATION*, *81*(2), 91–102. doi: 10.1177/0037549705052227

Huebscher, M. C., & McCann, J. A. (2004, August). Simulation Model for Self-adaptive Applications in Pervasive Computing. In *Proceedings of the 15th International Workshop on Database and Expert Systems Applications (SAACS 04)* (pp. 694–698). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/DEXA.2004.1333555

Huhns, M., & Singh, M. (1998, September). Cognitive Agents. *IEEE Internet Computing*, *2*(6), 87–89. doi: 10.1109/4236.735992

Hurley, P. J. (2006). *A Concise Introduction to Logic* (9th ed.). Belmont, CA, USA: Wadsworth.

Huttenlocher, P. R., & Dabholkar, A. S. (1997, October). Regional Differences in Synaptogenesis in Human Cerebral Cortex. *The Journal of Comparative Neurology*, *387*(2), 167–178. doi: 10.1002/(SICI)1096-9861(19971020)387:2⟨167::AID-CNE1⟩3.0.CO;2-Z

Hwang, M. H., & Zeigler, B. P. (2009, July). Reachability Graph of Finite and Deterministic DEVS Networks. *IEEE Transactions on Automation Science and Engineering*, *6*(3), 468–478. doi: 10.1109/TASE.2009.2021352

IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)–Framework and Rules. (2010). *IEEE Std 1516–2010*, i–22. doi: 10.1109/IEEESTD.2000.92296

IEEE Standard Glossary of Modeling and Simulation Terminology. (1989). *IEEE Std 610.3-1989*, 19. doi: 10.1109/IEEESTD.1989.94599

Ighoroje, U. B., Maïga, O., & Traoré, M. K. (2012). The DEVS-driven Modeling Language: Syntax and Semantics Definition by Meta-modeling and Graph Transformation. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium.* Retrieved from `https://dl.acm.org/citation.cfm?id=2346665` (Article No. 49)

Jacob, F., & Monod, J. (1961, June). Genetic Regulatory Mechanisms in the Synthesis of Proteins. *Journal of Molecular Biology*, *3*(3), 318–356. doi: 10.1016/S0022-2836(61)80072-7

Jamshidi, M. (2008). Introduction to System of Systems. In M. Jamshidi (Ed.), *System of systems engineering: Innovations for the twenty-first century* (2008th ed., pp. 1–20).

Hoboken, NJ, USA: John Wiley & Sons, Inc.

Jech, T. (1997). *Set Theory* (2nd ed.). Berlin, Germany: Springer-Verlag Berlin Heidelberg.

Jech, T. (2002). *Set Theory: The Third Millennium Edition, revised and expanded* (3rd ed.). Berlin, Germany: Springer.

Jifeng, H., Li, X., & Liu, Z. (2005). Component-based Software Engineering: The Need to Link Methods and Their Theories. In D. V. Hung & M. Wirsing (Eds.), *Theoretical Aspects of Computing (ICTAC)* (Vol. 3722, pp. 70–95). Springer. doi: 10.1007/11560647_5

John, M., Lhoussaine, C., & Niehren, J. (2009, September). Dynamic Compartments in the Imperative pi-Calculus. In P. Degano & R. Gorrieri (Eds.), *Computational methods in systems biology* (pp. 235–250). Berlin, Heidelberg, Germany: Springer-Verlag Berlin Heidelberg. doi: 10.1007/978-3-642-03845-7_16

Karnopp, D. C., Margolis, D. L., & Rosenberg, R. C. (2012). *System Dynamics: Modeling, Simulation, and Control of Mechatronic Systems* (5th ed.). Hoboken, NJ, USA: John Wiley & Sons, Inc.

Kasputis, S., & Ng, H. C. (2000, December). Composable Simulations. In J. A. Joines, J. J. Barton, K. Kang, & P. A. Fishwick (Eds.), *Proceedings of the 2000 Winter Simulation Conference* (Vol. 2, pp. 1577–1584). Los Alamitos, CA, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2000.899142

Kaye, P., Laflamme, R., & Mosca, M. (2007). *An Introduction to Quantum Computing.* Oxford, UK: Oxford University Press.

Khoussainov, B., & Nerode, A. (2001). *Automata Theory and its Applications* (2001st ed.). Boston, MA, USA: Birkhäuser.

Kirste, T. (2006, August). Smart Environments. In E. Aarts & J. L. Encarnação (Eds.), *True Visions: The Emergence of Ambient Intelligence* (2006th ed., pp. 321–337). Berlin, Germany: Springer-Verlag Berlin Heidelberg. doi: 10.1007/978-3-540-28974-6_17

Kleene, S. C. (1967). *Mathematical Logic* (2002nd ed.). Mineola, NY, USA: Dover Publications Inc.

Kofman, E., & Junco, S. (2001, September). Quantized-state Systems: A DEVS Approach for Continuous System Simulation. *Transactions of the Society for Computer Simulation International*, *18*(3), 123–132.

Korn, G. A., & Wait, J. V. (1978). *Digital Continuous-system Simulation.* Prentice Hall.

Krüger, F., Steiniger, A., Bader, S., & Kirste, T. (2012, September). Evaluating the Robustness of Activity Recognition Using Computational Causal Behavior Models. In A. K. Dey, H.-H. Chu, & G. Hayes (Eds.), *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp '12)* (pp. 1066–1074). New York, New York, USA: ACM Press. doi: 10.1145/2370216.2370443

Kwon, Y. W., Park, H. C., Jung, S. H., & Kim, T. G. (1996, March). Fuzzy-DEVS Formalism: Concepts, Realization and Applications. In *Proceedings of the 1996 Conference on AI, Simulation and Planning in High Autonomy Systems (AIS 1996)* (pp. 227–234).

Lau, K.-K., & Ntalamagkas, I. (2009, August). Component-based Construction of Concurrent Systems with Active Components. In *Proceedings of the 35th EUROMICRO Conference on Software Engineering and Advanced Applications* (pp. 497–501). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/SEAA.2009.45

Law, A. M., & Kelton, W. D. (2000). *Simulation Modeling and Analysis* (3rd ed.). McGraw-Hill Higher Education.

Leemis, L. M., & Park, S. K. (2006). *Discrete-Event Simulation: A First Course.* Upper Saddle River, NJ, USA: Pearson Prentice Hall.

The Levels of Conceptual Interoperability Model. (2003). In *Proceedings of the 2003 Fall Simulation Interoperability Workshop.*

Leye, S. (2013). *Toward Guiding Simulation Experiments* (Dissertation, University of Rostock, Rostock, Germany). doi: 10.18453/rosdok_id00001365

Li, Y., Li, B. H., Hu, X., & Chai, X. (2011, March). Formalization of Multi-resolution Modeling based on Dynamic Structure DEVS. In *2011 International Conference on Information Science and Technology (ICIST 2011)* (pp. 855–864). Institute of Electrical and Electronics Engineers. doi: 10.1109/ICIST.2011.5765113

Lin, J. T., & Lee, C.-C. (1993, June). A Three-phase Discrete Event Simulation with EPNSim Graphs. *SIMULATION*, *60*(6), 382–392. doi: 10.1177/003754979306000603

Livny, M. (1983). *The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems* (Dissertation). Weizmann Institute of Science, Rehovot, Isreal.

Ljung, L., & Glad, T. (1994). *Modeling of Dynamic Systems*. Englewood Cliffs, NJ, USA: Prentice-Hall, Inc.

Martin, J. C. (2010). *Introduction to Languages and the Theory of Computation* (4th ed.). London, UK: McGraw-Hill.

Martin, M., & Nurmi, P. (2006, July). A Generic Large Scale Simulator for Ubiquitous Computing. *2006 Third Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services*, 1–3. doi: 10.1109/MOBIQ.2006.340388

Mattsson, S. E., & Elmqvist, H. (1998, April). An Overview of the Modeling Language Modelica. In *Proceedings of the 3rd EUROSIM Congress (Eurosim'98)* (pp. 1–5). Retrieved from `http://user.asc.tuwien.ac.at/eurosim/index.php?id=44`

Maus, C. (2008). Component-based Modelling of RNA Structure Folding. In M. Heiner & A. M. Uhrmacher (Eds.), *Computational methods in systems biology* (Vol. 5307, pp. 44–62). Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg. doi: 10.1007/978-3-540-88562-7

Maus, C. (2012). *Toward Accessible Multilevel Modeling in Systems Biology* (Dissertation). University of Rostock.

Maus, C., Rybacki, S., & Uhrmacher, A. M. (2011). Rule-based Multi-level Modeling of Cell Biological Systems. *BMC Systems Biology*, *5*(166). doi: 10.1186/1752-0509-5-166

McGlinn, K., O'Neill, E., Gibney, A., O'Sullivan, D., & Lewis, D. (2010). SimCon: A Tool to Support Rapid Evaluation of Smart Building Application Design Using Context Simulation and Virtual Reality. *Journal of Universal Computer Science*, *16*(15), 1992–2018. doi: 10.3217/jucs-016-15-1992

Miller, J. G. (1978). *Living Systems* (1st ed.). McGraw-Hill.

Milner, R. (1982). *A Calculus of Communicating Systems* (1980th ed., Vol. 92). Berlin, Germany: Springer-Verlag. doi: 10.1007/3-540-10235-3

Milner, R. (1992, June). Functions as Processes. *Mathematical Structures in Computer Science*, *2*(02), 119–141. doi: 10.1017/S0960129500001407

Milner, R. (1999). *Communicating and Mobile Systems: The Pi-Calculus* (1st ed.). Cambridge, UK: Cambridge University Press.

Minsky, M. L. (1965). Models, Minds, Machines. In W. A. Kalenich (Ed.), *Proceedings of IFIP Congress* (pp. 45–49). Spartan Books.

Mittal, S. (2013, March). Emergence in Stigmergic and Complex Adaptive Systems: A Formal Discrete Event Systems Perspective. *Cognitive Systems Research*, *21*, 22–39. doi: 10.1016/j.cogsys.2012.06.003

Modelica Association. (2012, May). *Modelica - A Unified Object-oriented Language for System Modeling* (Language Specification). Modelica Association. Retrieved from `http://www.modelica.org`

Molter, H. G. (2012). *SynDEVS Co-Design Flow: A Hardware / Software Co-Design Flow Based on the Discrete Event System Specification Model of Computation*. Wiesbaden, Germany: Springer Vieweg. doi: 10.1007/978-3-658-00397-5

Molter, H. G., Seffrin, A., & Huss, S. A. (2009). DEVS2VHDL: Automatic Tansformation of XML-specified DEVS Model of Computation into Synthesizable VHDL Code. In *Forum on Specification & Design Languages* (pp. 1–6). Piscataway, NJ, USA: The Institute of

Electrical and Electronics Engineer, Inc.

MontiCore: A Framework for Compositional Development of Domain Specific Languages. (2010, September). *International Journal on Software Tools for Technology Transfer (SITT)*, *12*(6), 353–372. doi: 10.1007/s10009-010-0142-1

Moore, E. F. (1956, April). Gedanken-experiments on Sequential Machines. In C. E. Shannon & J. McCarthy (Eds.), *Automata studies* (pp. 129–153). Princeton University Press.

Morris, R. L., & Hollenbeck, P. J. (1995, December). Axonal Transport of Mitochondria along Microtubules and F-actin in Living Vertebrate Neurons. *The Journal of Cell Biology*, *131*(5), 1315–1326.

Morris, W. T. (1967, August). On the Art of Modeling. *Management Science*, *13*(12), B-707–B-717. doi: 10.1287/mnsc.13.12.B707

Muzy, A., & Zeigler, B. P. (2014, September). Specification of Dynamic Structure Discrete Event Systems Using Single Point Encapsulated Control Functions. *International Journal of Modeling, Simulation, and Scientific Computing*, *5*(3), 1450012–20. doi: 10.1142/S1793962314500123

Nance, R. E. (1994). The Conical Methodology and the Evolution of Simulation Model Development. *Annals of Operations Research*, *53*(1), 1–45. doi: 10.1007/BF02136825

Nishikawa, H., Yamamoto, S., Tamai, M., Nishigaki, K., Kitani, T., Shibata, N., ... Ito, M. (2006). UbiREAL: Realistic Smartspace Simulator for Systematic Testing. In P. Dourish & A. Friday (Eds.), *Ubicomp 2006: Ubiquitous computing* (Vol. 4206, pp. 459–476). Berlin, Heidelberg, Germany: Springer Berlin Heidelberg. doi: 10.1007/11853565

Nixon, P. A., Lacey, G., & Dobson, S. (Eds.). (2000). *Managing Interactions in Smart Environments: 1st International Workshop on Managing Interactions in Smart Environments (MANSE'99), Dublin, December 1999*. London, UK: Springer-Verlag London. doi: 10.1007/978-1-4471-0743-9_1

Nixon, P. A., Wagealla, W., English, C., & Terzis, S. (2004, November). Security, Privacy and Trust Issues in Smart Environments. In D. J. Cook & S. K. Das (Eds.), *Smart environments: Technology, protocols, and applications* (1st ed., pp. 249–270). Hoboken, NJ, USA: John Wiley & Sons, Inc.

Noble, J., Silverman, E., Bijak, J., Rossiter, S., Evandrou, M., Bullock, S., ... Falkingham, J. (2012, December). Linked Lives: The Utility of an Agent-based Approach to Modeling Partnership and Household Formation in the Context of Social Care. In C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, & A. M. Uhrmacher (Eds.), *Proceedings of the 2012 Winter Simulation Conference* (pp. 1–12). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2012.6465264

Nyolt, M., Steiniger, A., Bader, S., & Kirste, T. (2013, August). Describing and Evaluating Assistance using APDL. In Q. Bai, T. Ito, M. Zhang, F. Ren, & X. Tang (Eds.), *Proceedings of the International SSMCS Workshop* (pp. 38–49).

Nyolt, M., Steiniger, A., Bader, S., & Kirste, T. (2015). Describing and Evaluating Assistance Using APDL. In Q. Bai, F. Ren, M. Zhang, T. Ito, & X. Tang (Eds.), *Smart modeling and simulation for complex systems: Practice and theory* (Vol. 564, pp. 59–81). Tokyo, Osaka, Japan: Springer Japan. doi: 10.1007/978-4-431-55209-3_5

Odell, J. J., Van Dyke Parunak, H., Fleischer, M., & Brueckner, S. (2003). Modeling Agents and Their Environment. In F. Giunchiglia, J. Odell, & G. Weiß (Eds.), *Agent-Oriented Software Engineering III: Third International Workshop (AOSE 2002)* (pp. 16–31). Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg. Retrieved from `http://dl.acm.org/citation.cfm?id=1754726.1754729`

Ören, T. I. (1975, June). Simulation of Time-varying Systems. In J. Rose (Ed.), *Advances in cybernetics and systems* (pp. 1229–1238). London, UK: Gordon & Breach Science Publishers.

Ören, T. I., & Zeigler, B. P. (1986, January). From Stone Tools to Cognizant Tools: The Quest

Continues. In G. C. Vansteenkiste, E. J. H. Herkhoffs, L. Dekker, & J. C. Zuidervaart (Eds.), *Proceedings of the 2nd European Simulation Congress* (pp. 801–807). Leiden, The Netherlands: Brill Academic Pub.

Otter, M., & Elmqvist, H. (2000). Modelica - Language, Libraries, Tools, Workshop and EU-Project. *Simulation News Europe (SNE)*(29/30), 3–8.

Otter, M., Erik Mattsson, S., & Elmqvist, H. (2007). Multidomain Modeling with Modelica. In P. A. Fishwick (Ed.), *Handbook of dynamic system modeling* (pp. 36-1–36-27). Boca Raton, FL, USA: Chapman & Hall/CRC. doi: 10.1201/9781420010855.pt5

Palikaras, K., Lionaki, E., & Tavernarakis, N. (2015, September). Balancing Mitochondrial Biogenesis and Mitophagy to Maintain Energy Metabolism Homeostasis. *Cell Death and Differentiation*, *22*(9), 1399–1401. doi: 10.1038/cdd.2015.86

Park, J., Lee, J., & Choi, C. (2011, August). Mitochondrial Network Determines Intracellular ROS Dynamics and Sensitivity to Oxidative Stress through Switching Inter-mitochondrial Messengers. *PloS ONE*, *6*(8), e23211. doi: 10.1371/journal.pone.0023211

Park, J., Moon, M., Hwang, S., & Yeom, K. (2007, August). CASS: A Context-aware Simulation System for Smart Home. In H.-K. Kim, J. Tanaka, B. Malloy, R. Lee, C. Wu, & D.-K. Baik (Eds.), *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)* (pp. 461–467). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/SERA.2007.60

Patel, P. K., Shirihai, O., & Huang, K. C. (2013, January). Optimal Dynamics for Quality Control in Spatially Distributed Mitochondrial Networks. *PLoS Computational Biology*, *9*(7), e1003108. doi: 10.1371/journal.pcbi.1003108

Pawletta, T., Lampe, B. P., Pawletta, S., & Drewelow, W. (1996, September). A New Approach for Simulation of Variable Structure Systems. In Z. Vukić (Ed.), *Proceedings of the 41th Annual Conference KoREMA '96* (Vol. 4, pp. 83–87). Zagreb, Croatia: KoREMA. doi: 10.1.1.54.1321

Peckham, S. D., Hutton, E. W., & Norris, B. (2013, April). A Component-based Approach to Integrated Modeling in the Geosciences: The Design of CSDMS. *Computers & Geosciences*, *53*, 3–12. doi: 10.1016/j.cageo.2012.04.002

Peng, D., Ewald, R., & Uhrmacher, A. M. (2014, May). Towards semantic model composition via experiments. In J. D. A. Hamilton, Jr., G. F. Riley, & R. M. Fujimoto (Eds.), *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS' 14)* (pp. 151–162). New York, New York, USA: ACM Press. doi: 10.1145/2601381.2601394

Perlis, A. J. (1982, September). Special Feature: Epigrams on Programming. *ACM SIGPLAN Notices*, *17*(9), 7–13. doi: 10.1145/947955.1083808

Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice Hall.

Petri, C. A. (1962). *Kommunikation mit Automaten* (Dissertation, Technische Universität Darmstadt, Bonn, Germany). Retrieved from `http://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/`

Petty, M. D., & Weisel, E. W. (2003a, April). A Composability Lexicon. In *Proceedings of the Spring 2003 Simulation Interoperability Workshop* (pp. 181–187).

Petty, M. D., & Weisel, E. W. (2003b). A Formal Basis for a Theory of Semantic Composability. In *Proceedings of the Spring 2003 Simulation Interoperability Workshop*.

Pierce, B. C. (2002). *Types and Programming Languages* (1st ed.). Cambridge, MA, USA: The MIT Press.

Pirotte, A. (1982, September). A Precise Definition of Basic Relational Notions and of the Relational Algebra. *ACM SIGMOD Record*, *13*(1), 30–45. doi: 10.1145/984514.984516

Plotkin, G. D. (2004a). The Origins of Structural Operational Semantics. *Journal of Logic and Algebraic Programming*, *60-61*(SUPPL.), 3–15. doi: 10.1016/j.jlap.2004.03.009

Plotkin, G. D. (2004b). A Structural Approach to Operational Semantics. *The Journal of Logic*

and Algebraic Programming, *60-61*(SUPPL.), 17–139. doi: 10.1016/j.jlap.2004.05.001

Poslad, S. (2009). *Ubiquitous Computing: Smart Devices, Environments and Interactions* (1st ed.). Chichester, UK: John Wiley & Sons, Inc.

Praehofer, H. (1991). Systems Theoretic Formalisms for Combined Discrete-Continuous System Simulation. *International Journal of General Systems*, *19*(3), 226–240. doi: 10.1080/03081079108935175

Praehofer, H. (1992). *System Theoretic Foundations for Combined Discrete-Continuous System Simulation* (Dissertation). Johannes Kepler University Linz, Wien, Austria.

Praehofer, H., & Pree, D. (1993, December). Visual Modeling of DEVS-based Multiformalism Systems Based on Higraphs. In *Proceedings of the 1993 Winter Simulation Conference* (pp. 595–603). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1145/256563.256737

Risco-Martín, J. L., de la Cruz, J. M., Mittal, S., & Zeigler, B. P. (2009, November). eUDEVS: Executable UML with DEVS Theory of Modeling and Simulation. *SIMULATION*, *85*(11-12), 750–777. doi: 10.1177/0037549709104727

Rivera, J. E., Duran, F., & Vallecillo, A. (2009, September). A Graphical Approach for Modeling Time-dependent Behavior of DSLs. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 51–55). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/VLHCC.2009.5295300

Roberts, S. D., & Pegden, D. (2017, December). The History of Simulation Modeling. In W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. A. Wainer, & E. Page (Eds.), *Proceedings of the 2017 Winter Simulation Conference* (pp. 308–323). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2017.8247795

Rodriguez, D. A., & Wainer, G. A. (1999). Redefinition of a Specification Language for Cell-DEVS Models. In *Proceedings of Information Systems Analysis and Synthesis, ISAS'99.*

Rogovchenko, O., & Malenfant, J. (2010). Composition and Compositionality in a Component Model for Autonomous Robots. In B. Baudry & E. Wohlstadter (Eds.), *Software composition* (pp. 34–49). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-14046-4_3

Röhl, M. (2006). Platform Independent Specification of Simulation Model Components. In W. Borutzky, A. Orsoni, & R. Zobel (Eds.), *Proceedings of the 20th European Conference on Modelling and Simulation (ECMS 2006)* (pp. 220–225). Nottingham, UK: ECMS.

Röhl, M. (2008). *Definition und Realisierung einer Plattform zur modellbasierten Komposition von Simulationsmodellen* (Dissertation, Universität Rostock). doi: 10.18453/rosdok_id00000298

Röhl, M., & Morgenstern, S. (2007, December). Composing Simulation Models Using Interface Definitions Based on Web Service Descriptions. In S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, & J. J. Barton (Eds.), *Proceedings of the 2007 Winter Simulation Conference* (pp. 815–822). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2007.4419677

Röhl, M., & Uhrmacher, A. M. (2006, December). Composing Simulations from XML-Specified Model Components. In L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, & R. M. Fujimoto (Eds.), *Proceedings of the 2006 Winter Simulation Conference* (pp. 1083–1090). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/WSC.2006.323198

Röhl, M., & Uhrmacher, A. M. (2008, December). Definition and Analysis of Composition Structures for Discrete-Event Models. In S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, & J. W. Fowler (Eds.), *Proceedings of the 2008 Winter Simulation Conference* (pp. 942–950). Piscataway, NJ, USA: Institute of Electrical and Electronics

Engineers, Inc. doi: 10.1109/WSC.2008.4736160

Ropohl, G. (1999). Philosophy of Socio-technical Systems. *Techné: Research in Philosophy and Technology*, *4*(3), 186–194. doi: 10.5840/techne19994311

Rosen, K. H. (2007). *Discrete Mathematics and Its Applications.* McGraw-Hill Higher Education.

Rowson, J. A., & Sangiovanni-Vincentelli, A. (1997). Interface-based Design. In *Proceedings of the 34th annual Design Automation Conference (DAC '97)* (pp. 178–183). New York, New York, USA: ACM Press. doi: 10.1145/266021.266060

Rozenblit, J. W., & Zeigler, B. P. (1993, December). Representing and Constructing System Specifications Using the System Entity Structure Concepts. In G. W. Evans, M. Mollaghasemi, E. C. Russell, & W. E. Biles (Eds.), *Proceedings of the 1993 Winter Simulation Conference* (pp. 604–611). New York, NY, USA: ACM Press. doi: 10.1145/256563.256742

Russel, S. J., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Upper Saddle River, NJ, USA: Prentice Hall.

Russell, B. (1903). *The Principles of Mathematics* (1st ed.). Cambridge, UK: Cambridge University Press.

Rybacki, S., Haack, F., Wolf, K., & Uhrmacher, A. M. (2014). Developing Simulation Models - from Conceptual to Executable Model and Back - an Artifact-based Workflow Approach. In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques (SIMUTools)* (pp. 21–30). doi: 10.4108/icst.simutools.2014.254650

Sanmugalingam, K., & Coulouris, G. (2002, September). A Generic Location Event Simulator. In G. Borriello & L. E. Holmquist (Eds.), *Proceedings of the 4th International Conference on Ubiquitous Computing (UbiComp '02)* (pp. 308–315). Berlin, Germany: Springer-Verlag. doi: 10.1007/3-540-45809-3_24

Sarjoughian, H. S. (2006, December). Model Composability. In L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, & R. M. Fujimoto (Eds.), *Proceedings of the 2006 Winter Simulation Conference* (pp. 149–158). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2006.323047

Sarjoughian, H. S., & Elamvazhuthi, V. (2009). CoSMoS: A Visual Environment for Component-based Modeling, Experimental Design, and Simulation. In *Proceedings of the Second International ICST Conference on Simulation Tools and Techniques.* ICST. doi: 10.4108/ICST.SIMUTOOLS2009.5744

Sarjoughian, H. S., & Huang, D. (2005). A Multi-formalism Modeling Composability Framework: Agent and Discrete-event Models. In A. Boukerche, S. J. Turner, D. Roberts, & G. K. Theodoropoulos (Eds.), *Proceedings of the 2005 Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications* (pp. 249–256). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/DISTRA.2005.4

Sarjoughian, H. S., & Zeigler, B. P. (2000, December). DEVS and HLA: Complementary paradigms for modeling and simulation? *Transactions of the Society for Computer Simulation International*, *17*(4), 187–197.

Savory, P., & Mackulak, G. (1994). The Science of Simulation Modeling. In C. E. Knadler Jr. & H. Vakilzadian (Eds.), *International Conference on Simulation in Engineering Education: Proceedings of the 1994 Western Multiconference* (pp. 115–119). Society for Computer Simulation.

Schalles, C. (2013). *Usability Evaluation of Modeling Languages.* Wiesbaden, Germany: Springer Gabler. doi: 10.1007/978-3-658-00051-6

Schmidt, J. W., & Taylor, R. E. (1970). *Simulation and Analysis of Industrial Systems.* Richard D. Irwin, Inc.

Self-reproducible DEVS formalism. (2005, November). *Journal of Parallel and Distributed Computing*, *65*(11), 1329–1336. doi: 10.1016/j.jpdc.2005.05.004

Shannon, R. E. (1975). *Systems Simulation: The Art of Science*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Shiginah, F. A. S. B. (2006). *Multi-layer Cellular DEVS Formalism for Faster Model Development and Simulation Efficiency* (Unpublished doctoral dissertation). University of Arizona.

Silverman, B. G., Might, R., Dubois, R., Shin, H., Johns, M., & Weaver, R. (2001). Toward A Human Behavior Models Anthology for Synthetic Agent Development. In *Proceedings of the 10th Conference on Computer Generated Forces and Behavioral Representation* (pp. 277–285).

Smith, J. M., & Smith, D. C. P. (1977, June). Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, *2*(2), 105–133. doi: 10.1145/320544.320546

Steiniger, A., Krüger, F., & Uhrmacher, A. M. (2012, December). Modeling Agents and their Environment in Multi-Level-DEVS. In C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, & A. M. Uhrmacher (Eds.), *Proceedings of the 2012 Winter Simulation Conference.* Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. (Article No. 233) doi: 10.1109/WSC.2012.6465113

Steiniger, A., & Uhrmacher, A. M. (2010, December). Modeling and Simulation for User Assistance in Smart Environments. In B. Johansson, S. Jain, J. Montoya-Torres, & E. Yücesan (Eds.), *Proceedings of the 2010 Winter Simulation Conference* (pp. 490–499). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2010.5679138

Steiniger, A., & Uhrmacher, A. M. (2013, July). Composing Variable Structure Models: A Revision of COMO. In T. Ören, J. Kacprzyk, L. Leifsson, M. S. Obaidat, & S. Koziel (Eds.), *Proceedings of the 3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2013)* (pp. 286–293). SCITEPRESS. doi: 10.5220/0004486302860293

Steiniger, A., & Uhrmacher, A. M. (2016, January). Intensional Couplings in Variable-Structure Models: An Exploration Based on Multilevel-DEVS. *ACM Transactions on Modeling and Computer Simulation*, *26*(2), 9-1–9-27. doi: 10.1145/2818641

Steiniger, A., Zinn, S., Gampe, J., Willekens, F., & Uhrmacher, A. M. (2014, December). The Role of Languages for Modeling and Simulating Continuous-Time Multi-Level Models in Demography. In A. Tolk, S. D. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley, & J. A. Miller (Eds.), *Proceedings of the 2014 Winter Simulation Conference* (pp. 2978–2989). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2014.7020137

Stiffel, S. G. (2014). *Simulating the Issue Lifecycle with Attached SLAs in the Context of an IT Service Corporation* (Unpublished master's thesis). University of Rostock, Rostock, Germany.

Sundstrom, T. (2013). *Mathematical Reasoning: Writing and Proof* (3rd ed.). CreateSpace Independent Publishing Platform.

Syriani, E., & Vangheluwe, H. L. M. (2010, December). DEVS as a Semantic Domain for Programmed Graph Transformation. In G. A. Wainer & P. J. Mosterman (Eds.), *Discrete-Event Modeling and Simulation: Theory and Applications* (1st ed., pp. 3–28). Boca Raton, FL, USA: CRC Press.

Syriani, E., & Vangheluwe, H. L. M. (2013, May). A Modular Timed Graph Transformation Language for Simulation-based Design. *Software & Systems Modeling*, *12*(2), 387–414. doi: 10.1007/s10270-011-0205-0

Syropoulos, A. (2001, August). Mathematics of Multisets. In C. S. Calude, G. Păun, G. Rozenberg, & A. Salomaa (Eds.), *Multiset processing: Mathematical, computer science, and molecular computing points of view* (pp. 347–358). Berlin, Heidelberg,

Germany: Springer-Verlag Berlin Heidelberg. doi: 10.1007/3-540-45523-X_17

Szabo, C. (2010). *Composable Simulation Models and Their Formal Validation* (Unpublished doctoral dissertation). National University of Singapore.

Szabo, C., & Teo, Y. M. (2007, March). On Syntactic Composability and Model Reuse. In D. Al-Dabass, R. Zobel, A. Abraham, & S. Turner (Eds.), *Proceedings of the First Asia International Conference on Modelling & Simulation* (pp. 230–237). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/AMS.2007.74

Szabo, C., & Teo, Y. M. (2009, June). An Approach for Validation of Semantic Composability in Simulation Models. In *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation* (pp. 3–10). Washington, DC, USA: IEEE Computer Society. doi: 10.1109/PADS.2009.14

Szyperski, C. A. (2002). *Component Software: Beyond Object-Oriented Programming* (2nd ed.). London, UK: Pearson Education Limited.

Teo, Y. M., & Szabo, C. (2008, April). CODES: An Integrated Approach to Composable Modeling and Simulation. In *41st Annual Simulation Symposium (ANSS-41 2008)* (pp. 103–110). IEEE. doi: 10.1109/ANSS-41.2008.24

Thomas, C. (1994). Interface-oriented Classification of DEVS Models. In *Proceedings of the 5th Annual Conference on AI, and Planning in High Autonomy Systems (AIS 1994)* (pp. 208–213). IEEE Comput. Soc. Press. doi: 10.1109/AIHAS.1994.390472

Tocher, K. D. (1963). *The Art of Simulation*. London, UK: English Universities Press. Retrieved from `https://archive.org/details/TheArtOfSimulation`

Tolk, A. (2013, October). Interoperability, Composability, and Their Implications for Distributed Simulation: Towards Mathematical Foundations of Simulation Interoperability. In *17th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, {DS-RT} 2013* (pp. 3–9). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/DS-RT.2013.8

Tolk, A., & Miller, J. A. (2011, August). Enhancing Simulation Composability and Interoperability Using Conceptual/Semantic/Ontological models. *Journal of Simulation*, *5*(3), 133–134. doi: 10.1057/jos.2011.18

Traore, M. K. (2006, December). Analyzing Static and Temporal Properties of Simulation Models. In L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, & R. M. Fujimoto (Eds.), *Proceedings of the 2006 Winter Simulation Conference* (pp. 897–904). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2006.323173

Uhrmacher, A. M. (1995, March). Reasoning about Changing Structure: A Modeling Concept for Ecological Systems. *Applied Artificial Intelligence: An International Journal*, *9*(2), 157–180. doi: 10.1080/08839519508945472

Uhrmacher, A. M. (2001, April). Dynamic Structures in Modeling and Simulation: A Reflective Approach. *ACM Transactions on Modeling and Computer Simulation*, *11*(2), 206–232. doi: 10.1145/384169.384173

Uhrmacher, A. M., Ewald, R., John, M., Maus, C., Jeschke, M., & Biermann, S. (2007, December). Combining Micro and Macro-modeling in DEVS for Computational Biology. In S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, & R. R. Barton (Eds.), *Proceedings of the 2007 Winter Simulation Conference* (pp. 871–880). Piscataway, NJ: IEEE Press. doi: 10.1109/WSC.2007.4419683

Uhrmacher, A. M., Himmelspach, J., & Ewald, R. (2010, December). Effective and Efficient Modeling and Simulation with DEVS Variants. In G. A. Wainer & P. J. Mosterman (Eds.), *Discrete-event modeling and simulation: Theory and applications* (1st ed., pp. 139–176). Boca Raton, FL, USA: CRC Press.

Uhrmacher, A. M., Himmelspach, J., Röhl, M., & Ewald, R. (2006, December). Introducing Variable Ports and Multi-couplings for Cell Biological Modeling in DEVS. In L. F. Per-

rone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, & R. M. Fujimoto (Eds.), *Proceedings of the 2006 Winter Simulation Conference* (pp. 832–840). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/WSC.2006.323165

Uhrmacher, A. M., & Kuttler, C. (2006). Multi-level modeling in Systems Biology by Discrete Event Approaches. *IT Themenheft Systems Biology*, *48*(3), 148–153. doi: 10.1524/itit.2006.48.3.148

Uhrmacher, A. M., & Priami, C. (2005, December). Discrete Event Systems Specification in Systems Biology - A Discussion of Stochastic Pi Calculus and DEVS. In M. E. Kuhl, N. M. Steiger, F. B. Armstrong, & J. A. Joines (Eds.), *Proceedings of the 2005 Winter Simulation Conference* (pp. 317–326). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2005.1574266

Uhrmacher, A. M., & Zeigler, B. P. (1996, March). Variable Structure Models in Object-oriented Simulation. *International Journal of General Systems*, *24*(4), 359–375. doi: 10.1080/03081079608945128

Valentin, E. C., & Verbraeck, A. (2002, December). Guidelines for Designing Simulation Building Blocks. In E. Yücesan, C.-H. Chen, J. L. Snowdon, & J. M. Charnes (Eds.), *Proceedings of the 2002 Winter Simulation Conference* (pp. 563–571). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2002.1172932

Valentin, E. C., Verbraeck, A., & Sol, H. G. (2003). Advantages and Disadvantages of Building Blocks in Simulation Studies. In A. Verbraeck & V. Hlupic (Eds.), *Proceedings of the 15th European Simulation Symposium* (pp. 142–148). Delft, Netherlands: SCS-European Publishing House.

Van Tendeloo, Y., & Vangheluwe, H. L. M. (2017, December). Classic DEVS Modelling and Simulation. In W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. A. Wainer, & E. Page (Eds.), *Proceedings of the 2017 Winter Simulation Conference* (pp. 644–658). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2017.8247822

Van Tendeloo, Y., & Vangheluwe, H. L. M. (2018). *Extending the DEVS Formalism with Initialization Information* (Tech. Rep.). University of Antwerp. Retrieved from `http://arxiv.org/abs/1802.04527`

van der Giezen, M. (2011, August). Mitochondria and the Rise of Eukaryotes. *BioScience*, *61*(8), 594–601. doi: 10.1525/bio.2011.61.8.5

Vangheluwe, H. L. M. (2000). DEVS as a Common Denominator for Multi-formalism Hybrid Systems Modelling. In *Proceedings of the 2000 IEEE International Symposium on Computer Aided Control System Design* (pp. 129–134). Piscataway, NJ, USA: IEEE Computer Society. doi: 10.1109/CACSD.2000.900199

Vangheluwe, H. L. M. (2001). *The Discrete EVent System specification (DEVS) formalism.* Retrieved from `https://www.cs.mcgill.ca/~hv/classes/MS/DEVS.pdf`

Vangheluwe, H. L. M., de Lara, J., & Mosterman, P. J. (2002, April). An Introduction to Multi-Paradigm Modelling and Simulation. In F. J. Barros & N. Giambiasi (Eds.), *Proceedings of the 2002 Conference on AI, Simulation and Planning in High Autonomy Systems (AIS 2002)* (pp. 9–20). Retrieved from `https://biblio.ugent.be/publication/158059`

Varga, A. (2001). The OMNET++ Discrete Event Simulation System. In *Proceedings of the 15th European Simulation Multiconference (ESM'2001)* (pp. 319–324). SCS Europe.

Verbraeck, A. (2004). Component-based Distributed Simulations. The Way Forward? In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS'04)* (pp. 141–148). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/PADS.2004.1301295

Verbraeck, A., & Valentin, E. C. (2008, December). Design Guidelines for Simulation Building Blocks. In S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, &

J. W. Fowler (Eds.), *Proceedings of the 2008 Winter Simulation Conference* (pp. 923–932). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2008.4736158

Vijayaraghavan, V., & Barton, J. J. (2001). WISE – A Simulator Toolkit for Ubiquitous Computing Scenarios. In *Proceedings of the UbiTools'01: Workshop on Application Models and Programming Tools for Ubiquitous Computing.*

von Bertalanffy, L. (1969). *General System Theory: Foundations, Development, Applications* (revised ed.). New York, NY, USA: George Braziller, Inc.

von Neumann, J. (1966). *Theory of Self-reproducing Automata* (A. W. Burks, Ed.). Champaign, IL, USA: University of Illinois Press.

Wainer, G. A. (1998). *Discrete-event Cellular Models with Explicit Delays* (Unpublished doctoral dissertation). Université d' Aix-Marseille III.

Wainer, G. A. (1999). Abstract Cell-DEVS Simulators. In *Proceedings of the 5th International Conference on Information Systems Analysis and Synthesis.*

Wainer, G. A. (2002, November). CD++: A Toolkit to Develop DEVS Models. *Software: Practice and Experience*, *32*(13), 1261–1306. doi: 10.1002/spe.482

Wainer, G. A., Frydman, C. S., & Giambiasi, N. (1997). An Environment for Simulation of Cellular DEVS Models. In *Proceedings of the 1997 SCS European Simulation Multiconference.* Retrieved from `http://cell-devs.sce.carleton.ca/publications/1997/WFG97`

Wainer, G. A., & Giambiasi, N. (1998). *Specification, Modeling and Simulation of Timed Cell-DEVS Spaces* (Tech. Rep.). Buenos Aires, Argentina: Universidad de Buenos Aires.

Wainer, G. A., & Liu, Q. (2009, March). Tools for Graphical Specification and Visualization of DEVS Models. *SIMULATION*, *85*(3), 131–158. doi: 10.1177/0037549708101182

Walter, T., Parreiras, F. S., & Staab, S. (2014, February). An Ontology-based Framework for Domain-specific Modeling. *Software & Systems Modeling*, *13*(1), 83–108. doi: 10.1007/s10270-012-0249-9

Wang, Y.-H. (1992). *Discrete-event Simulation on a Massively Parallel Computer* (Dissertation, University of Arizona). Retrieved from `http://arizona.openrepository.com/arizona/handle/10150/185913`

Wang, Y.-H., & Zeigler, B. P. (1993, July). Extending the DEVS Formalism for Massively Parallel Simulation. *Discrete Event Dynamic Systems: Theory and Applications*, *3*(2-3), 193–218. doi: 10.1007/BF01439849

Warnke, T., Klabunde, A., Steiniger, A., Willekens, F., & Uhrmacher, A. M. (2015). ML3: A Language for Compact Modeling of Linked Lives in Computational Demography. In L. Yilmaz, I.-C. Moon, W. K. Chan, T. Roeder, C. Mascal, & M. Rossetti (Eds.), *Proceedings of the 2015 Winter Simulation Conference* (pp. 2764–2775). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1109/WSC.2015.7408382

Weinreich, R., & Sametinger, J. (2001, June). Component Models and Component Services: Concepts and Principles. In G. T. Heineman & W. T. Councill (Eds.), *Component-Based Software Engineering: Putting the Pieces Together* (pp. 33–48). Upper Saddle River, NJ, USA: Addison-Wesley.

Weiser, M. (1999, July). The Computer for the 21st Century. *Mobile Computing and Communications Review*, *3*(3), 3–11. doi: 10.1145/329124.329126

Weyns, D., Helleboogh, A., Holvoet, T., & Schumacher, M. (2009, January). The Agent Environment in Multi-Agent Systems: A Middleware Perspective. *Multiagent and Grid Systems*, *5*(1), 93–108. doi: 10.3233/MGS-2009-0121

Wilson, W. K. (1998). *The Essentials of Logic* (revised ed.). Piscataway, NJ, USA: Research & Education Association.

Winsberg, E. (2009, September). Computer Simulation and the Philosophy of Science. *Philosophy Compass*, *4*(5), 835–845. doi: 10.1111/j.1747-9991.2009.00236.x

Wittgenstein, L. J. J. (1958). *Philosophical Investigations* (3rd ed.). Englewood Cliffs, NJ, USA: Prentice Hall.

Wolfram, S. (1984, October). Cellular Automata as Models of Complexity. *Nature*, *311*(5985), 419–424. doi: 10.1038/311419a0

Wymore, A. W. (1967). *A Mathematical Theory of Systems Engineering: The Elements*. New York, NY, USA: John Wiley & Sons, Inc.

Yilmaz, L., & Ören, T. I. (2007). Agent-directed Simulation Systems Engineering. In G. A. Wainer (Ed.), *Proceedings of the 2007 Summer Computer Simulation Conference* (pp. 897–904). San Diego, CA, USA: Society for Computer Simulation International. doi: 10.1145/1357910.1358050

Zeigler, B. P. (1976). *Theory of Modeling and Simulation* (1st ed.). New York, NY, USA: John Wiley & Sons, Inc.

Zeigler, B. P. (1984). *Multifacetted Modelling and Discrete Event Simulation* (1st ed.). Academic Press, Inc.

Zeigler, B. P. (1987, January). Toward a simulation methodology for variable structure modeling. In M. S. Elzas, T. I. Ören, & B. P. Zeigler (Eds.), *Modelling and simulation methodology in the artificial intelligence era* (pp. 195–210). Amsterdam, The Netherlands: Elsevier Science Ltd.

Zeigler, B. P. (1990). *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems* (1st ed.). San Diego, CA, USA: Academic Press, Inc.

Zeigler, B. P., & Hammonds, P. E. (2007). *Modeling and Simulation-Based Engineering: Introducing Pragmatics into Ontologies for Net-Centric Information Exchange* (1st ed.). Burlington, MA, USA: Academic Press.

Zeigler, B. P., Moon, Y., Kim, D., & Kim, J. G. (1996). DEVS-C++: a high performance modelling and simulation environment. In *Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences* (pp. 350–359). IEEE. doi: 10.1109/HICSS.1996.495481

Zeigler, B. P., & Muzy, A. (2016, April). Some Modeling & Simulation Perspectives on Emergence in System-of-Systems. In *Proceedings of the Modeling and Simulation of Complexity in Intelligent, Adaptive and Autonomous Systems 2016 (MSCIAAS 2016) and Space Simulation for Planetary Space Exploration (Space 2016)*. San Diego, CA, USA: Society for Computer Simulation International. (Article No. 11)

Zeigler, B. P., & Ören, T. I. (1986, December). Multifaceted, Multiparadigm Modeling Perspectives: Tools for the 90's. In J. Wilson, J. Henriksen, & S. Roberts (Eds.), *Proceedings of the 1986 Winter Simulation Conference* (pp. 708–712). Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Inc. doi: 10.1145/318242.318513

Zeigler, B. P., & Praehofer, H. (1990). Systems Theory Challenges in the Simulation of Variable Structure and Intelligent Systems. In F. Pichler & R. Moreno-Diaz (Eds.), *Computer Aided Systems Theory - EUROCAST'89* (pp. 41–51). Berlin, Germany: Springer-Verlag. doi: 10.1007/3-540-52215-8_4

Zeigler, B. P., Praehofer, H., & Kim, T. G. (2000). *Theory of Modeling and Simulation* (2nd ed.). San Diego, CA, USA: Academic Press.

Zeigler, B. P., & Sarjoughian, H. S. (1999, March). Support for Hierarchical Modular Component-based Model Construction in DEVS/HLA. In *Spring simulation interoperability workshop.*

Zinn, S. (2011). *A Continuous-Time Microsimulation and First Steps Towards a Multi-Level Approach in Demography* (Dissertation, University of Rostock, Rostock, Germany). doi: 10.18453/rosdok_id00000951

Eidesstattliche Erklärung

Hiermit erkläre ich durch eigenhändige Unterschrift, die vorliegende Dissertation selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die aus den Quellen direkt oder indirekt übernommenen Gedanken sind also solche kenntlich gemacht. Die Dissertation ist in dieser Form noch keiner anderen Prüfungsbehörde vorgelegt worden.

Satow, 01.10.2018

Ort, Datum	Unterschrift

Curriculum Vitae

Personal Data

Name:	Alexander Steiniger
Date of birth:	25.03.1983
Place of birth:	Bad Muskau, Germany
Nationality:	German

Working Experience

since 08/2017	Application developer and software architect, comdirect bank AG, Rostock, Germany
10/2015 – 01/2017	Software developer, Empora Group GmbH, Wismar, Germany
05/2013 – 08/2015	Research associate and assistant in the modeling and simulation group, Institute of Computer Science, University of Rostock, Germany

Education

12/2009 – 07/2018	Doctoral student in the field of modeling and simulation at the Institute of Computer Science, University of Rostock, Rostock, Germany
12/2009 – 03/2013	Scholarship holder in the research training group "MuSAMA," University of Rostock, Rostock, Germany
10/2003 – 11/2009	Diploma studies of informatics with a specialization in computer graphics and visualization at the University of Rostock, Rostock, Germany
05/1995 – 06/2002	Abitur at the Friderico Francisceum Gymnasium, Bad Doberan, Germany

Alternative Civilian Service

09/2002 – 06/2003	Alternative civilian service, rehabilitation clinic "Moorbad," Bad Doberan, Germany

Thesis Statements

Title: Toward Composing Variable Structure Models and Their Interfaces: A Case of Intensional Coupling Definitions
Name: Alexander Steiniger

1. Many complex systems of interest consist of a number of homogeneous or heterogeneous components and have a dynamic structure, i. e., their structure changes over time.

2. Component-based modeling allows us to reduce the complexity of a model by defining it as a composition of smaller, interacting components. Furthermore, component-based modeling allows us to reduce the costs of developing models by reusing already existing components.

3. Traditional component-based modeling aims at a separation between component interfaces and component implementations and assumes a static model structures.

4. Variable structure modeling allows the modeler to explicitly capture the structure variability of a system of interest in its model. In variable structure modeling, structure changes become first-order abstractions.

5. Since traditional composition takes place at configuration time and variable structures are a runtime phenomena, there is a contradiction between both paradigms and their combination has certain implications on aspects such as (syntactic) composability.

6. By using supersets and intensional couplings we still can make statements about the composability of a composition beyond its initial state, if the involved components adhere to their interface definitions.

7. Maintaining structural consistency when specifying couplings in variable structure models is challenging, even more when variable interfaces are involved, since components and ports that are available at a certain time may not be available at another time.

8. We can adopt and exploit intensional definition techniques for defining couplings in variable structure models. Rather than enumerating each possible concrete coupling that can exist within the different incarnations of a variable structure model, intensional coupling definitions allow the modeler to define couplings in a more compact yet powerful way.

9. One intensional coupling definition can “encode” an arbitrary number of concrete couplings.

10. Intensional couplings have to be translated into concrete model couplings during simulation (model execution).

11. The translation algorithm can check the consistency of each potential concrete coupling that can be derived from an intensional coupling definition and discard all inconsistent couplings. Thus the translation can guarantee correctness by construction (structural consistency).

12. By exploiting intensional coupling definitions and a corresponding translation mechanism, the modeler does not need to take care about maintaining structural consistency, with respect to couplings.

13. The use of intensional coupling definitions is not confined to variable structure models. Intensional couplings can also be used in static structure models enabling modelers to streamline their model specifications.

14. Defining intensional couplings based on runtime instances of model interfaces allow us to define even more complex and sophisticated communication patterns concisely, e. g., based on concrete values of interface attributes.

15. The incorporation of an intensional coupling mechanism into the variable structure and multi-level modeling formalism Multi-Level DEVS (ML-DEVS) proves the applicability of the concept.

16. Intensional definitions cannot only be used to define couplings but also, e. g., to define and constrain sets of components that can become available in a variable composition.

Publications

Book Sections

1. Martin Nyolt, **Alexander Steiniger**, Sebastian Bader, and Thomas Kirste. Describing and Evaluating Assistance Using APDL. *Smart Modeling and Simulation for Complex Systems: Practice and Theory*, pp. 59–81, 2015

   DOI: 10.1007/978-4-431-55209-3_5

Journal Articles

1. **Alexander Steiniger** and Adelinde M. Uhrmacher. Intensional Couplings in Variable-Structure Models: An Exploration Based on Multilevel-DEVS. *ACM Transactions on Modeling and Computer Simulation* (TOMACS), 26(2):9-1–9-27, 2016

   DOI: 10.1145/2818641

Conference Contributions

1. Tom Warnke, Anna Klabunde, **Alexander Steiniger**, Frans Willekens, and Adelinde M. Uhrmacher. ML3: A Language for Compact Modeling of Linked Lives in Computational Demography (invited paper). In *Proceedings of the 2015 Winter Simulation Conference* (WSC '15), pp. 2764–2775, 2015

   DOI: 10.1109/WSC.2015.7408382

2. **Alexander Steiniger**, Sabine Zinn, Jutta Gampe, Frans Willekens, and Adelinde M. Uhrmacher. The Role of Languages for Modeling and Simulating Continuous-Time Multi-Level Models in Demography (invited paper). In *Proceedings of the 2014 Winter Simulation Conference* (WSC '14), pp. 2978–2989, 2014

   DOI: 10.1109/WSC.2014.7020137

3. **Alexander Steiniger** and Adelinde M. Uhrmacher. Composing Variable Structure Models: A Revision of COMO. In *Proceedings of the 3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications* (SIMULTECH 2013), pp. 286–293, 2013

   DOI: 10.5220/0004486302860293

4. **Alexander Steiniger**, Frank Krüger, and Adelinde M. Uhrmacher. Modeling Agents and their Environment in Multi-Level-DEVS. In *Proceedings of the 2012 Winter Simulation Conference* (WSC '12), Article 233, 12 pages, 2012

   DOI: 10.1109/WSC.2012.6465113

5. **Alexander Steiniger** and Adelinde M. Uhrmacher. Modeling and Simulation for User Assistance in Smart Environments. In *Proceedings of the 2010 Winter Simulation Conference* (WSC'10), pp. 490–499, 2010

   DOI: 10.1109/WSC.2010.5679138

Workshop Contributions

1. Martin Nyolt, **Alexander Steiniger**, Sebastian Bader, and Thomas Kirste. Describing and Evaluating Assistance using APDL. In *Proceedings of the International Workshop on Smart Simulation and Modelling for Complex Systems* (SSMCS'13), pp. 38–49, 2013

   URL: `http://www.uow.edu.au/~fren/SSMCS2013/SSMCS2013_Proceeding_V3.pdf` (last accessed February 2018)

2. Frank Krüger, **Alexander Steiniger**, Sebastian Bader, and Thomas Kirste. Evaluating the robustness of activity recognition using computational causal behavior models. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing* (UbiComp '12), pp. 1066–1074, 2012

   Workshop: SAGAware 2012 (held in conjunction with UbiComp 2012)

   DOI: 10.1145/2370216.2370443

Posters

1. Danuha Peng, **Alexander Steiniger**, Tobias Helms, and Adelinde M. Uhrmacher. Towards Composing ML-Rules Models. In *Proceedings of the 2013 Winter Simulation Conference* (WSC '13), pp. 4010–4011, 2013

   ISBN: 978-1-4799-2077-8

   URL: `http://informs-sim.org/wsc13papers/includes/files/397.pdf`[1]

Extended Abstracts

1. **Alexander Steiniger**. Component-based Modeling and Simulation for Smart Environments. In *Proceedings of the Joint Workshop of the German Research Training Groups in Computer Science, Dagstuhl 2012*, p. 242, 2012

   ISBN: 978-3-8482-0022-1

2. **Alexander Steiniger**. Component-based Modeling and Simulation for Smart Environments. In *Proceedings of the Joint Workshop of the German Research Training Groups in Computer Science*, p. 206, 2011

   ISBN: 978-3-942183-36-9

3. **Alexander Steiniger**. Component-based Modeling and Simulation for Smart Environments In *Proceedings of the Joint Workshop of the German Research Training Groups in Computer Science, Algorithmic synthesis of reactive and discrete-continuous systems* (AlgoSyn 2010), p. 143, 2010

   ISBN: 978-3-86130-146-2

[1] last accessed February 2018