COORDINATION AND VERIFICATION OF HETEROGENEOUS BEHAVIORAL SYSTEMS

Doctoral Dissertation by
Tim Kräuter

Thesis submitted for
the degree of Philosophiae Doctor (PhD)
in
Computer Science:
Software Engineering, Sensor Networks and Engineering Computing

Department of Computer Science,
Electrical Engineering and Mathematical Sciences

Faculty of Technology, Environmental and Social Sciences

Western Norway University of Applied Sciences

May 5, 2025

Series of dissertations submitted to
the Faculty of Technology, Environmental and Social Sciences
Western Norway University of Applied Sciences.

ISBN: 978-82-8461-200-3 (print)
ISBN: 978-82-8461-201-0 (digital)

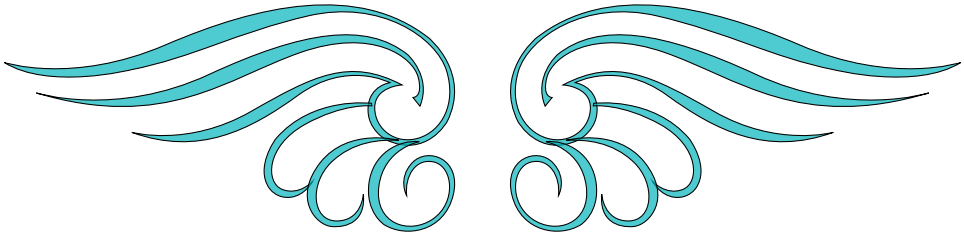
Author: Tim Kräuter
Title: Coordination and Verification of Heterogeneous Behavioral Systems

Printed production:
Aksell / Western Norway University of Applied Sciences

Bergen, Norway, 2025

TO MY FRIENDS, FAMILY, AND LOVED ONES,

Without you, this Ph.D. would never have taken off



PREFACE

The author of this thesis has been employed as a PhD research fellow in the software engineering research group at the *Department of Computer Science, Electrical Engineering and Mathematical Science* at Western Norway University of Applied Sciences.

The author has been enrolled in the PhD program in Computer Science: Software Engineering, Sensor Networks, and Engineering Computing, specializing in *Software Engineering* under the supervision of Prof. Adrian Rutle, Prof. Harald König, and Prof. Yngve Lamo.

This thesis is organized into two parts. Part I provides an introduction to the research field, background information, and state-of-the-art, as well as a summary of the contributions made in this thesis. Part II includes a collection of peer-reviewed papers—Papers A, B, C, and E—as well as a submitted paper, Paper D. Among these, Papers B and C are journal articles.

Paper A Harald König, Uwe Wolter, and **Tim Kräuter**. "Structural Operational Semantics for Heterogeneously Typed Coalgebras." In Proceedings of the 10th Conference on Algebra and Coalgebra in Computer Science (CALCO 2023), September 2023, doi:10.4230/LIPIcs.CALCO.2023.7.

Paper B **Tim Kräuter**, Harald König, Adrian Rutle, Yngve Lamo, and Patrick Stünkel. "Behavioral consistency in multi-modeling." Journal of Object Technology. Vol. 22, Issue 2, July 2023, doi:10.5381/jot.2023.22.2.a9.

Paper C **Tim Kräuter**, Adrian Rutle, Harald König, and Yngve Lamo. "A higher-order transformation approach to the formalization and analysis of BPMN using graph transformation systems." Logical Methods In Computer Science. Vol. 20, Issue 4, October, 2024, doi:10.46298/LMCS-20(4:4)2024.

Paper D **Tim Kräuter**, Adrian Rutle, Yngve Lamo, Harald König, Francisco Durán. "Towards the Coordination and Verification of Heterogeneous Systems with Data and Time." *Submitted to the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, April 2025.

Paper E **Tim Kräuter**, Patrick Stünkel, Adrian Rutle, Yngve Lamo, and Harald König. "BPMN Analyzer 2.0: Instantaneous, Comprehensible, and Fixable Control Flow Analysis for Realistic BPMN Models." Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Forum at BPM 2024 co-located with 22nd International Conference on Business Process Management (BPM 2024), September 2024, https://ceur-ws.org/Vol-3758.

Other Scientific Publications

During my PhD studies, I published four additional scientific papers not included in the thesis. **Paper 1** is an initial version of Paper B presented at the Doctoral Symposium during the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS) in 2021.

Papers 2 & 4 describe the Visual Debugger Plugin for IntelliJ IDEA[1]. The project originated during the mandatory coursework of my PhD. The Visual Debugger has since been downloaded over 15.000 times[2]. It showcases applying model-driven software engineering in practice to improve comprehension during debugging.

Paper 3 received the **Best Paper Award** at the International Conference on Graph Transformation (ICGT) in 2023 and was later extended into Paper C.

1. **Tim Kräuter**. "Towards behavioral consistency in heterogeneous modeling scenarios." In Proceedings of the International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), October 2021, doi:10.1109/MODELS-C53483.2021.00107.

2. **Tim Kräuter**, Harald König, Adrian Rutle, Yngve Lamo. "The Visual Debugger Tool." In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), October 2022, doi.org:10.1109/ICSME55016.2022.00066.

3. **Tim Kräuter**, Adrian Rutle, Harald König, Yngve Lamo. "Formalization and analysis of BPMN using graph transformation systems." In Proceedings of the International Conference on Graph Transformation (ICGT), **Best Paper Award**, July 2023, doi:10.1007/978-3-031-36709-0_11.

4. **Tim Kräuter**, Patrick Stünkel, Adrian Rutle, Yngve Lamo. "The Visual Debugger: Past, Present, and Future." In Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments co-located with the 46th International Conference on Software Engineering, IDE'24, August 2024, doi:10.1145/3643796.3648443.

[1] A demonstration of the Visual Debugger Plugin can be found on YouTube.
[2] Last checked on 05.05.2025, see Visual Debugger Plugin.

ACKNOWLEDGMENTS

When I started my bachelor's degree at FHDW Hannover in 2015, I would have never imagined that I would want to pursue a PhD ten years later. If you had asked me after my first week of lectures, I would have said it was more likely that I would quit than finish the bachelor's, let alone try to take on a PhD[3]. I am grateful for all the unique and amazing teachers I met along the way, especially Michael Löwe and Harald König, who shaped my understanding not only of computer science but of the world. The biggest hurdle in beginning a PhD in a foreign country was leaving behind the great friends I met along the way, such as *Jan*, *Jeremy*, *Thorben*, *Alex*, *Mara*, *Mario*, *Tom*, *Max*, *Micha*, *Manni*, and many more. I am always thrilled to connect with you again when I come to Germany (or wherever you live now) to see what you have been up to. Keeping in contact over long distances is not easy, but you all deserve it.

Without my three supervisors, this PhD would not have been possible. *Adrian Rutle* consistently challenges me to improve the clarity and flow of my writing, prompting me to consider the meaning of every word I choose carefully. In addition to providing invaluable support and assistance with research, he generously invited me to Christmas dinner during the COVID-19 travel restrictions, and we also took part in a backcountry skiing course together. Thanks to *Harald König*, who had already been my professor in Germany, I first learned about this PhD project. I am grateful to him not only for initiating this journey but also for his continuous, actionable feedback and forward-looking advice. Furthermore, together with *Uwe Wolter*, he was the driving force behind the theoretical underpinning of my work using coalgebras. Last but not least, *Yngve Lamo* encouraged me to think beyond conventional approaches, consistently offered new ideas, and helped me maintain a broad perspective on my PhD, ensuring I did not become too focused on the paper I was currently working on. Over the past four years, these three individuals have reviewed numerous drafts of this thesis and its associated papers, and I am deeply grateful for their unwavering support.

The colleagues and friends around me played a significant role in completing this PhD. First, I would like to thank the senior PhDs I met when coming to HVL: *Patrick*, *Frikk*, *Anton*, *Justus*, *Håkon*, *Salah*, and *Michele*, who welcomed me with open arms after arriving in a foreign country, which can feel like restarting your life from scratch. Special thanks to Michele for organizing countless hiking, skiing, and other adventures that kept me sane while I faced many challenges and rejections during my PhD. Together with *Gerard*, *Haakon*, *Fatemeh*, *Hunter*, *Lena*, and *Aurora*, you formed the best inner circle of friends I could have hoped for. Recently, *Ulises*, *Ivan*, and *Daniel* have joined this circle, with whom the social activities are in great hands for the next generation of PhDs. Of course, I must also thank all the other PhDs and Postdocs, such as *Thanh*, *Keila*, *Daudel*, *Guy*, *Naser*, *Mira*, *Ashlesh*, *Zainul*, *Yu-Chung*, *Amir*, *Mojtaba*, and many more, for the inspiring and motivating interactions.

I believe the demanding cognitive efforts involved in pursuing a PhD must be balanced by physical activity. I want to thank the people who made these activities

[3]Please imagine I knew what a PhD was at that point.

possible. The best climbing and belaying partners: *Rodolfo*, Michele, Gerard, Lena, Ulises, Haakon, Andreas, *Nikita*, and *Håvard Skibenes*. Thanks for the fantastic time inside, outside, and competing for HVL. Thanks to Haakon, Ivan, Ulises, Daniel, *Rizwan*, and *Quasim* for the many close table tennis matches and later Ping-Tac-Pong. Haakon, I am still waiting for a rematch to get back my trophy. Furthermore, I want to thank *Bergens Turnforening*, especially *Marius* and *Harald*, for letting me train with some of the best gymnasts in Norway. Finally, I want to thank all the people at *Bergen Grappling*, especially *Knut*, for helping me forget about all the small things in life during sparring.

I want to thank Professor *Julien Deantoni* for a great time during the research stay in France at the National Institute for Research in Digital Science and Technology at Université Côte d'Azur (*Inria*: Institut national de recherche en sciences et technologies du numérique). In particular, the feature model in chapter 3 of the thesis is based on discussions and knowledge gained during my research stay while working with Julien. In addition, I would like to thank *Maksym* and the other PhD students in France for their fantastic company during my four-month research stay.

Finally, I would like to thank my best friend, partner, and family. My best friend, *Michi*, is adept at getting me off my high horse, slowing me down when needed, and helping me appreciate what I have already achieved. He remains willing to chat about anything and play video games with me, even though I have not reached Master in LoL. The most important person I met during my PhD was Marianne, who showed me the best parts about living in Norway, like properly celebrating the 17th of May. She is the kindest and most helpful person I know, and she is open to any kind of adventure or mischief that I can come up with. I am deeply grateful to my parents for their continuous support, encouragement to pursue a PhD abroad, as well as legendary care packages full of sweets and anything else I needed. I truly appreciate your visits and those from my brother and sister. I am especially grateful for always being warmly welcomed and having a place to stay with great company, whether in Lilienthal, Hannover, or Hamburg.

ABSTRACT

Over the past 50 years, computing has profoundly transformed the world, reshaping numerous industries. As a result, computer programs have permeated nearly every facet of modern life, including production, transportation, infrastructure, healthcare, science, finance, administration, defense, and entertainment. To fulfill their ever-growing requirements, software systems are becoming not only more complex individually but also increasingly interconnected. Simultaneously, their widespread success—and our resulting reliance on them—has made them vital for our safety, security, health, and overall well-being.

To handle increasing complexity, software engineers have gradually developed higher levels of abstraction, starting with transitioning from machine code to programming languages and later providing fully reusable components like databases and web servers. Model-Driven Software Engineering (MDSE) elevates the abstraction level further by introducing *models* to define systems or their most critical parts. A model is a partial representation of reality that simultaneously describes and prescribes a system's structure and behavior. A collection of models describing the structure and behavior of a software system is commonly called a *multi-model*. By leveraging the prescriptive nature of models, I aim to verify—and ideally guarantee—that a system behaves as intended before any code is deployed to production. Consequently, MDSE and the research presented in this thesis contribute to managing the complexity of software systems to meet their growing demands for reliability, security, and safety.

This thesis focuses on *behavioral models* within the multi-model that describe how different parts of a system react to inputs (user interactions or changes in the environment). Individual models represent distinct moving parts of a system that must finally interact to realize its overall behavior. Analyzing and verifying the global behavior that emerges from the interactions, i.e., the coordination of *heterogeneous* models, is an open research challenge we address in this thesis.

Our proposed coordination framework allows for the specification of the coordination of heterogeneous behavioral models in a multi-model using a domain-specific language. Then, one can formally verify if the overall system exhibits the desired behavior. Additionally, the framework facilitates data exchange between heterogeneous models and supports real-time capabilities to cover all typical system interactions. In summary, our research's main contribution is a framework that enables coordination and formal verification within a heterogeneous multi-model environment that supports data exchange and real-time functionality.

SAMMENDRAG

I løpet av de siste 50 årene har databehandling dyptgående forvandlet verden og endret en rekke industrier. Som et resultat har dataprogrammer gjennomsyret nesten alle aspekter av det moderne liv, inkludert produksjon, transport, infrastruktur, helsevesen, vitenskap, finans, administrasjon, forsvar og underholdning. For å møte stadig økende behov, blir programvaresystemer ikke bare mer komplekse individuelt, men også stadig mer sammenkoblet. Samtidig har deres utbredte suksess – og vår påfølgende avhengighet av dem – gjort dem avgjørende for vår sikkerhet, trygghet, helse og generelle velferd.

For å håndtere økende kompleksitet har programvareingeniører gradvis utviklet høyere nivåer av abstraksjon, fra overgangen fra maskinkode til programmeringsspråk, og senere til fullt gjenbrukbare komponenter som databaser og webservere. Modelldrevet programvareutvikling (Model-Driven Software Engineering, MDSE) hever abstraksjonsnivået ytterligere ved å introdusere modeller for å definere systemer eller deres mest kritiske deler. En modell er en delvis representasjon av virkeligheten som samtidig beskriver og foreskriver et systems struktur og oppførsel. En samling modeller som beskriver strukturen og oppførselen til et programvaresystem kalles ofte en multi-modell. Ved å utnytte modellers foreskrivende natur, er målet mitt å verifisere – og ideelt sett garantere – at et system oppfører seg som tiltenkt før noen kode tas i bruk i produksjon. Dermed bidrar MDSE og forskningen presentert i denne avhandlingen til å håndtere kompleksiteten i programvaresystemer for å møte deres økende krav til pålitelighet, sikkerhet og trygghet.

Denne avhandlingen fokuserer på atferdsmodeller innenfor multi-modellen som beskriver hvordan ulike deler av et system reagerer på input (brukerinteraksjoner eller endringer i omgivelsene). Individuelle modeller representerer distinkte bevegelige deler av et system som til slutt må samhandle for å realisere systemets helhetlige oppførsel. Å analysere og verifisere den globale oppførselen som oppstår fra disse interaksjonene – altså koordineringen av heterogene modeller – er en åpen forskningsutfordring vi adresserer i denne avhandlingen.

Vårt foreslåtte koordineringsrammeverk muliggjør spesifikasjon av koordinering av heterogene atferdsmodeller i en multi-modell ved hjelp av et domenespesifikt språk. Deretter kan man formelt verifisere om det overordnede systemet utviser ønsket oppførsel. I tillegg legger rammeverket til rette for datautveksling mellom heterogene modeller og støtter sanntidsfunksjonalitet for å dekke alle typiske systeminteraksjoner. Oppsummert er hovedbidraget i vår forskning et rammeverk som muliggjør koordinering og formell verifikasjon innenfor et heterogent multi-modellmiljø som støtter datautveksling og sanntidsfunksjonalitet.

Contents

Part I

OVERVIEW

"Do one thing every day that scares you."

— Eleanor Roosevelt

CHAPTER 1

INTRODUCTION

Computing has changed the world radically in the last 50 years and disrupted countless industries. Nowadays, computer programs play a crucial role in almost any aspect of modern life, such as production, transportation, infrastructure, healthcare, science, finance, administration, defense, and entertainment. Consequently, programs have become increasingly complex to fulfill their ever-growing requirements, meaning today they can have *millions of lines of code* written and maintained over decades by hundreds of people from large companies and open-source communities. At the same time, the success of computer programs has made them essential for our safety, security, health, and overall well-being [40]. For example, on July 19, 2024, a faulty software update from a security vendor caused an outage affecting millions of systems, resulting in what may be regarded as the most significant IT outage in history. This incident disrupted operations across airports, hospitals, public transit, financial services, and media outlets. In summary, past developments have made producing correct programs significantly more challenging while simultaneously amplifying their need for correctness due to our reliance on them.

To address the increasing complexity, software engineering has evolved to operate on higher and higher levels of abstraction (see Figure 1.1). As Jimmy Lin emphasizes in [109], *"Computer scientists have one incredibly powerful tool to manage complexity: abstraction."* Early software engineering went from machine code to assembler and then to structured programming, introducing control structures such as `if`, `while`, and subroutines to abstract from assembly instructions and encourage reuse. Object-oriented programming increases the level of abstraction by introducing classes that encapsulate data and functions and by implementing polymorphism to abstract the control flow. Additionally, most object-oriented languages, such as Java, C#, and Go, abstract memory management using automatic garbage collection, eliminating manual memory management. Then, software engineers build another level of abstraction on top of existing programming languages in the form of libraries and frameworks, which allow for the reuse of components and further separation of concerns on a larger scale. For example, NumPy is a Python library that implements standard math calculations for scientific computing, and Angular is a framework that simplifies building interactive websites by implementing automatic change detection, among other features. Nowadays, software engineers even reuse entire components as building blocks for their systems, such as databases, web servers, BLOB storage systems, workflow engines, caches, load-balancers, and message queues, which can

be configured to fit their needs. These components can even be run as managed services in the cloud, effectively abstracting physical hardware and even can employ tools like Kubernetes and Docker to dynamically scale the resulting systems to match user consumption. This ongoing increase in abstraction allows software engineers to keep building systems of growing complexity. In summary, to deal with rising complexity, software engineers nowadays separate a system's concerns and solve them on a high abstraction level by employing existing libraries, frameworks, or even entire components. It is crucial to ensure that the different aspects, which are solved somewhat in isolation to make them manageable, still fit together in the bigger picture to fulfill the system's goals, which is the underlying motivation of this thesis, as we will see later.

Figure 1.1 shows the increase in abstraction in software engineering. Most engineers nowadays operate on a much higher abstraction level than before. Occasionally, software engineers must delve into lower abstraction levels, such as when performance or memory consumption issues arise. However, this is typically temporary, with the default operation mode being at a higher abstraction level, at least on the level of object-oriented programming, if not higher, in Figure 1.1.

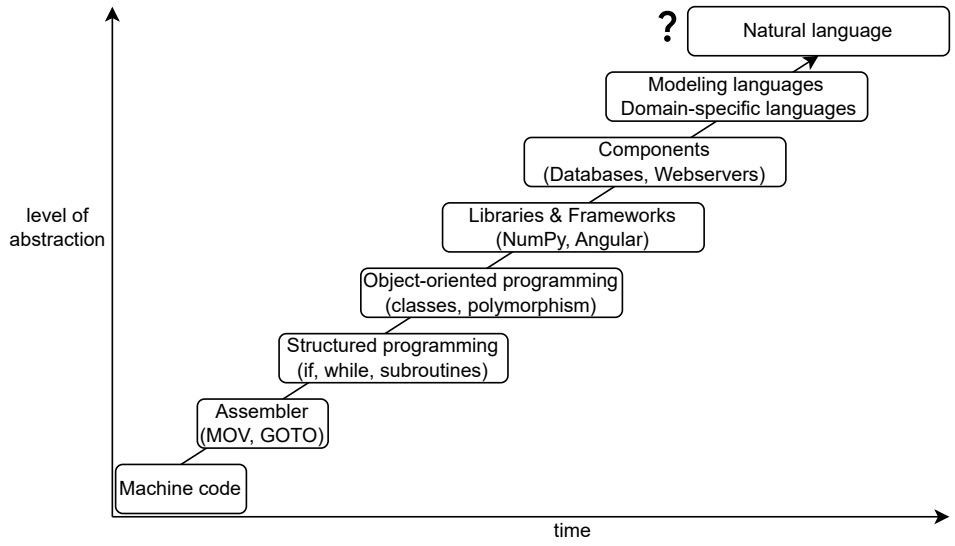


Fig. 1.1: Increasing abstraction in software engineering (adapted from [29])

Using AI, particularly large-language models (LLMs), we once again strive to elevate the abstraction level, making natural language sufficient for developing software systems. The ultimate goal is to generate code using textual prompts that leverage existing abstractions, such as libraries, frameworks, and components, to build a system that meets the requirements. In my opinion, the capabilities of current LLMs, even with their new "Agent" features, remain insufficient. Models such as GPT-4o or Claude 3.7 are primarily able to produce small, well-known applications or code blocks, which are frequently unreliable, prone to errors, and require significant energy consumption. Despite this, they can help in software engineering by searching documentation, generating code snippets, or troubleshooting issues.

In this thesis, we do not use LLMs to increase abstraction but rather build upon a different, more structured, and controlled approach. The goal is to improve software

engineering, that is, to make the production of reliable and correct systems easier despite the outlined challenges. To achieve this, we raise the abstraction level by employing *models* to describe a system or its critical parts. We define a model as a partial representation of reality to accomplish a task or convey information about a system [29]. In Model-driven Software Engineering (MDSE), the approach that we build upon, models are treated as the primary artifacts in software development rather than merely serving as a supplement to source code (if they are present at all). The main idea is to use models' prescriptive nature to verify that a system behaves as desired before a single line of code is written. This will allow us to deal with complexity by raising the abstraction level, separating concerns into different *heterogeneous* models, and verifying that the resulting system still *behaves* as expected. Our approach aims to guarantee software systems' ever-increasing reliability, security, and safety requirements. In the next section, we provide a more detailed introduction to MDSE, a subfield of software engineering, before elaborating on our approach.

1.1 Model-Driven Software Engineering

Model-Driven Software Engineering (MDSE) seeks to enhance the efficiency and effectiveness of software development [29]. In MDSE, models serve as the primary artifacts, enabling work at a higher level of abstraction. As discussed in the previous section, lifting the abstraction level is increasingly important for managing the growing complexity of software engineering. Models abstractly represent real-world objects by generalizing their features, classifying them into coherent groups, and aggregating simpler objects into more complex structures [29]. Again, in this thesis, we define a model as a partial representation of reality to accomplish a task or convey information about a system [29] (for a broader discussion on modeling history, see [172]). Thus, using MDSE simplifies development by abstracting the details of the underlying programming language, enabling developers to concentrate on the application's high-level concepts. Based on this definition, models can serve various purposes [29, 123]:

1. **Models as sketches:** Models are used for communication and documentation purposes, and only parts of a system are specified,
2. **Models as blueprints:** Models are used to provide a complete and detailed specification of the system,
3. **Models as programs:** Models, instead of code, are used to develop the system.

Models as sketches is the most common use case for models. They are often used during development and the design phase to clarify and communicate ideas and document the chosen design decisions. Sometimes, models are created after the system is implemented to understand its behavior and diagnose potential issues. Usually, such models only cover the critical parts of a system.

These models can then be completed to provide a **blueprint** of the entire system. Often, blueprints cover the system's structure, such as domain classes and database schema, but do not fully specify the system's behavior. To understand how models

function as **programs**, it is essential first to recognize the classification of models into *structural* (or static) and *behavioral* (or dynamic), see Figure 1.2 [29, 176].

Structural models describe which entities exist in a system and how they are related. Furthermore, structural models can describe the architecture or structure of the system itself. Examples of structural models are class diagrams [144], component diagrams [144], object diagrams [144], and database schemas. For example, the class diagram in Figure 1.2 describes the model taxonomy, i.e., a structure.

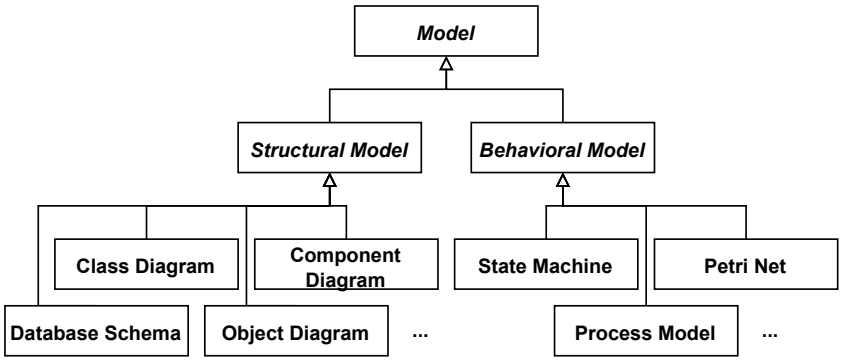


Fig. 1.2: Model taxonomy (adapted from [176])

Behavioral models represent the dynamic aspects of a system, specifying how it responds to events or generates outputs based on given inputs. These models are typically state-based, capturing a system's state and how it evolves due to environmental inputs or temporal progression. Examples of behavioral models are different types of state machines [71], Petri Nets [85, 86], and process models [79, 135, 136, 143] (see Figure 1.2).

Examples of **models as programs** can be found both for structural and behavioral models. **Structural models** are commonly used to generate code for various applications. Within MDSE research, class diagrams are frequently employed, whereas the industry often relies on textual models. Data modeling and serialization are typical industry applications of MDSE, enabling the automatic generation of code for multiple programming languages. Examples of frameworks used for data modeling and efficient serialization and deserialization of data across programming languages include Bond [132], Protocol Buffers [65], Apache Thrift [11], and Apache Avro [10]. Leading software engineering organizations, including major technology companies and non-profit organizations, back these frameworks.

Two standard methods exist to use **behavioral models as programs**. However, before exploring these ways, it is essential to highlight the advantages of behavioral models over behavior directly expressed in source code. Because models operate at a higher level of abstraction, they facilitate analysis capabilities and a clearer understanding of the system behavior. For example, our approach employs model checking (see section 2.3) to formally verify that the system adheres to its specified behavior, thereby significantly increasing confidence in its correctness. Verification is either impossible or substantially more challenging when working directly with source code.

The first method utilizes behavioral models for **code generation**. For example, one can generate code from state charts using the industrial tool itemis CREATE [82]. The generated code implements the defined state chart, enabling modelers to describe

system behavior independently of the underlying programming language and platform details. Based on my experience, code generation from behavioral models is less commonly used in the industry.

However, the second method, **model interpretation** [29], i.e., direct *model execution*, is widely used in practice, specifically in the business process management (BPM) domain. In this context, the Business Process Model and Notation (BPMN) can define executable business process models. A business process is a sequence of steps that, when executed, results in a desired outcome. The BPMN model represents these steps as activities and includes sequence flows and gateways that define the order in which the activities must occur. BPMN models can be executed on Business Process Execution Engines (BPEEs), such as Camunda [31] or Activiti [5]. A BPMN model starts as a sketch describing a key business process in a company, involving input from software engineers and business stakeholders. This input turns the model into a detailed blueprint of how the organization implements the business case. Finally, with the help of software engineers who implement individual activities and a BPEE, the BPMN models become part of the system (models as programs). This process shows the power of models for communication, documentation, detailed specification, and development, which plays a key part in the success of BPMN models in the industry.

Besides its success in BPM, MDSE is "marketed" under the name of low-code platforms in the industry. For example, OutSystems [150], Mendix [129], and the Microsoft Power Platform [133] are some industrial applications of MDSE. These systems let users build systems or automate processes using graphical interfaces based on models rather than coding. An example is illustrated in Figure 1.3, showcasing an automated workflow created in Power Automate [134]. The workflow initiates when a pull request is submitted, triggering the automatic dispatch of messages or emails to the designated reviewers. To configure this process, users arrange predefined triggers, control-flow elements, and actions with assistance from chat-based LLMs, requiring *minimal or no coding effort*. Nevertheless, MDSE has not replaced traditional coding but complemented it in specific application areas [2, 192]. For instance, as shown in Figure 1.3, software engineers are still responsible for creating the building blocks the end user utilizes.

In summary, Model-Driven Software Engineering (MDSE) describes a system by modeling its structure and behavior, captured in multiple interconnected models. Essentially, MDSE views a software system as a combination of its structural and behavioral components, as illustrated in Equation 1.1 (adapted from [194]). It is important to note that this depiction is a simplified abstraction of the intricate realities of software systems, yet it provides a valuable framework for understanding and further analysis.

$$\mathit{Structure} + \mathit{Behavior} = \mathit{Software\ System} \tag{1.1}$$

It is important to remember that the higher abstraction levels presented in Figure 1.1 neither invalidate nor completely replace the lower levels. While there is a general shift toward higher abstraction, software engineers must be able to move seamlessly both up *and* down the abstraction hierarchy depending on the specific use case. Similarly, the literature on MDSE and my experience as a software engineer in the industry

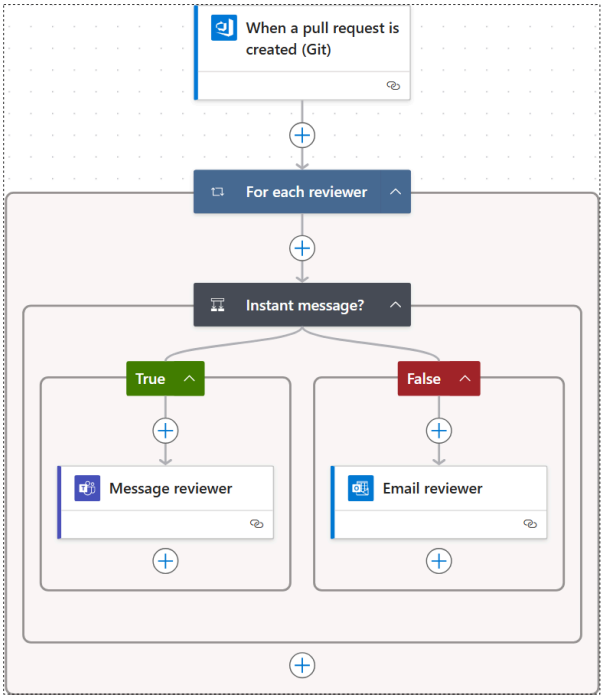


Fig. 1.3: Process in Power Automate [133, 134] to notify reviewers about a pull request

indicate that it complements rather than replaces traditional software engineering [192]. For example, the three industrial cases involving MDSE presented in [139] conclude that organizations must identify where they need MDSE and not apply it blindly. Furthermore, software engineering remains crucial for the development and ongoing maintenance of MDSE solutions [2].

1.2 Multi-Modeling

Large software projects frequently utilize multiple modeling languages to represent a software system's various domains and aspects. The resulting collection of models is referred to as a *multi-model*. A defining characteristic of a multi-model is that its constituent models are not required to conform to the same modeling language. Different modeling languages are often employed to capture various parts of the same (or different) system(s), each focusing on specific aspects. There can even be different structural or behavioral modeling languages that are used at the same time. Much research has been conducted, especially on the structural models in a multi-model, for example [27, 91, 177–179]. The literature describes how to keep information consistent that is shared between multiple structural models. Typically, one starts with *matching* [94], i.e., finding which data is shared between models. Then, *verification* checks consistency between models based on previously defined consistency rules [187]. Finally, if any inconsistencies are found, *repair* takes place, which changes the models to restore consistency [21, 114]. Each of these steps is already challenging, and putting them together into a comprehensive *model management* framework is even more difficult. Detailed surveys, meta-analyses, and discussions about model management can be found, for example, in [8, 17, 177, 179].

Model management approaches, to the best of our knowledge, focus on the

structural aspects of models [93]. In contrast, in this thesis, we focus on *behavioral* models in a multi-model context since a system is given by **both** its structure and behavior (remember Equation 1.1). Current MDSE approaches allow for the analysis of individual behavioral models, but analyzing the collective behavior of possibly multiple interacting software systems is challenging. Our approach supports analyzing the system behavior, i.e., the behavior of an entire multi-model. We call the behavior of a multi-model *global behavior*, which is given by the composition of the individual behavioral models. Since behavioral models might not be compatible, combining them can result in unexpected emergent behavior [49], similar to integration errors when combining parts of a software system. To address this, we developed a framework to define coordination among behavioral models, specify the expected global behavior, and verify that the system's global behavior aligns with these expectations. We refer to this framework as the *coordination framework*. In the following section, we describe the challenges that must be overcome to analyze a system's global behavior, i.e., build a coordination framework rather than focusing solely on individual behavioral models.

1.3 Challenges

The key challenge for coordinating and verifying behavioral models is their *heterogeneity* [70]. The heterogeneity challenge manifests across different dimensions and requires a different approach than that used for structural models. One of the key ideas in model management approaches is to homogenize structural models by interpreting them as *graph-like structures* [177, 178]. However, heterogeneity in behavioral models lies in their execution semantics, leading us to the following three challenges.

First, each behavioral modeling language in a multi-model can have its own **distinct execution semantics**, possibly including a unique way of handling data and time progression. Therefore, we need a flexible formalism capable of representing the execution semantics of various modeling languages as the foundation of our coordination framework.

Second, while there are ways to coordinate, for example, multiple Petri Nets with each other [85], coordinating heterogeneous behavioral models remains challenging due to differences in their execution semantics. Consequently, **coordinating heterogeneous models** encompassing data exchange and the consistent progression of global time is a complex task. Synchronization or data exchange—such as through a web service—is a typical interaction between systems that must be accurately modeled to describe the global system behavior.

Third, since the objective is to verify the system's global behavior—i.e., the behavior emerging from the coordination of behavioral models—we must be able to define and verify properties that span multiple models within a multi-model. Additionally, if any of these properties are violated, visualization in the original modeling languages is essential to diagnose and resolve the issues. In an ideal scenario, we would also suggest fixes to address the identified issues, similar to model repair techniques for structural models.

1.4 Research Questions

The previously discussed challenges highlight that coordinating and verifying heterogeneous behavioral models remains a largely unsolved challenge in MDSE. To guide the research in this PhD project, I identified and explored the following research questions to build a coordination framework that tackles these challenges:

RQ 1 How can we specify *coordination* between heterogeneous behavioral models, including real-time features and data exchange?

RQ 2 How can we *specify global behavioral requirements*, i.e., properties spanning heterogeneous models?

RQ 3 How can we *verify global behavior* of heterogeneous models, including real-time features and data exchange?

RQ 4 How can we *present violated properties* to a user and *suggest fixes*?

These four research questions (RQs) are best explained along our high-level coordination and verification process as highlighted in Figure 1.4. **RQ 1** focuses on defining the *global system model* by combining different models and specifying how and when these models interact, i.e., coordinate[1]. This leads to the initial step in the process of coordination and verification.

In the second step, the goal is to define the requirements for the system's global behavior, meaning the *global properties* that the system must satisfy. This step originated from **RQ 2**.

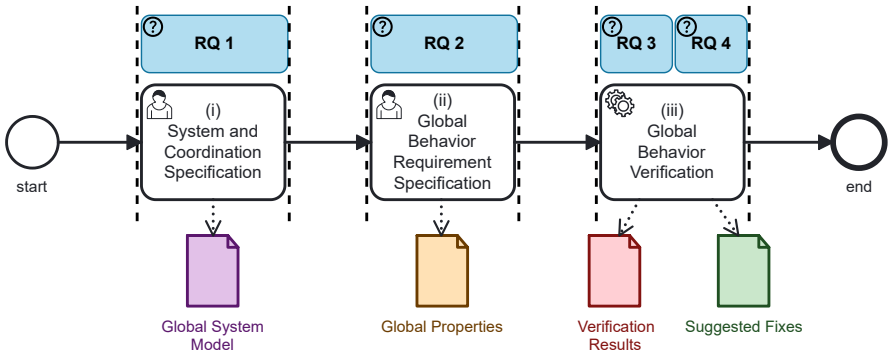


Fig. 1.4: Research questions & the coordination and verification process

Finally, the global properties are automatically checked in the third step, leading to *verification results*. In an ideal scenario, this step would also *suggest fixes* if a property is unsatisfied. Fixes are similar to model repair for structural models. The third step is about verification, i.e., directly addressing **RQ 3**, while **RQ 4** focuses on presenting these violations to the user and how potential solutions can be identified and proposed. In the conclusion of the thesis, we revisit the research questions and explain how our contributions address them.

[1]We use the terminology of interaction (Paper B) and coordination (Paper D) interchangeably, meaning how multiple heterogeneous models are composed together into a global behavior.

Our coordination and verification steps align well with the multi-model consistency management process used for structural models [177, 178]. Instead of identifying overlapping information and maintaining consistency, we identify behavioral interactions—i.e., coordination—and verify that these interactions produce the intended global behavior (behavioral consistency).

1.5 Contributions

The main contribution of this thesis is the coordination framework, which implements the previously defined coordination and verification process shown in Figure 1.4. Given a multi-model, our framework can coordinate the contained behavioral models, leading to a global system model. Concretely, we employ a domain-specific language (DSL) to define how the actions in the individual behavioral models are coordinated. Then, one defines global properties, i.e., the behavior requirements for the global system, by combining propositions originating from individual behavioral models. Finally, the framework can automatically verify the defined properties for the global system model. This outputs verification results, i.e., whether each property is fulfilled, including additional information if a property is not fulfilled (counterexamples). One of our framework's main innovations is its ability to specify and verify global properties in heterogeneous scenarios. In contrast, other approaches typically only allow for the specification and verification of properties in homogeneous scenarios or specific combinations of behavioral models [99]. Furthermore, the framework has other unique features, such as being *non-intrusive* and *parametrizable* by specific formalisms (such as graph transformation and rewriting logic). In addition, it supports *data exchange* and *real-time capabilities* [101].

Ideally, one would visualize these counterexamples and suggest fixes that resolve the found violations. However, similar to model repair for structural models, this is a highly challenging task. We tackle these challenges in a narrower scope, meaning limiting ourselves to one behavioral language (BPMN). Doing so demonstrates that our framework can work for complex behavioral languages like BPMN. In addition, we provide concrete tools for BPMN, emphasizing promising ideas and the future potential of our coordination framework by demonstrating valuable insights gained from analyzing a single language [100]. We show how to define BPMN properties, visualize counterexamples when these properties are violated, and suggest fixes in specific scenarios [102].

In addition, we contribute to the area of Coalgebras, which provides a theoretical foundation for behavioral models. We generalize the coordination (structural operational semantics rules) of behavior expressed as coalgebras to work with heterogeneously typed coalgebras, enabling the theory to be used for heterogeneous systems [95]. We use the coalgebraic paradigm for behavior, i.e., thinking about state and state-changes, to guide our research. Similar to approaches for structural models that interpret each model as something akin to a graph, we interpret behavioral models as describing a set of states and state-changing elements. Moreover, specification of global properties based on coalgebras is well-established [103]. While Coalgebras is an elegant theory, we use less abstract formalisms for the main contributions of this

thesis since they come with well-known state-of-the-art tools[2]. Nevertheless, these formalisms can be regarded as specific instances of the coalgebraic formalism.

1.6 Example of coordinating behavioral Systems

As a concrete example of a multi-model of behavioral systems that must be coordinated, we show the use case from Paper D in Figure 1.5. The use case illustrates a traffic management system for a road-rail crossing composed of three subsystems: a sensor system, a barrier system, and a crossing manager.

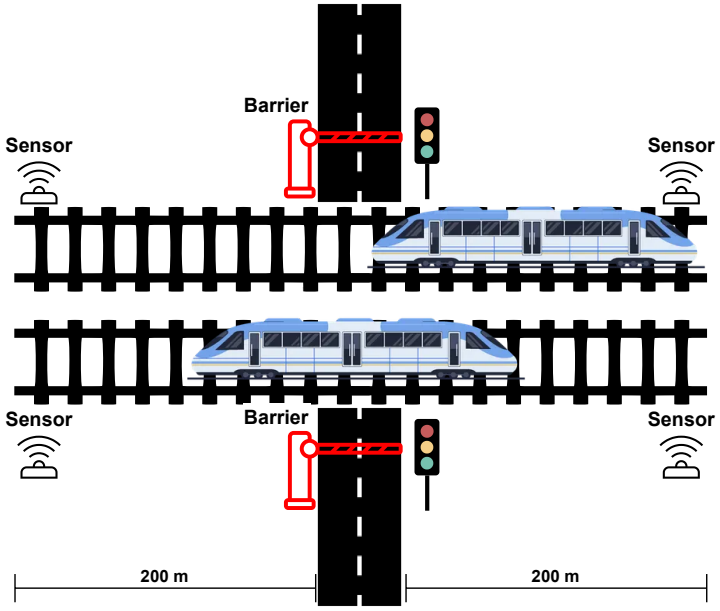


Fig. 1.5: Traffic management system (see Paper D)

The objective is to ensure that cars and trains always pass safely through the crossing, meaning there is no risk of collision on the train tracks. In this scenario, the multi-model consists of two state charts and a colored Petri Net. Furthermore, data exchange between the models and timed periodic actions pose additional challenges. We describe how our framework can verify the behavior of the traffic management system in Paper D. Other approaches have difficulty addressing this scenario due to the heterogeneity of behavioral models. They typically evaluate only individual models or homogeneous scenarios, which prevents them from assessing the traffic management system's global behavior, including how its different components are integrated, i.e., coordinated.

1.7 Research Methodology

In this research, I adopt the *design science* methodology, which is well-suited for applied disciplines such as software engineering and widely recognized in the research

[2]We used Maude for rewriting logic and Groove for graph transformations where state-changing elements are rewrite-rules and states are terms or graphs, respectively.

community. Design science offers several key benefits, including its problem-solving focus, structured methodology, iterative refinement process, and balanced emphasis on theory and practical application [76]. I will introduce design science in the following subsections and then explain its application within my research project.

1.7.1 Design Science

Design science research addresses challenges in an application domain by developing *artifacts* that capture new knowledge grounded in an existing knowledge base [76, 77, 120, 193]. Artifacts are categorized into four types: **(1)** constructs, **(2)** models, **(3)** methods, and **(4)** instantiations [120, 131]. **First**, *constructs* provide a language of concepts to characterize phenomena. **Second**, these can be combined in higher-order constructions, i.e., *models* to describe tasks, situations, or artifacts. **Third**, *methods* are developed to perform activities with a specific goal. **Finally**, the preceding can be *instantiated*, resulting in products implementing certain tasks. Notably, design scientists focus not on formulating theories as natural scientists do but on creating innovative and valuable artifacts [120]. The design science research process is best explained by its three closely related cycles of activities, see Figure 1.6 [76].

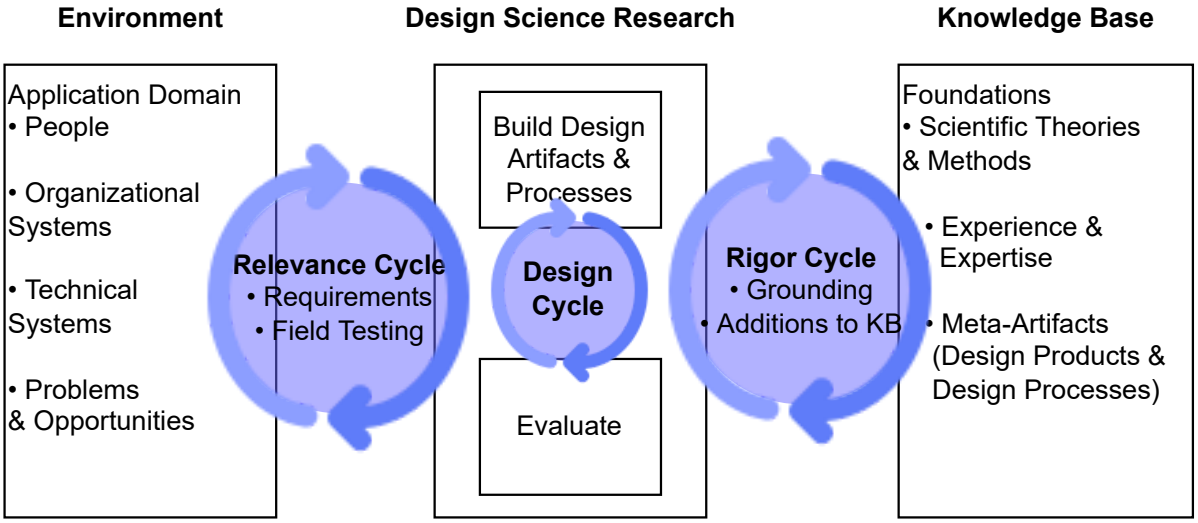


Fig. 1.6: Design Science Research Cycles [76]

Relevance Cycle: Design science focuses on solving problems within a specific application domain. The domain includes people, organizational, and technical systems that work together to achieve specific goals. When issues or opportunities arise in the application domain, they trigger the relevance cycle by generating requirements for design science research. Moreover, the domain also provides the criteria for accepting these requirements. Consequently, artifacts developed in design science research are ultimately tested and evaluated in the application domain. The outcome of this field testing determines whether another iteration of the relevance cycle is necessary, typically when the artifacts do not meet the original requirements or need further refinement [76].

Rigor Cycle: Design science research builds upon an extensive knowledge base of scientific theories and engineering methods rather than starting from scratch. This

knowledge base also includes practical experience, expertise, design artifacts, and design processes that represent the state-of-the-art within the application domain. The state-of-the-art is a benchmark for evaluating newly developed design artifacts and processes. Most importantly, design science research adds new knowledge to the knowledge base. This new knowledge may take several forms, including improvements or adaptations of current artifacts, the creation of entirely new artifacts, and the insights gained from field testing artifacts within their application domain. The additions made to the knowledge base during the rigor cycle support and inspire future research [76].

Design Cycle: The design cycle is at the core of design science research. In this process, artifacts are created, evaluated, and refined based on evaluation feedback. These evaluations are guided by the requirements provided through the relevance cycle and the latest experience, theories, and methods from the rigor cycle, i.e., the existing knowledge base. The process involves rapid iterations between artifact creation and evaluation, exploring multiple alternative artifacts. However, there must be a balance between the time and effort dedicated to developing and evaluating these evolving artifacts [76].

1.7.2 *Design Science in this PhD project*

My papers present *artifacts* developed through multiple design cycle iterations, as summarized in Figure 1.7. In the following paragraphs, we will summarize these artifacts, categorize them according to the types outlined by [120], and explain how they were shaped by the rigor and relevance cycle.

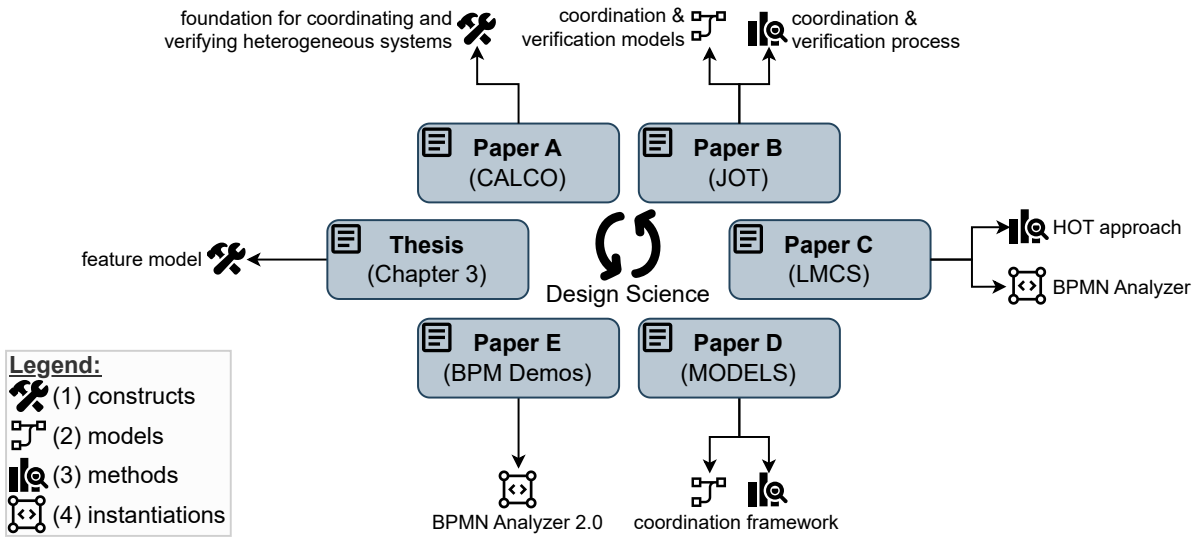


Fig. 1.7: Overview of the created artifacts

Paper A [95] anchors my research in the established theory of coalgebras (rigor). It further expands this theory to address the need for coordinating *heterogeneous* behavioral systems, making a meaningful contribution to the existing body of knowledge. We generalize the coordination (structural operational semantics (SOS) rules) of behavior expressed as coalgebras to work with heterogeneously typed coalgebras. The coalgebraic paradigm, which interprets behavioral models as states and state-changing elements, lays the foundation for coordinating and verifying heterogeneous systems.

Paper B [99] develops *models* and *methods* for coordinating and verifying heterogeneous systems. We introduce an initial coordination and verification process, along with the corresponding models, that are the foundation for the coordination and verification process given in this thesis (Figure 1.4). Furthermore, we benchmark and evaluate the approach for a traffic management system as a use case.

Paper C [100] describes a new approach (*method*) to formalize execution semantics based on higher-order model transformations and applies it to BPMN with the goal of formal control-flow analysis. Using this approach, we can analyze BPMN models that incorporate nearly all features used in the industry, a crucial requirement stemming from the relevance cycle [52]. We provide an *instantiation* [120] in an open-source tool to evaluate its scalability and compare it to other tools. The tool is available online [181] alongside a demonstration video [182].

Paper D [101] introduces a coordination framework (*method*) aimed at facilitating the coordination and verification of heterogeneous systems, particularly in contexts involving **data exchange** and **real-time** constraints. To do so, the coordination framework introduces new *models* and builds upon previous artifacts (rigor), for example, from Paper B [99]. Furthermore, we implement the framework to evaluate it by modeling a railroad crossing as a use case.

Paper E [102] describes the BPMN Analyzer 2.0 tool. You can watch a demonstration of the BPMN Analyzer 2.0 [184] and try it online [183], which is an *instantiation* according to the categorization of [120]. The paper focuses on the relevance cycle, since it meets previously identified industry requirements [52].

In addition to the papers, chapter 3 introduces a feature model to compare different coordination approaches. The feature model is an artifact of type *construct* [120] since it provides a common language for characterizing coordination approaches. Moreover, applying this feature model to my research and other approaches helps anchor the work within the existing body of knowledge by offering a concrete basis for comparison (rigor).

1.8 Thesis Outline

The thesis is structured into two main parts. In Part I, I situate my research within the broader context of related work in software engineering and provide the necessary background information to ensure the thesis is self-contained. Part II includes five papers that constitute the core contribution of this work. Besides this introduction, Part I consists of the following chapters:

Theoretical background: This chapter presents an overview of the theoretical foundations that form the basis of our contributions. We start by describing the coordination of behavioral systems using transition systems as a simple example. This naturally leads to an informal introduction to Coalgebras, a theory that aims to unify the representation of software system behaviors beyond transition systems. Next, we provide an overview of the fundamental concepts of model checking, which are essential for verifying the behavior of individual behavioral models and systems composed of multiple interacting behavioral models, as examined in this thesis. Finally, we present the key concepts of rewriting logic and graph transformation, illustrating

them with concrete examples from the tools Maude and Groove, which we use in our contributions.

State of the Art: In this chapter, we offer a detailed overview of the research landscape and introduce a framework for categorizing and comparing different approaches to system coordination. To build this framework, we perform a *meta-analysis* of existing literature studies and develop a *feature model* that highlights the scope, characteristics, and underlying motivations of different coordination approaches. Finally, we present the resulting feature model, use it to classify various coordination approaches, and analyze trends in the coordination domain.

Related Work: In this chapter, we present related work using a running example that models a pedestrian crossing with traffic lights. We demonstrate how this example can be implemented using a representative from each category of coordination approaches.

Conclusion: In this last chapter, we revisit the research questions outlined earlier and summarize our key contributions. We discuss the evaluation and limitations of our work and suggest possible directions for future research. Finally, we conclude the first part of the thesis with some closing remarks.

"The happiness of your life depends upon the quality of your thoughts."

— Marcus Aurelius

CHAPTER 2

THEORETICAL BACKGROUND

This chapter establishes the theoretical groundwork for the thesis. **First**, we present the coordination of behavioral systems through Structural Operational Semantics (SOS) rules applied to transition systems, as a simple example (see section 2.1). This discussion naturally progresses to Coalgebras, which we informally introduce in section 2.2, as they offer the most general and abstract formalism capable of modeling heterogeneous scenarios, extending beyond transition systems. Coalgebras provide a general paradigm to reason about behavioral models and a formal underpinning for coordination (see Paper A).

Then, we introduce model checking in section 2.3, a formal method for verifying a system's behavior, which we use to implement the coordination and verification process, i.e., provide global behavior verification of heterogeneous systems. A thorough understanding of model checking is essential for this thesis.

Finally, we present an introduction to rewriting logic and Maude in section 2.4. In addition, we provide an introduction to graph rewriting rules and Groove in section 2.5 using the same example. Rewriting logic and graph transformation are well-established high-level formalisms with robust tooling that we use to implement our coordination framework in Paper B and Paper D, as well as the BPMN Analyzer in Paper C.

2.1 Coordinating behavioral Systems

In this thesis, we examine multiple systems that must collaborate to achieve a shared global objective through what we refer to interchangeably as *interaction* or *coordination*. For example, the road-rail crossing in section 1.6 comprises three subsystems that must coordinate to ensure safe traffic at the crossing[1].

Following the MDSE approach, we assume each system is specified in a behavioral model. One of the simplest, well-known behavioral models is transition systems (TS), described in Definition 1.

[1]We initially used the term *interactions* in Paper B but later adopted *coordination* in Paper D, which we use throughout the remainder of the thesis.

Definition 1: Transition System (with actions)

A transition system is a tuple $TS = (S, Act, I, \rightarrow)$, where

- S is a set of **states**,
- Act is a set of **actions**,
- I is the set of **initial states**,
- $\rightarrow \subseteq S \times Act \times S$ is a **transition relation**.

We write $s_1 \xrightarrow{\alpha} s_2$ for $(s1, \alpha, s2) \in \rightarrow$. [18, 39]

We use transition systems as an example to introduce the coordination of behavioral systems, which can be formally specified using Structural Operational Semantics (SOS) [3]. Equation 2.1 presents the general notation used in SOS. In this format, if the premise stated above the fraction bar is satisfied, then the conclusion below the bar follows. This form of notation is called an inference rule, or simply a rule [18].

$$\frac{\text{premise}}{\text{conclusion}} \tag{2.1}$$

Using SOS rules, the coordination ($\|$) of two transition systems can be defined based on a set $Sync$ of synchronization actions, as described in Definition 2. The SOS rules in Definition 2 define synchronization (also known as hand-shaking) for all actions $\alpha \in Sync$, meaning that both transition systems must update their states simultaneously in a single atomic step. Actions $\alpha \notin Sync$, on the other hand, are allowed to interleave freely in any order.

Definition 2: Transition System Coordination (Synchronous)

Let $TS_i = (S_i, Act_i, I_i, \rightarrow_i)$, $i = 1, 2$, be transition systems, and let $Sync \subseteq Act_1 \cap Act_2$. The transition system $TS_1 \parallel TS_2$ is defined as follows:

$$TS_1 \parallel TS_2 = (S_1 \times S_2, Act_1 \cup Act_2, I_1 \times I_2, \rightarrow)$$

where the transition relation $\rightarrow$ is defined as:

- *interleaving* for $\alpha \notin Sync$:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2)} \qquad \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{(s_1, s_2) \xrightarrow{\alpha} (s_1, s_2')}$$

- *synchronization* for $\alpha \in Sync$:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \ \wedge \ s_2 \xrightarrow{\alpha}_2 s_2'}{(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')}$$

[18, Sec 2.2.3]

With synchronous communication as a primitive (as in Definition 2), one can define asynchronous communication and vice versa [4]. Typically, one defines asynchronous

coordination using intermediate entities, which are usually called buffers, queues, or channels [4]. Both synchronous and asynchronous coordination are required in realistic coordination scenarios.

In this thesis, we define domain-specific languages (DSLs) to specify the coordination of systems. Compared to the simple Definition 2, our DSLs are more flexible since they work for more than two systems and provide a name mapping, i.e., do not require identical action names across systems ($Sync \subseteq Act_1 \cap Act_2$). For instance, Paper D employs a DSL to specify asynchronous coordination. In contrast, Paper B (refer to Example 1) introduces a DSL designed to synchronize the actions of four systems for managing traffic at a T-Junction (see Figure 2.1).

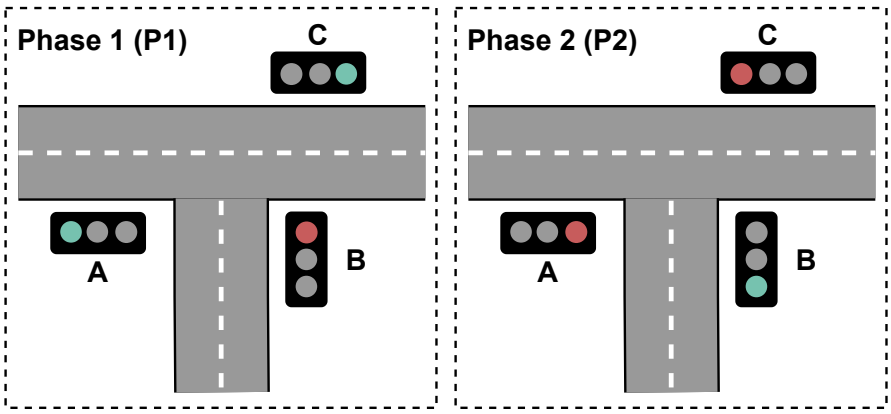


Fig. 2.1: Traffic phases of a T-Junction [99]

Example 1: DSL coordinating four Components (T-Junction)

```
// The T-Junction switches to Phase 1.
// This turns traffic lights A & C green, while B turns red.
synchronize(TJunction.Switch_to_P1,
            TJunction.A.turn_green,
            TJunction.B.turn_red,
            TJunction.C.turn_green)

// The T-Junction switches to Phase 2
// This turns traffic lights A & C red, while B turns green.
synchronize(TJunction.Switch_to_P2,
            TJunction.A.turn_red,
            TJunction.B.turn_green,
            TJunction.C.turn_red)
```

[99]

We use DSLs instead of formal SOS rules because MDSE practitioners may not be familiar with SOS. More importantly, DSLs help preserve the appropriate level of abstraction and narrow the gap between our coordination specifications and the modeling languages. From a theoretical standpoint, however, our coordination DSLs effectively serve as a more convenient method for specifying the SOS rules that

govern the interactions between the behavioral systems, simultaneously specifying the necessary name mapping.

In addition, DSLs assist in managing the heterogeneity of systems that need to be coordinated, as Definition 2 is limited to coordinating transition systems. Because we must address heterogeneous scenarios involving different formalisms, and potentially incorporating data and time aspects, we must move beyond the simple formalism of transition systems. Instead, we shift to coalgebras, as introduced in the next section, which aim to provide a unified approach to specifying behavioral systems.

2.2 Coalgebras

In this section, we introduce coalgebras informally to contextualize the theoretical contributions in Paper A. The coordination of heterogeneous coalgebras, i.e., heterogeneous systems using SOS rules, is the formal underpinning for the coordination DSLs in Paper B and Paper D that govern the coordination of the different heterogeneous systems. However, SOS rules are historically tailored to interact with homogeneously typed parts, such as the two transition systems in (2). To enable heterogeneity, one must find common ground for behavioral systems in general.

The field of coalgebra has emerged in theoretical computer science with the claim to be a unifying framework to express the behavior of computer systems [84]. Yet, coalgebras are surprisingly simple, only consisting of a set of states, denoted by X, and a structure map of the form $X \xrightarrow{\alpha} \mathcal{B}(X)$. Here, $\mathcal{B}$ describes some expression involving X, the possible outcomes or transformations that can result from applying the structure map to a state. Therefore, the structure map $X \xrightarrow{\alpha} \mathcal{B}(X)$ captures the system's behavior by specifying how each state in X changes as expressed by α [84].

The set X is usually called *set of states* or *state space*, which the coalgebra acts on. Furthermore, the function α is sometimes called *transition function*. Using these two ingredients, coalgebras can generally describe state-based systems[2] with their behavior given by the function α. Given a state $x \in X$, the result $\alpha(x)$ specifies the successor states of x if any exist. One can then encode different state-based systems by changing the structure map, including its codomain $\mathcal{B}(X)$ [84]. For example, Figure 2.2 shows deterministic and nondeterministic automata, which can be expressed as coalgebras.

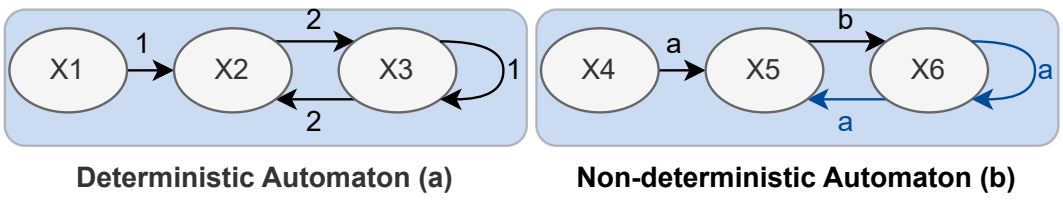


Fig. 2.2: Automata Examples

Example 2 shows how deterministic automata are encoded in the coalgebraic framework, while Example 3 defines nondeterministic automata [84, 168]. It shows the simplicity of coalgebras, which manages to express different behavioral formalisms by adjusting $\mathcal{B}(X)$ accordingly.

[2]This thesis focuses on software systems with discrete state spaces, meaning they are state-based.

Example 2: Deterministic Automata

We define $\mathcal{B}(X) = X^A$, to obtain the coalgebra $X \xrightarrow{\alpha} X^A$. A is a fixed set of input symbols (alphabet), and X^A denotes partial functions from A to X.
This coalgebra encodes *deterministic* automata [84]. For example the automaton in Figure 2.2 (a), with alphabet $A = \{1, 2\}$, can be encoded as follows:
The state space is $X = \{X1, X2, X3\}$ and the transition function $\alpha : X \to X^A$ is

$$\begin{aligned} \alpha(X1)(1) &= \alpha(X3)(2) = X2 \\ \alpha(X2)(2) &= \alpha(X3)(1) = X3 \end{aligned}$$

$\alpha(X1)(1)$ means function composition, i.e., the argument 1 is applied to the result of $\alpha(X1)$.

Example 3: Nondeterministic Automata

We define $\mathcal{B}(X) = \mathcal{P}(X)^A$, to obtain the coalgebra $X \xrightarrow{\alpha} \mathcal{P}(X)^A$. A is a fixed set of input symbols (alphabet), and $\mathcal{P}(X)$ is the power set of X.
This coalgebra encodes *non-deterministic* automata [84]. For example the automaton in Figure 2.2 (b), with alphabet $A = \{a, b\}$, can be encoded as follows:
The state space is $X = \{X4, X5, X6\}$ and the transition function $\alpha : X \to \mathcal{P}(X)^A$ is

$$\begin{aligned} \alpha(X4)(a) &= \{X5\} \\ \alpha(X5)(b) &= \{X6\} \\ \alpha(X6)(a) &= \{X5, X6\} \\ \alpha(X4)(b) = \alpha(X5)(a) &= \alpha(X6)(b) = \emptyset. \end{aligned}$$

$\alpha(X6)(a) = \{X5, X6\}$ shows that the automata is nondeterministic (blue arrows in Figure 2.2 (b)), which is possible due to using the powerset in the codomain.

Representing state-based systems in a general way is very useful, as it can be used to express heterogeneous systems (see Figure 1.4 (I)). However, we also need to describe how systems interact with each other within the coalgebraic framework. The bialgebraic framework that combines algebras (structure) and coalgebras (behavior) is the main abstract approach for SOS [92]. SOS rules are formalized as *distributive laws* if the rules are in specific formats, such as GSOS. A distributive law can be identified with a commutative diagram as in (2.2) where Σ is the syntax assignment, μ assigns to a coalgebra its state space, and ϕ is the assignment of a single behavior to a new behavior according to SOS rules [92]. We will not go into the details here, but interested readers can find a comprehensive explanation and specific examples in [92].

$$\begin{array}{ccc} \mathcal{B}\text{-}Coalg & \xrightarrow{\Phi} & \mathcal{B}\text{-}Coalg \\ {\scriptstyle\mu}\downarrow & = & \downarrow{\scriptstyle\mu} \\ Set & \xrightarrow[\Sigma]{} & Set \end{array} \tag{2.2}$$

Distributive laws guarantee that global behavior is determined by local behavior [92]. This allows for reasoning about the global system by reasoning about local components [73] and guarantees that a behavior-preserving change of the local components does not change the behavior of the global system.

However, SOS rules and, thus, compositionality only work for a single fixed $\mathcal{B}$, which is insufficient for a heterogeneous setting. For example, the distributive law in (2.2) can be used to combine coalgebras of type B into a single coalgebra of the same type (like transition system coordination in Definition 2). However, this is insufficient for heterogeneous systems, which require handling different types of behavior.

In Paper A, we introduce a formal structure for describing the composition (SOS rules) of *heterogeneously typed* coalgebras, where we prove that compositionality still holds. This theoretical work addresses **RQ 1** and provides the basis for **RQ 2** (see Figure 1.4).

The field of coalgebras is still in its early stages [84]. Most well-established practical tools we have worked with focus on more concrete formalisms, such as *term-rewriting* and *graph-transformation*. However, the coalgebraic paradigm, which interprets behavioral models as states (X) and state-changing elements (α), works well to understand heterogeneous behavioral models and abstracts the concrete formalisms we have used. Furthermore, coalgebras provide a foundation for verification. In this thesis, we make use of Linear Temporal Logic (LTL), which is one of the coalgebraic logics described in [103] and will be discussed in more detail in the following section. Some approaches even extend the coalgebraic theory to incorporate data exchange [57] and real-time aspects [89], which we handle in Paper D using rewriting logic.

2.3 Model Checking

In this section, we will briefly explain model checking. After a motivation in subsection 2.3.1, we give an overview of the basic model checking methodology in subsection 2.3.2 and introduce the necessary theoretical definitions in subsection 2.3.3, before giving a concrete example in subsection 2.3.4. Model checking is crucial in our approach because we use it to verify the behavior of heterogeneous systems.

2.3.1 *Motivation*

Model checking is a computer-aided verification method for *dynamic* systems that can be modeled using state-transition systems, i.e., finite-state systems such as hardware or finite-state abstractions of software. Nowadays, model checking is widely used for verifying hardware and software systems [40]. Ideally, one would like to develop a program together with a mathematical proof that proves the program's correctness. However, in practice, software engineers resort to static analysis and automated/manual testing, which may discover bugs but cannot guarantee their absence [47]. Unfortunately, mathematical proofs of correctness do not scale to real-world software systems due to their complex data structures, the inclusion of lower-level system code and libraries, and the integration of multiple systems [40].

In contrast, model checking of software systems has been applied to safety-critical and distributed systems. Both application areas share a high-stakes nature, meaning

that any error, even if minimal, can lead to disastrous outcomes. For safety-critical systems [28], for example aviation [26], railways [1], and spaceflight [121], errors can directly impact human well-being. Similarly, errors in core distributed systems protocols can lead to system unavailability or data loss, which damages customers' trust and leads to substantial financial losses. For example, model checking is used at Amazon Web Services (AWS) [141] to verify components of key offerings such as S3 (Blob storage) and DynamoDB (NoSQL database), as well as at Microsoft [196] to verify the correctness of a Paxos implementation which is running in production [196]. AWS is currently working towards *unbounded* model checking at the *implementation level*, i.e., code level, without the need for an additional formal language [116].

2.3.2 *Methodology*

Figure 2.3 depicts the basic model checking methodology [40]. First, one must create a *system model*, usually in a formal language, which describes the system's behavior. Since the system model is defined formally, it can be compiled to a *Kripke structure K* by a corresponding model-checking tool. A Kripke structure specifies every state within a system, the transitions among these states, and the propositions that hold at each state. In the next section, we give a formal definition of Kripke structures. In addition, one must translate informal system requirements to a *temporal-logic formula* φ, which together with K is input to the model checker.

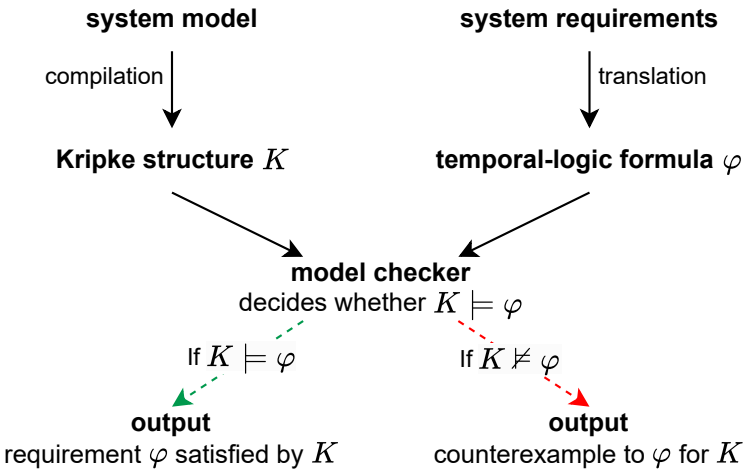


Fig. 2.3: Basic model checking methodology [40]

The model checker decides whether $K \models \varphi$, i.e., if the system model fulfills the system requirement. If the system model does not fulfill the system requirement ($K \not\models \varphi$), a *counterexample*, i.e., a trace demonstrating the violation, is given. Usually, the counterexample leads to a change in the system model to fix the violation. However, errors can also occur in the temporal-logic formula introduced during translation from the system requirements. After changing the system model or fixing the formula, one can repeat these steps until the model checker guarantees satisfaction of the requirements ($K \models \varphi$). In the next section, we will introduce the basic model checking theory (*Kripke Structures* and *temporal logic*).

Modern model checking does not always strictly follow the basic methodology shown in Figure 2.3. For example, *on the fly* model checking improves performance by constructing a Kripke structure continuously while checking a temporal-logic formula [40] since often counterexamples can be found without compiling the entire system model.

2.3.3 *Theory*

In this section, we introduce the basic theory behind model-checking, i.e., Kripke structures and Linear Temporal Logic (LTL). We focus on the intuition but also state the most important formal definitions. Then, the next section will contain an example application of the theory and the previous model-checking methodology (see Figure 2.3).

First, Kripke structures are directed graphs where nodes are labeled with atomic propositions. They are formally defined in Definition 3. Nodes and edges are called *states* and *transitions*, respectively. One can also describe them as state-labeled transition systems [40].

Kripke structures can also be described by $\mathcal{B}(S) = \mathcal{P}(S) \times \mathcal{P}(AP)$ to obtain the coalgebra $S \xrightarrow{\alpha} \mathcal{P}(S) \times \mathcal{P}(AP)$. The coalgebra combines the transition relation $\rightarrow$ and labeling function L in α.

Definition 3: Kripke Structure

A Kripke structure over a set of atomic propositions **AP** is a tuple $K = (S, S_0, \rightarrow, L)$, where

- S is a set of **states**,
- $S_0 \subseteq S$ is a set of **initial states**,
- $\rightarrow \subseteq S \times S$ is a **transition relation**, and
- $L : S \rightarrow \mathcal{P}(AP)$ is a **labeling function** associating each state with the set of atomic propositions, which are valid in that state.

[24, 39]

The atomic propositions **AP** are defined in advance and represent the facts about the system one is interested in, while other details are kept abstract in a state. For example, atomic propositions can be created for each defined constraint in a database to state whether the database fulfills these constraints. Then, for a given state $s \in S$ (instance of the database), the labeling function $L(s)$ describes which propositions (constraints) are fulfilled. For example, $L(s) = \{\texttt{foreignKey}\}$ describes that only the foreign key constraint is fulfilled, while $L(s) = \emptyset$ means no constraints are fulfilled.

Now that we know what a Kripke Structure is, we must understand how to obtain one from a system's behavioral model. Typically, model-checking tools explore the state space, generating a Kripke structure based on the given behavioral model. Intuitively, generation starts from the behavioral model's initial states ($\forall s_0 \in S_0$). For example, a behavioral model describing a database begins from an empty state or a state representing the last checkpoint. Then, the system can evolve according to its

specific rules. In our database example, a row is inserted, updated, or deleted from the database, leading to a new state $s \in S$. Transitions between states are recorded in the transition relation $\rightarrow$. Each state can fulfill a different set of atomic propositions, which is represented by L. For example, a foreign key constraint might be invalidated due to a deletion in the database.

Second, we will describe LTL by formally defining its formulas and giving an intuitive description of its semantics. Definition 4 describes how LTL formulas are built. We focus on pure future LTL formulas (i.e., no temporal operators for past states) since they are most common in practical applications. In addition, any past formula can be converted to a pure future formula [40].

Other temporal logics exist besides LTL, such as CTL and CTL*. Detailed information about CTL* and the intricate differences between CTL and LTL can be found in [40]. In addition, coalgebras, which elegantly describe behavioral systems, can be combined with temporal logics [83, 103].

Definition 4: LTL formulas

LTL formulas are built using standard boolean operators along with the temporal operators *next* ($\bigcirc$) and *until* (U). An LTL formula φ where $a \in AP$ is defined as:

$$\varphi ::= \mathsf{true} \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2$$

[18, 39]

The boolean operators $\wedge$ and $\neg$ are defined as usual, but the intuition behind the temporal operators is the following. The *next* operator ($\bigcirc$) indicates that the rest of the formula should be evaluated for the rest of the sequence in a path. The *until* operator (U) indicates that its first operand holds for a given path until some future point where its second operand holds [40].

The set of boolean operators $\{\wedge, \neg\}$ used in Definition 4 is *functionally complete*, meaning we can express all other boolean connectives $\{\vee, \rightarrow, \leftrightarrow\}$. Furthermore, we define the two standard abbreviations, eventually and globally, as shown in Definition 5.

Definition 5: Additional LTL operators

The temporal operators *eventually* ($\Diamond$) and *globally* ($\Box$) are defined as:

$$\Diamond \varphi \equiv \mathsf{true}\,\mathsf{U}\,\varphi$$
$$\Box \varphi \equiv \neg \Diamond \neg \varphi$$

[18, 39]

The boolean operators have their usual definitions, see [18]. The intuition behind *eventually* is that a formula is fulfilled at least once in the future, while globally means that a formula must always be fulfilled. Figure 2.4 summarizes the intuition behind the LTL operators. A formal definition of the LTL operators can be found in [39].

A key distinction in LTL formulas is between *safety* and *liveness*. Intuitively, a safety property ensures that something bad will never happen, while a liveness property guarantees that something good will eventually occur in the system. Algorithmically, checking safety properties is much easier than checking liveness properties. In practice, safety specifications are the most encountered form [40].

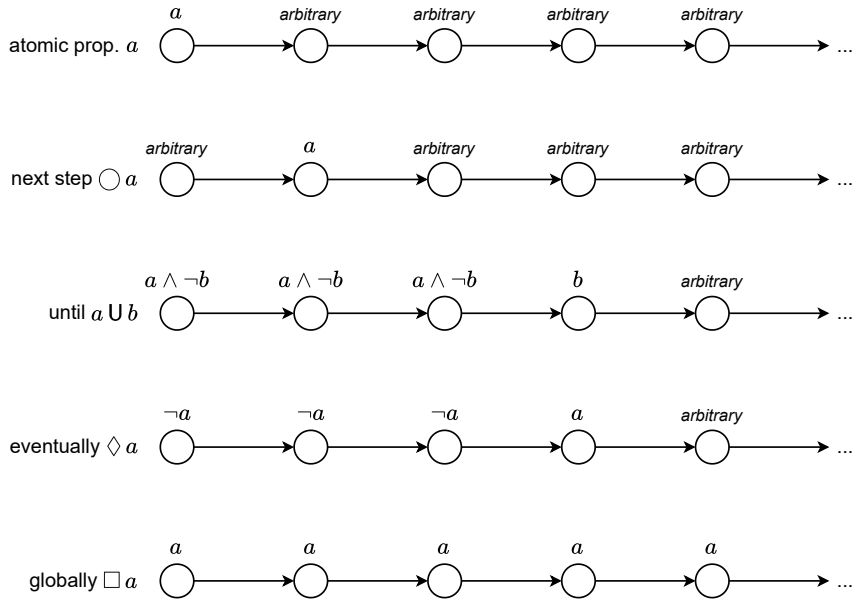


Fig. 2.4: Intuition behind LTL operators [18]

2.3.4 *Example*

Imagine a system for an e-commerce platform (like amazon.com or your favorite online store) that must handle order fulfillment, i.e., coordinate payment and shipping of goods. In this system, the order handling process is modeled using BPMN. Business Process Model and Notation (BPMN) is a standardized graphical modeling language for representing business process models. Thus, placing an order triggers the BPMN process depicted in Figure 2.5[3]. This example is *not* a distributed or safety-critical system but illustrates how key aspects of a system can be verified using model checking.

The BPMN process in Figure 2.5 is triggered by an order placement and intends to model retrieving payment and fetching goods in parallel. Afterward, the goods are shipped, which leads to a successful order delivery. Fetching and shipping goods are manual tasks that humans execute, while retrieving payment is an automated task. A detailed introduction to BPMN can be found in [58], while its application for process automation is discussed in [167].

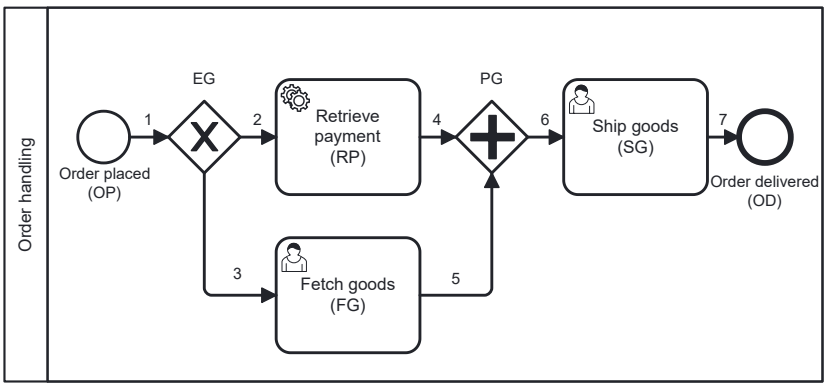


Fig. 2.5: A BPMN process for *Order handling* (adopted from [167])

[3]The process deliberately contains an error later identified and resolved in this section.

The BPMN process in Figure 2.5 represents the *system model*, while there is one *system requirement*. Each order placed must eventually result in a shipment to the customer. Thus, the next step is to compile the system model into a Kripke structure and translate the requirement into a temporal logic formula to check if it is fulfilled.

First, for the translation to a temporal logic formula, we need to define the set AP of atomic propositions, i.e., interesting facts of the system model we want to verify. In this case, we can use $AP = \{\texttt{running}, \texttt{terminated}\}$, to describe if our process is *running*, meaning there might not have been a shipment to the customer yet or *terminated* meaning the shipment has been carried out. We can then define the LTL formula φ in (2.3) to check if each placed order leads to a shipment, i.e., each invoked process eventually completes.

$$\varphi = \Diamond\, \texttt{terminated} \tag{2.3}$$

Second, compilation of the order handling process in Figure 2.5 results in the Kripke structure K shown in Figure 2.6. K consists of five states and four transitions. The proposition running holds in all of the states ($\forall s \in S : L(s) = \{\texttt{running}\}$).

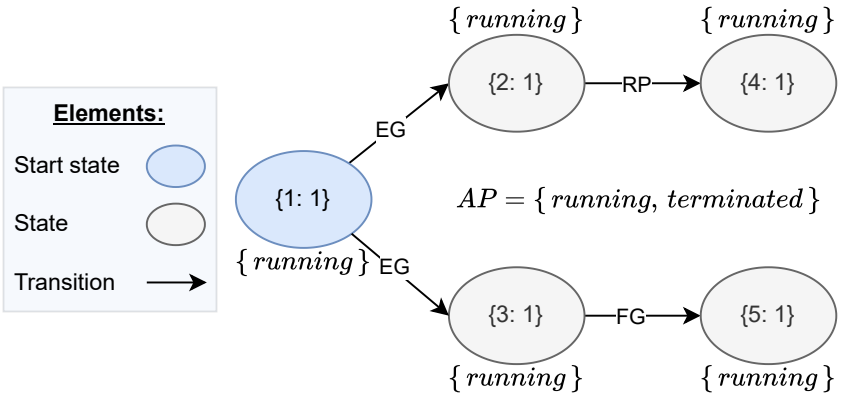


Fig. 2.6: Kripke structure K of the order handling process in Figure 2.5

The BPMN semantics is described using tokens distributed over the model, similar to Petri Nets [85, 86]. Each state in Figure 2.6 describes its token distribution, i.e., the start state $\{1:1\}$ means that there is one token at the edge (sequence flow) labeled 1 in Figure 2.5. A state in a BPMN model is labeled as terminated if its token distribution is empty ($\{\ \}$). The compilation from the BPMN model in Figure 2.5 to the Kripke structure, often called *state space*, is automated by encoding the BPMN execution semantics [143] and performing an exhaustive breadth-first search (BFS) or depth-first search (DFS) [40].

A model checker given the Kripke structure K and the formula φ will output $K \not\models \varphi$ alongside a counterexample (see, for example, the BPMN Analyzer 2.0 in the artifacts [185]). Regarding K, there are two possible counterexamples, either following the upper or lower transitions in Figure 2.6. Both lead to states that do not fulfill the proposition terminated. One can see that both states $\{4:1\}$ and $\{5:1\}$ do not have outgoing transitions, i.e., the BPMN execution is stuck just before the parallel gateway. A parallel gateway in BPMN requires one token for each incoming arrow (sequence flow) to execute. However, there is either a token on flow 4 or 5, but never both. An

interactive visualization of the counterexample is also available in our BPMN Analyzer 2.0 tool[4], featured in Paper E. We can fix the model by changing the exclusive gateway *EG* to a parallel gateway, see Figure 2.7 (also proposed by the BPMN Analyzer 2.0).

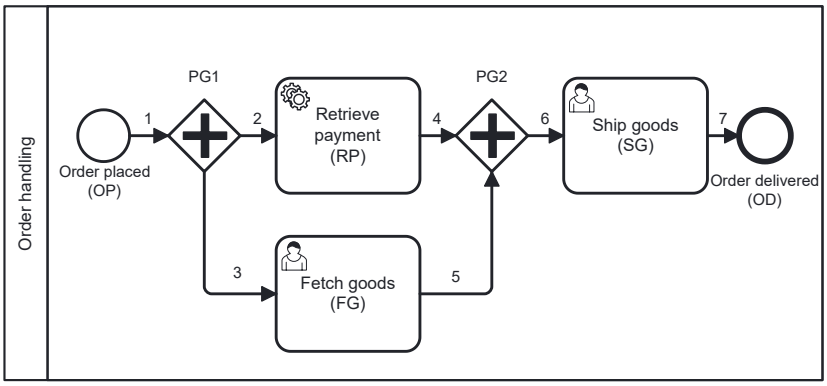


Fig. 2.7: Correct BPMN process for *Order handling* [167]

The compilation will result in a new Kripke structure, see Figure 2.8, which fulfills the formula φ defined in (2.3). One can see that eventually, the last state without any tokens is reached, which fulfills the property `terminated`.

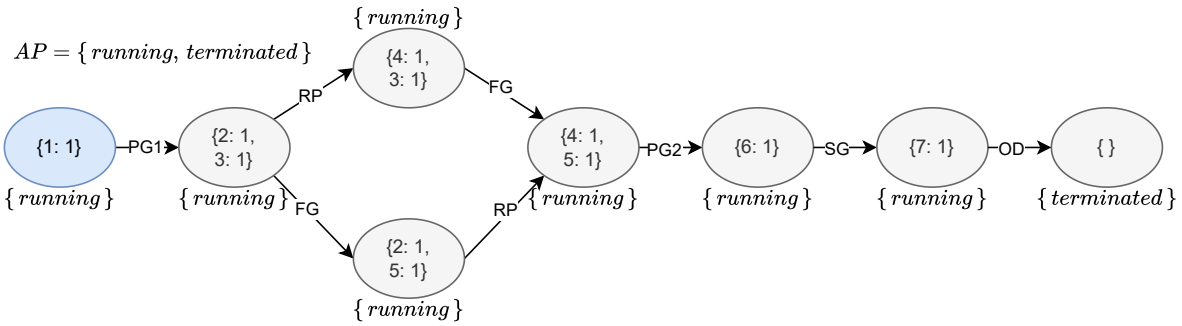


Fig. 2.8: Kripke structure K of the correct order handling process in Figure 2.7

2.4 Rewriting Logic & Maude

Maude[5] is an executable high-level language that is based on *rewriting logic* as a solid mathematical foundation and incorporates a wide range of analysis techniques [41, 119]. Rewriting logic is a novel combination of *algebraic specification* and *term rewriting* [41]. In this thesis, we use rewriting logic, specifically Maude, as a general semantic framework to give semantics to heterogeneous modeling languages by describing their state changes as rewriting rules [41, 122, 130]. Our implementations in Paper B and Paper D use Maude.

A rewriting theory is a pair (T, R), where T is a membership equational theory, and R is a set of labeled (possibly conditional) rewriting rules [41]. We will not include

[4]Open the tool at `https://timkraeuter.com/bpmn-analyzer-js?model=orderHandling` and click on *Guaranteed termination* in the top right to launch the visualization.

[5]Maude is open-source and freely available online at `http://maude.cs.uiuc.edu/`.

formal definitions of rewriting logic in this thesis; however, readers interested in the foundational theory of rewriting logic can find an accessible introduction in [148].

Structure Intuitively, one can think of the equational theory T in a rewriting theory as its *static* part, i.e., the definition of data types and their relations in a system. We call this the *structure* of a modeling language or system. One can define which sorts exist, what their elements are (using constants and operators), and how sorts are related (subsorts). In addition, object-oriented Maude modules enable the definition of typical elements found in object-oriented languages, such as classes, attributes, inheritance, and messages. This defines a set of terms that can represent the structure of a modeling language or system. By defining equations, one defines when two terms are equal (potentially modulo commutativity, distributivity, associativity, idempotency, and identity elements), which helps to simplify and unify terms (canonical form) before rewriting. We use equational theory to represent models conforming to heterogeneous languages and how the states of instances of these models are structured.

Example 4 shows a Maude specification describing a *simple state machine language* consisting of states and transitions between them and state machine instances. For state machines and their instances, we use Maude's object-oriented notation[6]. The corresponding UML class diagram [144] is depicted in Figure 2.9. We define two sorts, `State` and `Transition`, along with operators that allow us to construct elements of each sort. A transition between states A and B can then be defined as `state("A") -> state("B")`. Then, we define two classes using object-oriented Maude: `StateMachine` and `StateMachineInstance`. The `StateMachine` class represents a state machine that consists of a set of transitions, while the `StateMachineInstance` class represents an instance of a state machine, linking to its corresponding state machine model (`sm : Oid`) and maintaining a current state (`currentState : State`).

Example 4: Simple State Machine Language (Structure)

```
sorts State Transition .
op state : String -> State [ctor] .
op _->_ : State State -> Transition [ctor] .
class StateMachine | transitions : Set{Transition} .
class StateMachineInstance | sm : Oid, currentState : State .
```

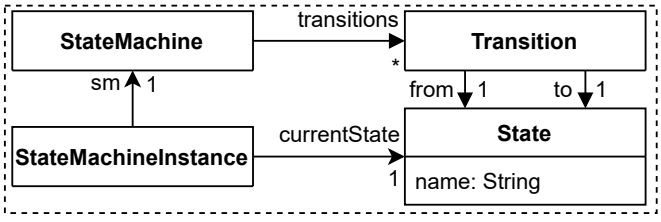


Fig. 2.9: Simple State Machines (UML Class Diagram)

[6]For didactic purposes, we did not use Maude's object-oriented notation to define state and transition.

We provide a concrete introduction to Maude and rewriting logic using the simple state machine example. The sources of the example are available in [185]. For a comprehensive overview of Maude and its features, see [119].

To define language metamodels, models specified by these metamodels, and instances of these models (representing their execution states), we adopt the methodology illustrated in Figure 2.9, where we collapse multiple levels of abstraction.

Behavior The rewriting rules R in a rewriting theory (T, R) can be understood as descriptions of system changes, representing the *dynamics of a system* or the operational semantics of a modeling language. We generally refer to this as the *behavior* of a system or modeling language, which is defined using rewriting rules in rewriting logic. A rewriting rule $r : t \to t'$ represents a possible transformation that can be applied within a system or modeling language. Intuitively, the rule r describes that whenever the system is in a state represented by the term t, it can transition to the state described by t'. The terms t and t' are elements of the set of terms defined by T, i.e., t and t' conform to the specified structure (see [148] for more details). Rewriting happens modulo the equations defined in T [41], meaning before rewriting, each term is first reduced to its canonical form using the equations in T. Thus, whenever the system is in a state *equivalent* to t regarding the defined equations, it can transition to the state t'. Multiple rewriting rules can apply to the same term, leading to numerous possible rewriting paths, which enables rewriting logic to represent nondeterministic systems [148]. Example 5 shows an example of a rewriting rule to implement the semantics of the state machine language.

Example 5: State Machine Rewriting Rule (Behavior)

```
vars SMId SMIId : Oid .
var Transitions : Set{Transition} .
var From To : State .
rl [fire-transition] :
    < SMId : StateMachine |
        transitions : (From -> To, Transitions) >
    < SMIId : StateMachineInstance | sm : SMId,
        currentState : From >
 =>
    < SMId : StateMachine | >
    < SMIId : StateMachineInstance | sm : SMId,
        currentState : To > .
```

The rule in Example 5 fires a state machine's transition by changing its current state from the source to the target of the transition. The state machine model remains unaltered, represented by a concise notation that does not repeat its attributes.

Rewriting logic supports a range of formal methods. In this thesis, we mainly use the explicit-state model-checking capabilities of Maude in Paper B and Paper D. For this purpose, the rewriting theory must be executable, meaning the equations must be both

terminating and Church-Rosser (whenever two terms can be reached from a common term, there exists a third term to which both can be further rewritten), and the equations and rules must be coherent such that their interleaving works properly [41, 119, 148]. In our examples, this is typically not an issue, but obtaining a finite state space can pose a challenge. Furthermore, rewriting logic can also be used for simulation and debugging, symbolic model-checking, statistical model-checking, or other formal proofs besides model checking [41, 119].

Model checking To perform model checking, one follows the methodology outlined earlier in Figure 2.3[7]. One must describe the system model in Maude, i.e., define the system's *structure* using sorts, operators, and classes from object-oriented Maude while describing its *behavior* using rewriting rules. Given an initial state, Maude associates a Kripke structure to a rewrite theory that can be used for model checking [119] (compilation). In a Kripke structure, the states are represented by the terms of a rewrite theory, and the corresponding rewriting rules define the transitions. To define atomic propositions, one defines equations on top of the system's structure. Example 6 shows how to define a proposition. The first line says that a `Configuration` (predefined sort in Maude), which represents a multi-set of objects and messages, describes the state of our system by defining a `subsort` relation. Thus, each state in the Kripke structure is a multi-set of objects and messages. In our case, only the objects, state machine models, and instances are used, as defined earlier.

Example 6: State Machine Atomic Propositions

```
subsort Configuration < State .
op is-in-state-X : -> Prop .
var C : Configuration .
var Id MId : Oid .
eq < Id : StateMachineInstance | sm : MId,
    currentState : state("X") > C |= is-in-state-X = true .
```

The atomic proposition `is-in-state-X` is true if a state machine instance is in the state `"X"`. One could even create parametric propositions by changing the operation `is-in-state-X`. Propositions are false by default. Maude has a built-in LTL model checker, which can construct Kripke structures and check LTL properties using propositions as defined in Example 6.

Usage We use the Maude LTL model checker in Paper B and Paper D. Furthermore, modeling data exchange in Paper D is straightforward because Maude supports basic data types and allows the definition of additional or more complex types as needed. To extend our analysis to real-time systems, we utilize Real-Time Maude [147], an extension of Maude [119]. In particular, we use time-bounded model checking and reachability analysis [146, 147].

[7] In practice, Maude does not strictly adhere to the basic model checking methodology because it incorporates optimizations such as on-the-fly model checking.

In a real-time rewriting theory, rules are classified into two types. *Ordinary* rewriting rules are instantaneous, meaning they take zero time to execute. On the other hand, *tick rewriting rules* represent the passage of time within the system. Real-time Maude extends the given system state by adding a clock, which can model dense or discrete time. This clock represents global time and changes during tick rule applications. One can then run model checking for systems that run indefinitely, i.e., have an infinite state space (typically reactive systems) until a certain time-bound is reached, which may[8] be finite [146].

2.5 Graph Transformation & Groove

Graph transformation, also known as graph rewriting, is similar to term rewriting; however, instead of rewriting terms, we modify graphs, as the name suggests. Graphs are fundamental in computer science and can be used to model entities and relationships in different domains, such as data structures, control flow, and software architectures. The field of graph transformation describes a systematic approach to changing graphs using a rule-based approach [72, 75, 166].

There are many different variations of graphs that are used in various scenarios. For example, graphs can be undirected or directed, have attributes on nodes or edges, and contain other typing-related information, such as abstract nodes and multiplicities. A graph feature model is given in [75], which shows the different possible graph models. In this thesis, we use *attributed typed graphs*, where attributes are only allowed on nodes. Such graphs are sufficient to represent the modeling languages I have encountered during my PhD project. In addition, attributed typed graphs are implemented in the graph transformation tool Groove [62], which we employ in this thesis. In the following, we give a short informal introduction to graph transformation using Groove. A detailed introduction to graph transformation for software engineers is given in [75], while the theoretical fundamentals of graph transformation are described in [43, 48, 72, 74, 140, 166].

A graph transformation system $GTS = (TG, P)$ consists of a type graph TG and a set of typed production rules P, i.e., typed graph transformation (GT) rules [72]. In a nutshell, given an initial graph that conforms to TG, the production rules P can transform the graph by adding and removing nodes as specified by P. By exploring every possible transformation of the initial graph, one can analyze the behavior of the graph transformation system.

Structure Much like the sorts and operations of a rewriting theory, the type graph describes the *structure*, i.e., the data types and relations in a system or the structure of models in a modeling language. The upper part in Figure 2.10 shows the simple state machine language type graph and an example instance in the lower part. Both are depicted in the syntax of the graph transformation tool Groove. The type graph is very close to the UML class diagram shown earlier in Figure 2.9. The graph in the lower part conforms to the type graph and models a state machine with one transition from

[8] The resulting state space is finite if time was the reason. However, other factors, such as the unbounded growth of data structures, can also lead to infinite state spaces.

state "A" to state "X" and its instance, which is currently in state "A". The complete example, executable in Groove, can be found in [185].

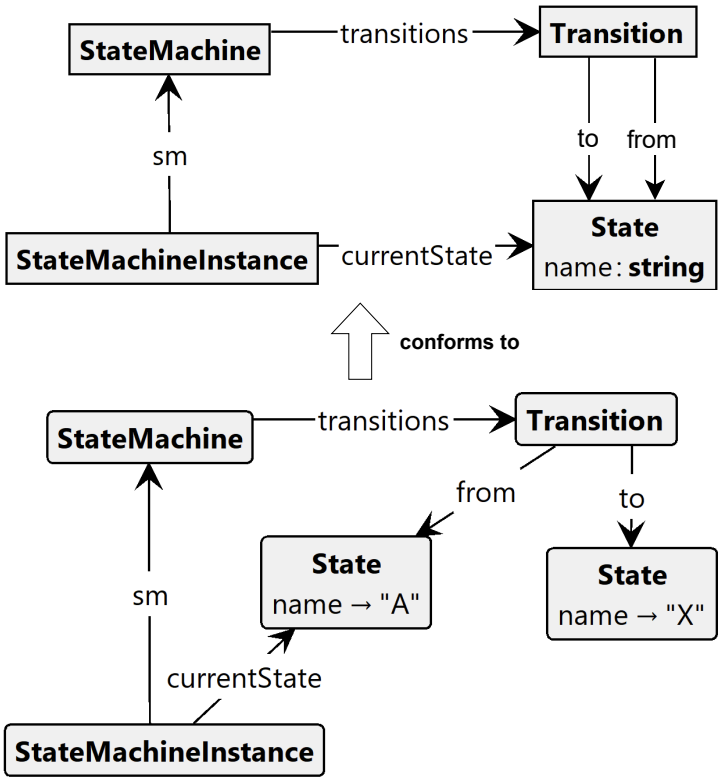


Fig. 2.10: State Machine Language: Type Graph & Conforming Graph (Groove)

An edge in the type graph indicates that there can be one or more edges between nodes conforming to the specified source and target types. Therefore, the type graph in Figure 2.10 permits multiple `Transition` nodes, allowing us to model the complete set of transitions within a state machine. While all other edges between any two nodes should ideally be unique, the type graph does not enforce this.

Behavior To describe the *behavior* of a system or modeling language, one defines P, a set of typed GT rules. These GT rules rewrite graphs conforming to the previously defined structure in the type graph TG. Figure 2.11 illustrates a sample graph transformation rule using Groove syntax that triggers state machine transitions. The black elements need to be present and will be preserved during transformation, while the dashed blue elements need to be present but will be removed. Furthermore, the fat green elements will be created. In Figure 2.11, only the state edge is modified to update the state of the state machine instance, similar to the earlier Maude rewriting rule. GT rules also allow for so-called Negative Application Conditions (NACs) to define which nodes and edges must not exist for the rule to be applicable.

Model checking Kripke structures are generated from GT systems. The generation starts from an initial state, i.e., a start graph. Each state in the generated Kripke structure is associated with a graph, and transitions correspond to GT rule applications [162]. Starting with the start graph, Groove repeatedly applies GT rules to each state to create

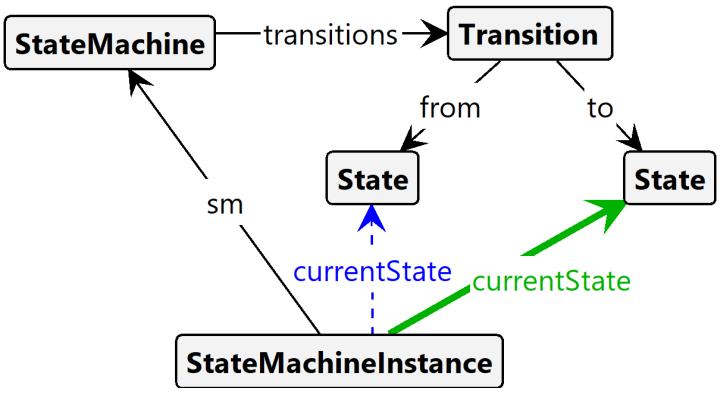


Fig. 2.11: State Machine Language: GT Rule (Groove)

a Kripke structure. The resulting Kripke structure is then used to check the model with LTL or CTL. One defines a GT rule that does not modify the underlying graph to specify an atomic proposition that can be used in a temporal logic property. For example, since it makes no modifications, the GT rule in Figure 2.12 defines an atomic proposition. The atomic proposition holds for a given state if there is a match of the GT rule in the graph associated with that state [162]. Intuitively, the atomic proposition is true if the graph defined in the GT rule can be found in a given state.

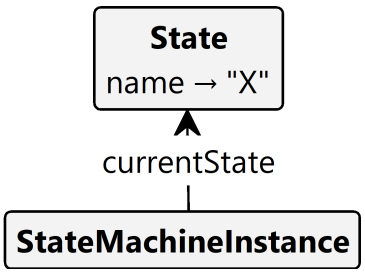


Fig. 2.12: State Machine Language: Atomic proposition (Groove)

Usage In Paper B and Paper C, we use Groove [160]. We leverage Groove's explicit-state model-checking capabilities [88, 162] to analyze BPMN models in Paper B and heterogeneous models in Paper C. Groove supports both LTL and CTL temporal logic properties for model checking. In addition, we utilize *nested rules* with quantification [161, 163, 164] in Paper B to make parts of a rule repeatedly applicable or optional.

We fully transitioned to Maude to implement our approach presented in Paper D, which introduces support for data exchange and real-time capabilities. Adding data exchange would not be a problem, as Groove supports basic data types. In addition, there is existing theory on timed GT systems [125, 173]. However, implementing timed GT systems in Groove would require development from scratch, whereas Maude offers existing implementations and established best practices that we can build upon.

"In the midst of chaos, there is also opportunity."

— Sun-Tzu

CHAPTER 3

STATE OF THE ART

In this chapter, we aim to provide a comprehensive overview of the research landscape and to establish a clear framework for categorizing and comparing various approaches to system coordination. We use *system* to refer to individual programs of varying sizes, a collection of systems that work together, or a model that describes the behavior of a system. To accomplish this, we conduct a *meta-analysis* of existing literature studies to construct a *feature model* [87]. The created feature model provides an understanding of the coordination domain regarding scope, features, and motivation.

This chapter is organized as follows. First, we present the various categories of coordination approaches identified during my research, which provide the foundation for the subsequent meta-analysis. Second, we outline the methodology of my analysis. Third, we present the resulting feature model, the classification of coordination approaches using the feature model, and additional findings. Finally, we conclude by highlighting the contributions of this chapter.

3.1 Categories of Coordination Approaches

Figure 3.1 gives an overview of the categories of coordination approaches. These categories broadly operate on three different abstraction levels, namely the *execution*, *model*, and *language* levels. We will explain each abstraction level from the bottom up and mention the accompanying categories. A detailed explanation of each category is given in the following subsections. In general, the right abstraction level to address coordination depends on the specific use case and goals at hand, i.e., approaches on each abstraction level have their advantages and disadvantages.

Execution level: Approaches on the *execution level* aim to deal with coordination by interfacing directly with executable binaries or source code. We identified two distinct categories on this level: co-simulation approaches and coordination languages. These approaches are discussed in subsection 3.1.1 and subsection 3.1.2.

Model level: *Architecture Description Languages (ADLs)* operate on the *model* level, i.e., each *component* is given by a behavioral model, for example, a process algebra. An overview of ADLs is given in subsection 3.1.3.

Language level: *Coordination frameworks* operate on the language level since they allow coordination of models conforming to *different* behavioral languages. They are outlined in subsection 3.1.4.

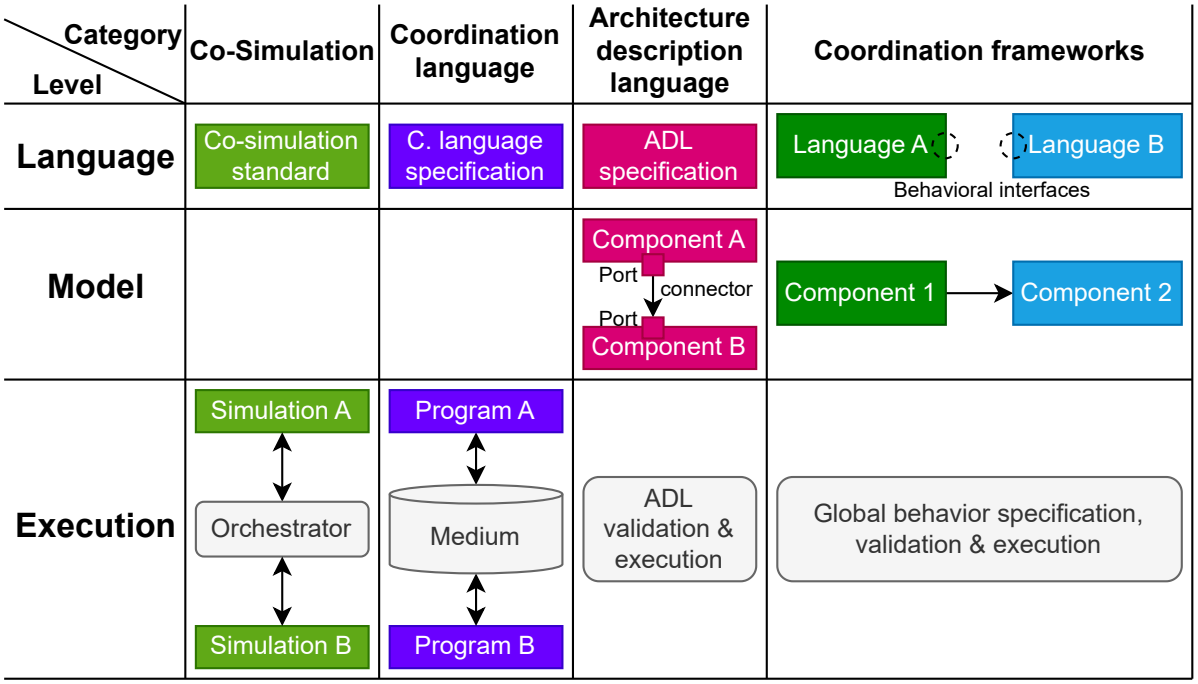


Fig. 3.1: Overview of coordination approaches

3.1.1 *Co-simulation Approaches*

Co-simulation approaches involve coordinating timely information exchanges between multiple simulations running concurrently. In this configuration, a simulation is the execution of a simulation unit that consumes inputs and produces outputs. A simulation unit is typically specified as a *black box*, i.e., its behavior is opaque and accessible only through its simulation interface. Using such interfaces, the co-simulation must ensure the temporal and causal relationships between each simulation. This is usually achieved by defining an *orchestrator* [63], which dictates the progression of simulated time and transfers data between simulations. Co-simulation approaches are categorized as *Discrete Event* (DE), *Continuous Time* (CT), or *Hybrid* if DE and CT aspects are mixed [63].

Currently, there are two primary standards for co-simulation: the Functional Mock-up Interface (FMI) [138] and the High Level Architecture (HLA) [45]. The FMI standard is better suited for CT co-simulation, while the HLA standard is better suited for DE co-simulation [63, 107]. FMI is based on a central orchestrator, the Master Algorithm, to coordinate the execution of the components, while HLA provides a middleware, the *RunTime Infrastructure* (RTI), for component interactions instead of an orchestrator. Researchers are investigating how to best combine the two approaches in the future [54].

To bridge CT and DE co-simulations, MECSYCO [32, 33] uses a DE-based orchestrator but achieves hybrid co-simulation by wrapping CT simulations specified using FMI in the Discrete Event System specification (DEVS). Wrapping CT simulations using DE simulations or vice versa is a common strategy to achieve Hybrid co-simulation [63].

3.1.2 Coordination Languages

Many different styles of coordination languages exist, which achieve coordination between systems differently. In this context, a system is an executable program of varying size. Two main coordination language families exist (see Figure 3.2): the Linda and IWIM family[37]. Both language families achieve coordination fundamentally differently.

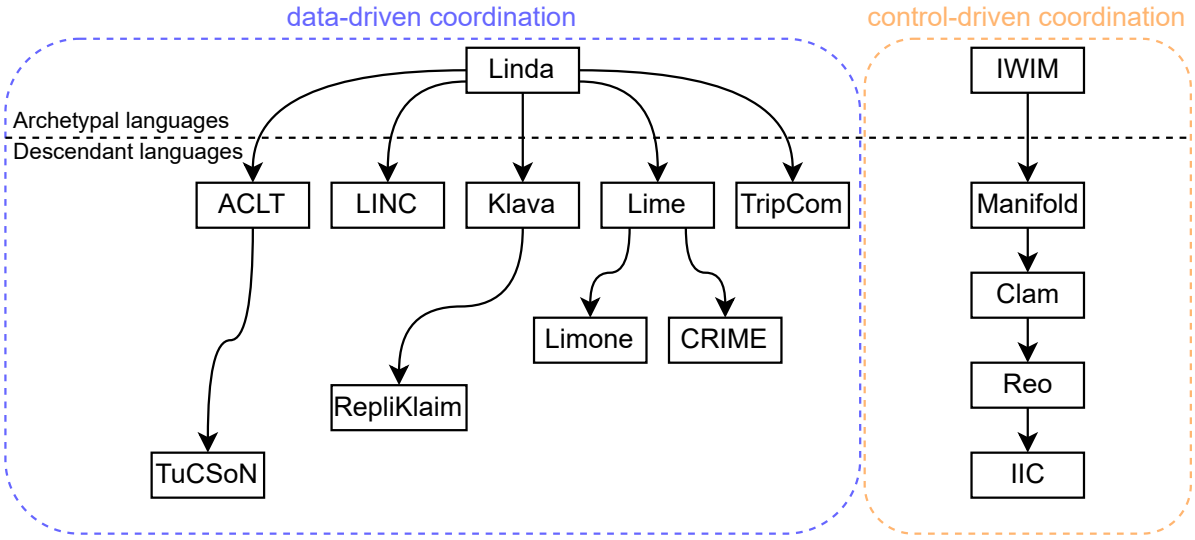


Fig. 3.2: Main coordination language families (adapted from [37])

The first family of coordination languages originates from Linda [34]. It is often called data-driven since coordination is achieved by data exchange. Linda provides a *set of operations* to enrich a host programming language with coordination capabilities. The operations in Linda are used to read from and write to a virtual shared memory, which serves as a *global* communication medium. Such coordination languages are often called *tuple-based* and classified as *data-driven* because the shared memory contains sequences of elements, i.e., tuples that represent exchanged data. Coordination languages from the Linda family are surveyed in [37, 142, 149, 165].

The second family is control-driven and comprises the descendants of the IWIM model for coordination [12]. For example, the languages Manifold [16, 156] and REO [14] do not rely on a global communication medium such as tuple-based languages but rather use *channels* to connect systems locally. Each system then uses I/O operations to interact with connected channels without knowing who has sent or will receive data through the channel. These coordination languages are *control-driven* [37] and share many ideas with Architecture Description Languages (ADLs) discussed in subsection 3.1.3.

Furthermore, coordination languages might also use communication models such as the actor or reactor [111] model. For example, the Lingua Franca [111, 112] coordination language is described using the reactor model, i.e., one defines coordination by defining how each system reacts to incoming events. A reaction to an event in one system might trigger reactions in connected systems. To encode reactions, the user can choose an existing programming language. We classify Lingua Franca as a data-driven coordination language. However, it is not tuple-based and thus not a direct descendant

of Linda.

Coordination approaches based on coordination languages have been enriched by concepts from chemistry and biology, leading to self-organizing coordination [186], and by concepts from physics, resulting in field-based coordination [118]. A detailed overview is given in [191].

3.1.3 Architecture Description Languages

Architecture Description Languages (ADLs) aim to describe the structure of systems, allowing developers to focus on high-level system components and their connections rather than lines of source code [42, 127, 128]. Many different ADLs have been proposed in the academic literature and by the industry [127, 195]. Nevertheless, clearly defining ADLs is challenging due to overlap with general-purpose modeling languages [42]. This distinction is deeply studied in [127].

The three building blocks of ADLs are defined as (1) *components*, (2) *connectors*, and (3) *architectural configuration* [127, 128]. A *component* is a unit of computation or data repository [127]. Components vary in size, ranging from representing individual services to entire systems. In this thesis, we use the more general term *system* when we are not in the immediate context of ADLs.

Connectors serve as architectural elements to model interactions between components and the regulations that oversee those interactions [127]. A difference to components is that connectors must not be implemented as distinct entities, such as message brokers, but can also represent shared variables or links between applications realized by client-server protocols [127].

Architectural configuration, also known as topology, represents the structural arrangement of components and connectors in a connected graph, defining the overall architecture [127]. This structure determines if the combined semantics satisfy the desired behavior. Verification relies heavily on specifying components and connectors. For instance, one common application ensures the architectural configuration is free from deadlock and starvation.

The first ADLs were developed in the 90s and use process algebras such as Communicating Sequential Processes (CSP), Calculus of Communicating Systems (CCS), and π-calculus [151]. For example, Wright uses CSP [7] while Darwin uses π-calculus [115]. More recent ADLs, such as MontiArc [67], use automata to define the behavior of components.

However, to my knowledge, no ADL supports heterogeneous components such as the coordination frameworks discussed in the next section [127]. Furthermore, despite the creation of numerous ADLs in the literature, they are not mainstream, i.e., not often used by practitioners in the industry [42, 126, 151, 153, 195].

3.1.4 Coordination Frameworks

We introduce the term *coordination framework* to describe the systematic establishment of coordination at the language level through the utilization of patterns, followed by an automatic derivation of the coordination on models that adhere to the languages (see Figure 3.1). Coordination frameworks primarily concentrate on coordinating models

that adhere to *different* behavioral languages. They employ the concept of *language behavioral interfaces* to define the *synchronization points* that can be used during the coordination of models. This is valuable because each part of the system might be defined using different behavioral languages best suited to its nature. Additionally, reifying coordination allows for 1) the inspection, sharing, and review of proposed coordination patterns and 2) automatically providing the coordination for each model conforming to that language.

For example, in Paper B, we define a standard language behavioral interface that identifies state-changing elements used across different behavioral languages. One can use this uniform interface to define coordination between systems specified in heterogeneous behavioral models. Concretely, one can define that a state change in one system should be synchronized with a state change in a different system by identifying the corresponding state-changing elements, i.e., synchronization points in each model. The semantics in Paper B are given by rewriting system specifications, such as graph- or term-rewriting, which can then be executed by corresponding tools such as Groove [160] and Maude [119] for simulation and verification purposes. In addition, coordination frameworks such as B-COoL [188, 189] and Ptolemy [49, 158] exist.

3.2 Methodology

Systematic literature reviews [90] (SLRs) and systematic mapping studies [157] (SMSs) constitute the highest-quality of literature studies. In addition, less rigorous types of studies exist, often called surveys, overviews, or reviews. These studies, including SLRs and SMSs, are known as secondary studies because they compile and analyze the primary research results. However, secondary studies typically require a lot of effort and ideally involve a large group of researchers from different institutions to ensure an unbiased selection and analysis of primary studies. Thus, to make a comprehensive study of related literature possible, we conducted a meta-analysis, i.e., **tertiary** study of the existing secondary studies that analyze each coordination category.

Figure 3.3 gives an overview of the *meta-analysis* methodology used to study existing literature and construct a feature model [87]. **First**, we searched for secondary studies in each coordination category. We found the following studies for coordination languages ([15, 37, 60, 66, 154, 155, 190]), ADLs ([42, 81, 110, 117, 127, 128, 137, 151, 153]), and co-simulation approaches ([63, 64, 68, 69, 174, 175]). The term coordination framework has been newly introduced, so no surveys have been conducted. It has only been used in a PhD thesis before [188]. We searched *Google Scholar*[1] and *Scopus*[2] to locate the listed studies. In conducting the literature search, I adhered to the PRISMA guidelines for reporting systematic reviews [152] within the available time constraints. However, it is vital to emphasize that the outcome is a meta-analysis, *not* a systematic review.

Figure 3.4 shows a detailed report of the search step in our meta-analysis following PRISMA [152]. To ensure manageability, we restricted our Google Scholar search to the first five pages of results (50 papers), as each query yielded over 4000 entries,

[1] https://scholar.google.com
[2] https://www.scopus.com

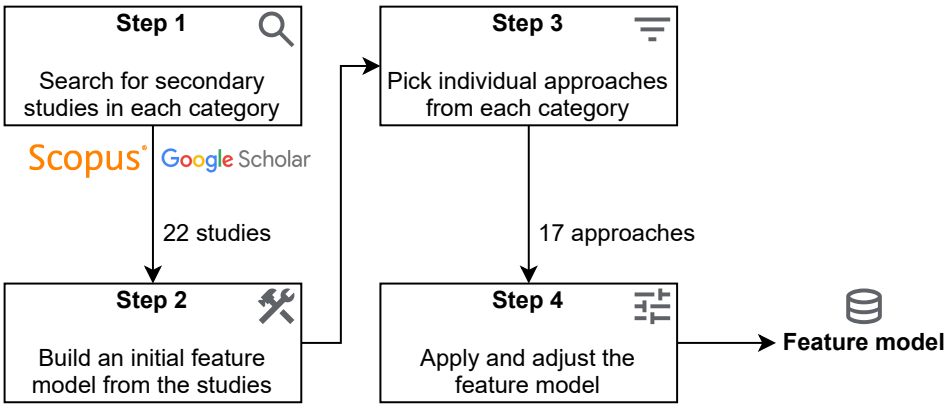


Fig. 3.3: Meta-analysis methodology

while for Scopus, all results were considered. A comprehensive export of all identified papers resulting from these queries is provided in [185]. In the next step, we filtered all identified papers to only include secondary studies relevant to the specified coordination categories by screening each paper's title, abstract, and content. Finally, we removed duplicate papers identified in both Google Scholar and Scopus, resulting in a final selection of the 22 papers listed above.

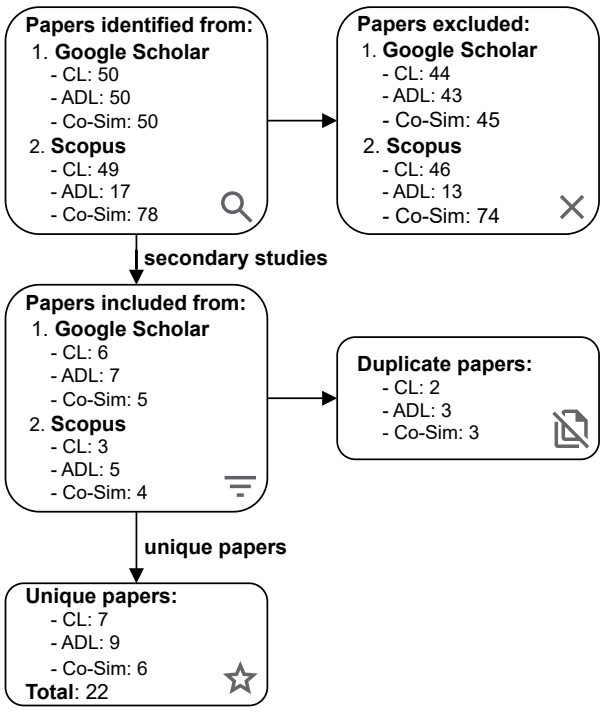


Fig. 3.4: Search step in the meta-analysis (Flow diagram [152])

The search queries used for each coordination category are outlined below. The search query to find studies about coordination languages is given in (3.1).

```
(survey OR taxonomy OR review OR overview
    OR state of the art OR framework)
                    AND
          coordination language
```
(3.1)

The search query to find studies about architecture description languages is given in (3.2). It is similar to the previous search query.

```
(survey OR taxonomy OR review OR overview
    OR state of the art OR framework)
                    AND
          architecture description language
```
(3.2)

The search query to find studies about co-simulation approaches is given in (3.3). We omitted the term `framework` from the query since co-simulation approaches are often called co-simulation *frameworks*.

```
(survey OR taxonomy OR review OR overview OR state of the art)
                    AND
          (co-simulation OR co simulation)
```
(3.3)

Second, I conducted a *meta-analysis* (tertiary study) based on the identified studies to create an initial feature model. **Third**, due to the immense number of approaches and limited time, I focused on the 17 approaches listed in section 3.4. I selected these specific approaches to have a good distribution across the four categories and heterogeneity inside each category. For example, I chose coordination languages from different families (Figure 3.2), namely Linda, Reo, and Lingua Franca, which employ different coordination models, namely tuple spaces, channels, and reactors.

Finally, I investigated these approaches to adjust and improve the feature model continuously. The final feature model is described in the next section. In addition, I classified all 17 approaches using the feature model [185].

3.3 Feature Model

In this section, we present the constructed *feature model* [87]. We aim to comprehensively understand the coordination landscape and compare coordination approaches from different categories. To achieve this goal, we first identify the coordination *foundation*: coordination mechanism, syntax, and semantics. Second, we investigate the coordination approaches *goal* such as simulation, execution, and formal verification. Third, we study the approaches' *properties*, such as system transparency, domain, execution, system languages, and specification.

The top level of the feature model is shown in Figure 3.5. A coordination approach has the three previously discussed features: *Foundation*, *Goal*, and *Properties*, which we depict and describe in detail in the following sections.

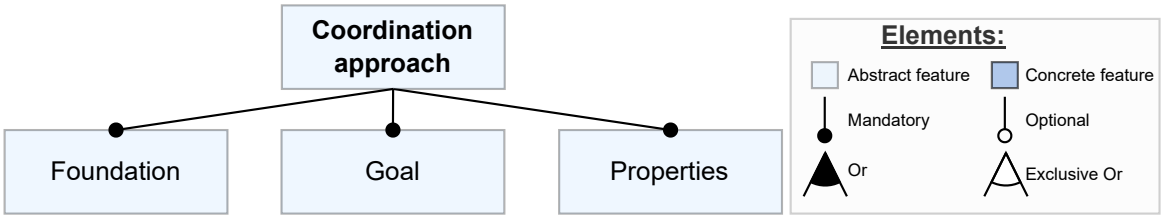


Fig. 3.5: Feature model overview

3.3.1 *Foundation*

Figure 3.6 shows the **Foundation** feature, which consists of **Syntax**, **Semantics**, and **Mechanism**.

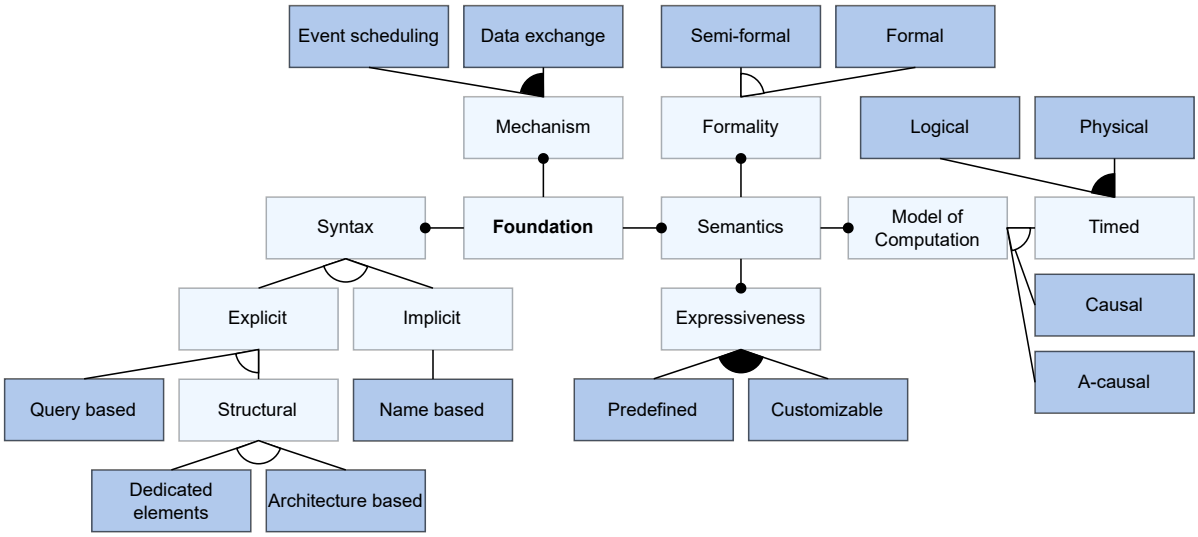


Fig. 3.6: Foundation feature

Syntax The coordination syntax of the elements on which to apply coordination can either be *explicit* or *implicit*. An implicit syntax is usually based on the names of elements. For example, one could synchronize the firing of transitions from multiple state machines if they have identical names.

Explicit syntax is either *query based* or *structural*. For example, in B-COoL, one can write queries to define which elements should be coordinated [188, 189].

A structural definition of coordination uses *dedicated elements* or is *architecture based*. For example, interactions in Paper B are specified by picking dedicated elements that should be coordinated, while coordination in ADLs depends on the architectural configuration [127], i.e., how ports of systems are connected.

Mechanism The main mechanism driving coordination can be *event scheduling*, *data exchange*, or a mix of both. In the literature, these mechanisms are sometimes called control/event-driven or data-driven, respectively [155, 188].

First, *event scheduling* synchronizes events or imposes a specific event order, i.e., order of state changes in different systems. Usually, event scheduling is used when systems are given by behavioral models, since there might not be a clear concept of an event in a programming language. For example, in B-COoL [189], one can define relationships between events, such as *happens before* or *synchronize*.

Data-exchange means systems communicate by exchanging data. For example, in the ADL MontiArc [67], systems send data from the output to the input port. Data can also be exposed as variables, which can be coupled with variables in other systems, for example, to be equal. However, data exchange might also happen between systems during the synchronization of events, which is why both coordination mechanisms can be supported simultaneously.

Semantics The semantics feature consists of *expressiveness*, *model of computation*, and *formality*. The expressiveness of coordination can be *predefined*, i.e., a user has a fixed set of coordination mechanisms, for example, a fixed set of connectors with predefined behavior. Expressiveness is *customizable* if one can define the semantics of coordination, typically by using a dedicated language. For example, in B-COoL, one can employ the Clock Constraints Specification Language (CCSL) [9] to define new coordination operators besides the predefined ones [188, 189].

Coordination approaches utilize different *models of computation* (MoC). A MoC is a set of rules that governs the timed, possibly concurrent execution and coordination of systems [158]. We categorize MoCs along three types: *timed*, *causal*, *a-causal*. A *timed* MoC can use *physical* or *logical* time. The categorization of *causal* and *a-causal* MoCs is also present in the co-simulation field [63]. A causal MoC means that there is a notion of inputs and outputs in the events and data exchanged between system parts. An output can cause reactions, i.e., state changes in the input of the other systems. For example, a system based on a state machine raises an event, which causes a reaction in a different system.

In an a-causal MoC, there is no notion of inputs or outputs of data, but instead of equality (in the mathematical sense). For example, two a-causally coordinated variables must be equal at any time. This is, for instance, typical in differential equations.

Logical time is a concept that provides a way to reason about the partial order of events (and data exchanges) without needing to rely on a physical clock. It allows for the specification of causal and a-causal relationships between events and helps ensure the consistency and correctness of loosely coupled systems [46, 56]. For instance, one can specify that an action is performed every ten startups of the system, regardless of how frequently the startup physically occurs.

In contrast, *physical* time describes time passing in the physical world. It is also referred to as *wall-clock* time [63]. For example, one can define that an event should happen 50 milliseconds after another. In Lingua Franca [111], logical time "chases" physical time, i.e., logical time tries to match the physical time provided by the execution platform.

Formality of a coordination approach is either *formal* or *semi-formal*. Semi-formal approaches are directly implemented in a general-purpose programming language, often with a guiding formalism in mind but without strict adherence to that formalism.

Formal coordination approaches are implemented in a *formal language*, which can then be run on an execution engine implemented for that formalism. For example, the coordination framework in Paper D uses rewriting logic as a formal language to coordinate heterogeneous systems. It relies on the rewriting implementation in Maude [119] for execution.

3.3.2 Goal

We identified three different *goals* of coordination approaches: **simulation**, **execution**, and **formal verification**, see Figure 3.7. A coordination approach can have one or multiple goals, which we will now describe in detail.

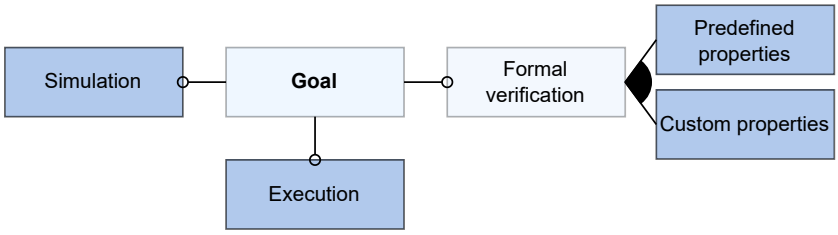


Fig. 3.7: Goal feature

Simulation The most common goal of coordination approaches is a coordinated simulation of multiple systems or behavioral models representing an entire system. A coordinated simulation is useful because the composition of multiple systems can lead to unexpected behavior, often called *emergent behavior* [49]. Emergent behavior can then be uncovered during simulation before deploying and using the system in a production environment. Co-simulation approaches such as DACCOSIM [44, 59] or MECSYCO (Multi-agent Environment for ComplexSYstem CO-simulation) [32, 33] are examples of approaches that have coordinated simulation as their goal. Simulation can also be a goal of coordination frameworks, such as described in Paper B, Paper D, and B-COoL [189]. Even recent ADLs such as MontiArc [67] provide simulation capabilities.

Execution In contrast to only simulation, some coordination approaches aim to provide a coordination infrastructure for system execution upon deployment. For example, Linda provides a virtual shared memory with read and write operations to coordinate systems.

Furthermore, the recently developed polyglot coordination language *Lingua Franca* provides determinism guarantees during distributed execution [112]. Thus, it eliminates typical coordination concerns faced during concurrent *execution*.

Formal verification Another goal of coordination approaches, especially for safety-critical systems, is formal verification of the coordinated system.

We encountered coordination approaches that validate *predefined properties*, while others support *custom properties*. For example, the ADL Wright [7] can check *deadlock freedom* of system interactions. My work in Paper B and Paper D supports custom properties written in temporal logic such as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL).

3.3.3 Properties

We identified several recurring properties when analyzing coordination approaches. Figure 3.8 depicts the resulting *properties* feature, which we will now explain in detail.

Domain Coordination approaches operate in different domains, like the categorization of co-simulation approaches described in [63, 107]. First, if a coordination approach is part of the *Discrete Event* (DE) domain, it allows discrete but not continuous

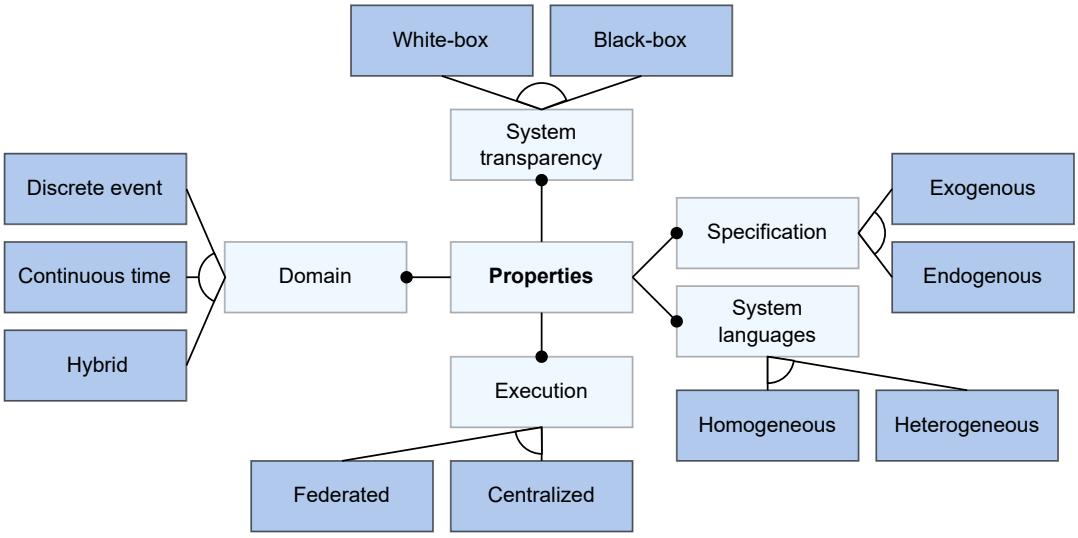


Fig. 3.8: Properties feature

state changes [63]. Typically, coordination frameworks and ADLs operate in the DE domain since variables change their values discretely during execution, i.e., immediately change between states.

Second, coordination approaches in the *Continuous Time* (CT) domain support systems to have a state that evolves continuously over time. Often, systems are given by differential equations, which induce continuous variable changes, such as physical systems involving springs and dampeners, see [63].

Third, the *hybrid* domain is a mix of the DE and CT domains that occur, for example, when simulating cyber-physical systems. In a hybrid domain, systems should coordinate where some change their state discretely (DE domain) while others evolve continuously over time (CT domain). Two strategies deal with hybrid systems: either adapt the systems from the CT to the DE domain or vice versa [63].

System transparency Some coordination approaches require constituent systems to be entirely transparent (*white-box*), while others only require a fixed interface (*black-box*). Usually, black-box approaches are needed to protect *intellectual property* (IP) when different companies want to work together without sharing all the details concerning their systems. For example, DACCOSIM only requires each system to conform to the FMI specification for Co-Simulation.

Specification We distinguish between *endogenous* and *exogenous* specification of coordination, as described for coordination languages in [13]. For example, when using the coordination language Linda [34], one uses coordination-specific operations such as out and in inside system programs to write to and read from the shared tuple space, respectively. Thus, Linda is *endogenous* since one must modify the behavior of the systems to add coordination.

On the other hand, *exogenous* coordination approaches keep coordination aspects outside of systems. However, systems must still be created with coordination in mind to expose possible coordination points. For example, B-COoL [189] uses language behavioral interfaces to define events, which can then be used to specify *exogenous* coordination rules outside the systems. Endogenous coordination approaches tend to mix computation and coordination [13].

Table 3.1: Approach classification

Feature \ Approach	DACCOSIM	Linda	MontiArc	Paper D
Foundation				
Syntax	Architecture based	Name based	Architecture based	Dedicated elements
Mechanism	Data-Exchange	Data-Exchange	Data-Exchange	Event-Scheduling, Data-Exchange
Expressiveness	Predefined	Predefined	Predefined	Predefined
MoC	Causal	Causal	Logical, Causal	Logical
Formality	Semi-formal	Semi-formal	Semi-formal	Formal
Goal				
Simulation	+	-	+	+
Execution	-	+	-	-
Formal verification	-	-	-	Custom properties
Properties				
Domain	Hybrid	Discrete event	Discrete event	Discrete event
Comp. transparency	Black-box	White-box	White-box	White-box
Specification	Exogenous	Endogenous	Endogenous	Exogenous
Comp. languages	Homogeneous	Homogeneous	Homogeneous	Heterogeneous
Execution	Federated	Federated	Centralized	Centralized

System languages Coordination approaches typically only support *homogeneous* systems. For example, coordination languages, such as Lingua Franca [111], currently only support coordination if each system is given in the same programming language.

However, coordination frameworks allow *heterogeneous* systems, i.e., systems specified in different behavioral modeling languages. For example, Paper B and Paper D support all modeling languages that one can describe using rewriting semantics.

Execution A coordination approach is realized using either a *centralized* or *federated* execution. In a *centralized* execution, all systems are executed by one engine, which enforces coordination. For example, in Paper B, the behavior of all system specifications is composed into a global behavior specification, which is then executed by a graph transformation or term-rewriting tool.

On the other hand, in a *federated execution*, systems run independently and only coordinate using a shared infrastructure through predefined connections. Federated execution can still happen on the same physical machine, i.e., it does not mean that systems must be *distributed* across different physical machines.

For example, the coordination language Lingua Franca [111] supports federated execution on one machine by providing a runtime infrastructure. DACCOSIM [59] goes one step further and allows a *distributed* federated execution.

3.4 Classification

Table 3.1 shows the classification of four different coordination approaches using the feature model. Each approach comes from a different category: co-simulation (*DACCOSIM* [44, 59]), coordination language (Linda [34, 35]), ADL (MontiArc [67]), and coordination framework (Paper D).

In addition, I classified the following approaches: Paper B, Ptolemy [49, 158], Wright [6, 7], CommUnity [55, 145], Metropolis [19], UMoC++ [124], Lingua Franca [111,

112], Reo [14], BIP [23, 25], MECSYCO [32, 33], CoSim20 [107, 108], B-COoL [188, 189], Manifold [16, 156], and ForSyDe [170, 171]. Due to space constraints, we cannot show all classifications, but they are available in the artifacts [185].

In Figure 3.9, we compare Paper D and MontiArc with Linda on the left and DACCOSIM on the right. The figure shows Venn diagrams comparing the features of each approach, where each part states which features are contained. In addition, the numbers state the number of features contained in each area.

For example, when looking at the Venn diagram on the left in Figure 3.9, the 1 in the circle part at the bottom describes that MontiArc has one *original* feature, while the 4 in the intersection of all approaches in the middle declares that they all have four features in *common*, see Table 3.1.

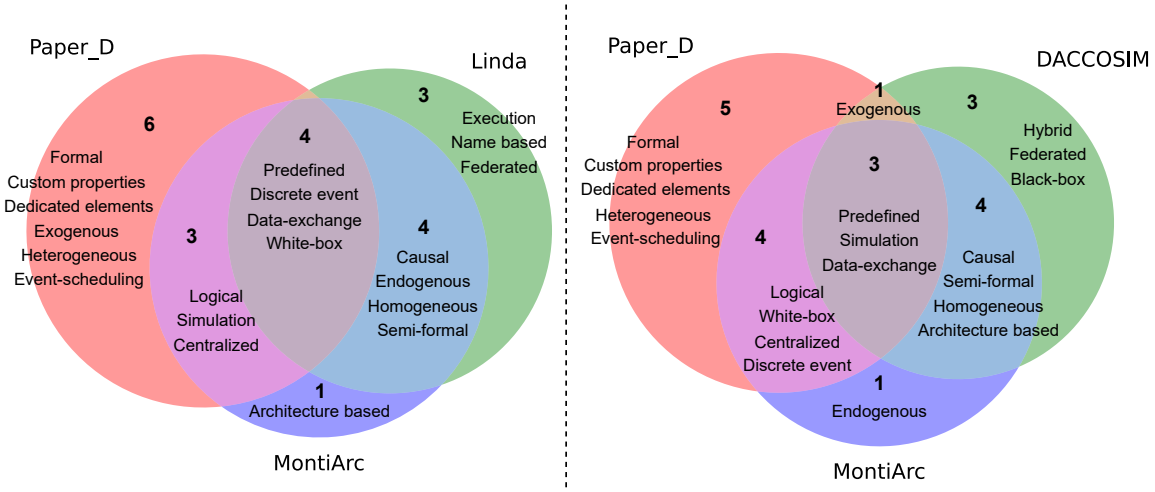


Fig. 3.9: Feature comparison: Paper D, MontiArc, and Linda/DACCOSIM

The Venn diagram on the left in Figure 3.9 shows that Linda has three original features compared to Paper D and MontiArc: execution as a goal, a name-based syntax, and a federated execution. Furthermore, one can see that coordination frameworks, i.e., Paper D (the diagram is similar for B-COoL [188, 189]), greatly differ from ADLs and coordination languages. Coordination frameworks have multiple original features, such as supporting heterogeneous system languages and formal verification using custom properties.

Similarly, on the right in Figure 3.9, Paper D and MontiArc are compared with the co-simulation approach DACCOSIM. Paper D and DACCOSIM do not have many features in common, probably due to their different goals as a coordination approach. Co-simulation approaches like DACCOSIM have original features, such as a federated execution, hybrid domain, and black-box system transparency.

Venn diagrams give a good approximation of the similarity between three specific coordination approaches. However, they do not tell the whole story since each feature is given equal weight, which is not always adequate. For example, a difference in the *domain* feature is more profound than most other differences. We stick to this simple weighting throughout this thesis since the importance of features can be subjective. However, one can adapt the analysis using the provided data and scripts [185].

3.5 Findings

This section describes my findings of applying the feature model to the different coordination approaches. First, we summarize the common features of each coordination category. Second, we discuss general observations I made while classifying the approaches. Finally, we cluster the approaches based on their feature similarity and discuss the results.

3.5.1 *Common Features*

ADLs primarily have *simulation* as their goal, while a small subset allows predefined properties to be verified. As expected, the coordination syntax of all ADLs is architecture-based, using components with ports and connectors between them. All ADLs operate in the discrete event domain. Furthermore, components in ADLs are white-box and homogeneous, and coordination specification is usually exogenous, i.e., part of the connectors. An exception is MontiArc, where ports can be marked as *synchronized*, only allowing synchronous interactions through a port; that is, the coordination specification is endogenous and partly present inside components.

Coordination languages can have simulation and execution as their goal while operating in the discrete event domain. Furthermore, coordination syntax is also most likely architecture-based, i.e., most coordination languages use the idea of subsystems, ports, and connectors/channels between them. In addition, the systems in coordination languages are white-box and homogeneous but have both endogenous and exogenous coordination specifications. Notably, some coordination languages offer a federated execution that ADLs do not support.

As the name suggests, all **co-simulation** approaches have merely simulation as their goal. Like ADLs and coordination languages, coordination syntax is architecture-based. However, compared to the other categories, they operate in the hybrid domain, facilitating data exchange between black-box systems. The systems can be homogeneous or heterogeneous by wrapping them in specific formalisms such as DEVS in the case of MECSYCO. Typically, execution is federated, while some approaches, such as DACCOSIM, even allow for a distributed execution.

Coordination frameworks have simulation and formal verification as their goal. Coordination syntax is diverse but usually applies event scheduling in the discrete event domain. Furthermore, coordination specification is exogenous to support heterogeneous systems, which are white-box. In contrast to coordination languages and co-simulation approaches, execution is centralized. In addition, my coordination frameworks (Paper B and Paper D) facilitate formal verification of custom properties, which is a rare feature.

3.5.2 *General Observations*

Architecture-based coordination syntax is widely used. Remarkably, coordination approaches from all categories use concepts such as components, ports, and connectors to express coordination. Although the precise coordination details may differ across

various approaches, there seems to be a broad consensus on accurately representing the system architecture for coordination.

Formal verification is uncommon. Most of the studied coordination approaches have simulation as their goal, while formal verification is uncommon. This could be attributed to the problem of formally representing heterogeneous systems engaged in coordination and the rapid growth of state spaces when multiple systems are involved.

Tool Quality Many tools accompanying the different coordination approaches are no longer retrievable or executable on current operating systems. This makes it challenging to investigate such contributions in detail since the features of approaches evolve and are spread out between multiple publications. We advocate for open-source repositories and more standardized tool formats based on modern technologies, such as containers that handle dependency management and simplify execution.

3.5.3 *Feature Clusters*

We further analyze the classification of the approaches mentioned in section 3.4 to determine how they compare across and within the four categories. We clustered the approaches to see similarities and differences by just looking at the feature data of each approach without considering to which category they belong. The data and scripts to reproduce my analysis are available in [185]. We apply a standard clustering algorithm using Jaccard distance to measure the similarity of the feature sets. Jaccard distance is the one-complement of Jaccard similarity for two sets, defined as the ratio of their intersection to their union [106]. It is a widely used and reasonable dissimilarity measure for sets [106].

Figure 3.10 shows a scatter plot with approximate positions of each approach derived solely from the distances between them. It provides a bird's-eye view of the similarity or dissimilarity between all the studied coordination approaches—this is in contrast to the previous Venn diagrams, which only compare three approaches. Furthermore, it highlights the clustering results by coloring each data point. We used clustering parameters to reduce the number of unclustered approaches while ensuring that not everything becomes part of one cluster (see [185] for the exact parameters).

The clustering algorithm finds three clusters, including all but one approach. **Cluster 1** contains only coordination frameworks (Paper B, Paper D, and B-COoL, while **Cluster 2** consists of a mix of coordination languages and ADLs, such as Linda and Lingua Franca, as well as Wright and MontiArc, respectively. **Cluster 3** includes three co-simulation approaches, but no cluster contains Ptolemy.

We interpret cluster 1 as representing the coordination framework category. Similarly, cluster 3 is the co-simulation cluster, showing that according to the feature model, co-simulation approaches are similar and have original features not found in the other categories. Cluster 2 contains coordination languages and ADLs. Even if the maximum distance between approaches is reduced to be considered part of the same cluster (clustering parameter), one cannot separate the approaches into two categories. Thus, we conclude that coordination languages and ADLs share more features than, for instance, co-simulation approaches or coordination frameworks.

As the only approach, Ptolemy remains unclustered. Although we would classify Ptolemy as a coordination framework, it includes features from all categories and thus

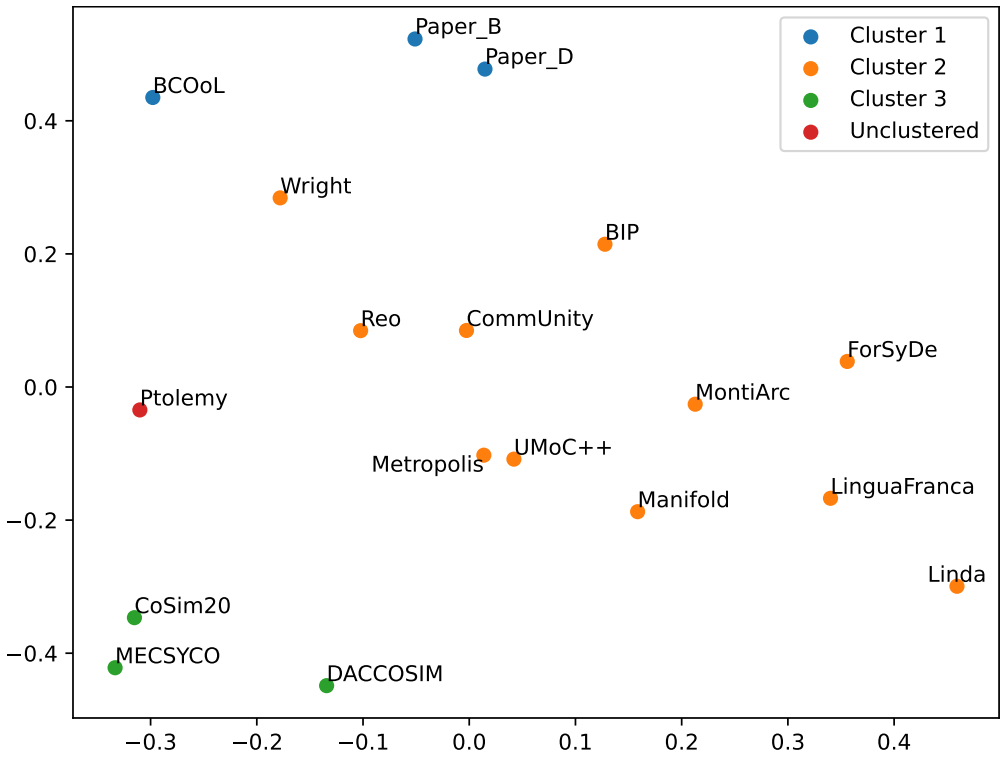


Fig. 3.10: Feature distance and clustering of approaches

remains unclustered.

3.6 Conclusion

The primary contribution of this chapter is the introduction of a new *feature model* for coordination approaches. This model enables the categorization and comparison of approaches across previously isolated categories. Furthermore, it gives a thorough overview of the coordination literature, highlighting its scope and underlying motivations.

Furthermore, we classified 17 coordination approaches using the feature model and published the results [185]. Analyzing the resulting data, we identified characteristic and unique features in each category. As overarching insights, we find that architecture-based coordination is widely used, formal verification is not well supported, and coordination frameworks do not support data exchange.

Additionally, we clustered the approaches based on their similarities, revealing that ADLs and coordination languages are similar from the coarse-grained perspective of the feature model, while the other categories remain distinct. This aligns with my notion that these two categories have recently begun to converge.

"If I have seen further it is by standing on the shoulders of Giants"

— Isaac Newton

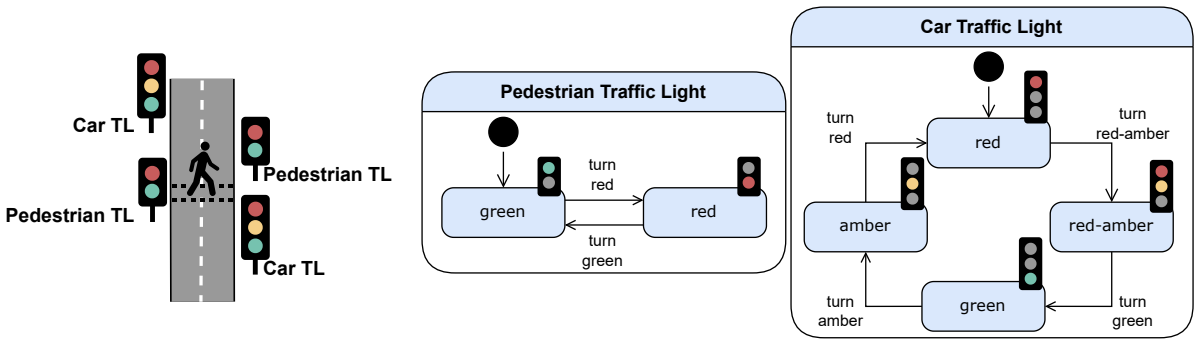
CHAPTER 4

RELATED WORK

In this chapter, we present the related work relevant to our coordination and verification approach for heterogeneous systems. We provide a detailed discussion of one representative from each coordination category, namely Lingua Franca [111, 112], MontiArc [67], DACCOSIM [44, 59], and B-COoL [188, 189]. In addition, we discuss Ptolemy [49, 158] since it is an approach with unique features as identified by our classification in chapter 3. These approaches fall into the discrete event or hybrid domain identified in Figure 3.8 of chapter 3. Furthermore, we mention specialized approaches that handle certain combinations of behavioral models.

4.1 Discreve Event Approaches

We examine three discrete-event-based approaches: Lingua Franca (a coordination language), MontiArc (an architecture description language), and B-COoL (a coordination framework). We utilize a straightforward running example for all three approaches to provide a more practical explanation. The goal is to model, simulate, and possibly verify a traffic system for a pedestrian crossing, as illustrated in Figure 4.1. Figure 4.1a gives an overview of a pedestrian crossing controlled by traffic lights.



(a) Overview **(b)** Traffic light models

Fig. 4.1: Running example: Pedestrian crossing

Figure 4.1b illustrates two statecharts representing the possible states of both traffic lights. Certain state combinations, however, must be disallowed to prevent unsafe situations, such as both lights showing green simultaneously. Therefore, these models require coordination to ensure safe traffic management and cannot be executed

independently. The following paragraphs describe how each proposed approach achieves this coordination.

4.1.1 Lingua Franca

Lingua Franca (LF) [111, 112] is a modern coordination language based on the formal model of reactors, first described in [113], that aims to deliver determinism in concurrent systems. LF is a coordination language where reactors are defined in a host language (C, C++, TypeScript, and Python) combined using LF as a *bridge language*. A reactor is a component with input and output ports connected to other ports of the opposite type. Ports serve as an additional indirection layer, making reactors modular and reusable. We observe a significant similarity between Lingua Franca and ADLs, such as MontiArc. One then defines the reactions to data arriving at each input port, which can lead to producing data at output ports that trigger further reactions. LF ensures the reaction graph is acyclic to avoid infinite loops. The ideas of ports and components overlap with the three Architecture Description Languages (ADLs) building blocks: components, connectors, and architectural configuration discussed in chapter 3.

A system consisting of two reactors, designed to implement the running example, is illustrated in Figure 4.2. There is one reactor for the crossing, containing one for the car traffic light and one for the pedestrian traffic light. In this implementation, the car traffic light controls the pedestrian traffic light by sending notifications through its output port, which is connected to the pedestrian traffic light's input port. The full executable implementation, including the reactions written in TypeScript, can be found in [185]. The implementation is textual, but LF simultaneously renders the diagram shown in Figure 4.2 (without blue annotations) that gives a succinct overview of the implementation.

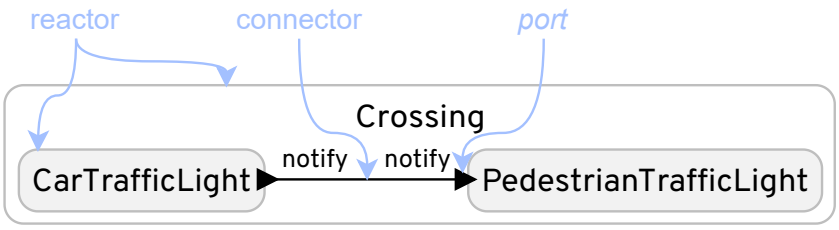


Fig. 4.2: Running example in Lingua Franca (generated by LF)

LF's goal is simulation and, most importantly, execution, which fits its semi-formal nature. The coordination language does not allow for heterogeneous reactors (components), i.e., each reactor must correspond to the same programming language. LF comes with mature tools for development and execution [180] but does not support formal verification. Thus, our feature model categorizes it as semi-formal [111]. Our approach aims to coordinate components represented by heterogeneous behavioral models, whereas LF coordinates components provided as executable programs. Like our approach, LF supports data exchange and a sophisticated model of time.

4.1.2 MontiArc

MontiArc [67] is an architecture description language (ADL) for modeling and simulating software architectures developed in the language workbench MontiCore [96, 97, 169]. It allows users to specify components, ports, and connectors between ports, which is typical for ADLs.

Figure 4.3 shows an overview of the implementation of the running example. Compared to Lingua Franca, this overview is not generated automatically. MontiArc is textual, and the implementation of the running example is given in Listing 4.1. MontiArc directly supports automatons to provide the behavior of components as shown by the DSL in Listing 4.1 for the `PedestrianTrafficLight` component.

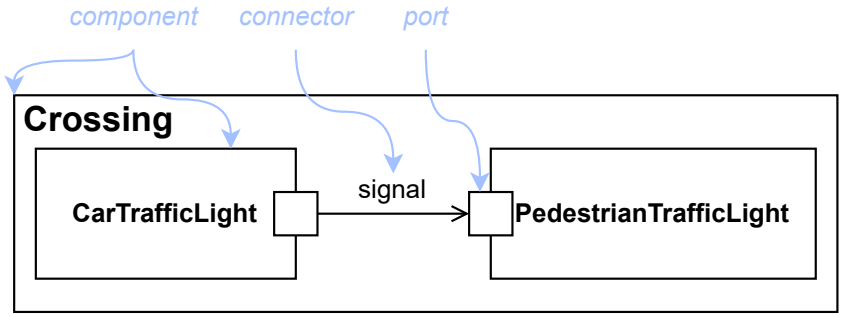


Fig. 4.3: Running example in MontiArc (overview)

```
1  component Crossing {
2    CarTrafficLight ctl;
3    PedestrianTrafficLight ptl;
4
5    ctl.signal -> ptl.signal;
6  }
7  component CarTrafficLight {
8    port
9      <<sync>> out Signal signal;
10   automaton {
11     initial state Red;
12     state RedAmber;
13     state Green;
14     state Amber;
15
16     Red -> RedAmber;
17     RedAmber -> Green / { signal = Signal.RED; };
18     Green -> Amber;
19     Amber -> Red / { signal = Signal.GREEN; };
20   }
21 }
22 component PedestrianTrafficLight {
23   port
24     <<sync>> in Signal signal;
```

```
25    automaton {
26        initial state Green;
27        state Red;
28
29        Red -> Green [signal == Signal.GREEN];
30        Green -> Red [signal == Signal.RED];
31      }
32  }
```

Listing 4.1: Running example in MontiArc

Like the Lingua Franca implementation, we let the `CarTrafficLight` control the `PedestrianTrafficLight` by sending notifications through the connection `signal`, which connects their two ports. Using synchronous ports ensures that both automatons change their state in one step. The syntax in lines 17-18 means these transitions are taken when a corresponding signal (Enumeration of green or red) is received. The implementation of the running example is executable and can be found in [185].

Like LF, MontiArc is semi-formal and intended for simulation rather than formal verification. However, because MontiArc operates at the model level, it is theoretically more suitable for formal verification. Currently, MontiArc only supports automata by default when defining component behavior. Like LF, it is an intriguing, well-maintained approach that is open-source and accessible online[1].

4.1.3 B-COoL

B-COoL [188, 189] is a coordination framework similar to our approach. The goal is to define the coordination of behavioral models to produce a single integrated model for simulation. Modelers using B-COoL define behavioral interfaces for each behavioral language, enabling the specification of coordination between models. To execute these models with the defined interactions, they are transformed into Clock Constraint Specification Language (CCSL) [9] models. B-COoL is implemented in a plugin for the GEMOC studio [61], which supports model execution and debugging.

Implementing the running example in B-COoL is straightforward because a behavioral interface for statecharts has already been established in [188]. This interface supports the coordination of transition occurrences, as shown in Listing 4.2. Unfortunately, the available tooling is outdated, preventing us from providing a readily executable implementation of the running example.

```
Operator CoordinationOfEvents(dse1 : sc1::occurs, dse2 : sc2::occurs)
    CorrespondenceMatching: when (dse1.id = dse2.id)
    CoordinationRule: RendezVous(dse1, dse2)
end operator
```

Listing 4.2: Synchronization of events in B-COoL

The B-COoL operator ensures synchronized execution (`RendezVous`) of two domain-specific events (`dse1` and `dse2`) coming from the two different statecharts (`sc1` and `sc2`) if their IDs are identical. The events correspond to the transitions in the statecharts

[1] https://github.com/MontiCore/montiarc

(`sc1::occurs` and `sc2::occurs`), which means that the rule creates a bidirectional dependency between them. Neither transition can occur unless the other is ready to execute simultaneously.

For the running example, we want to synchronize `turn red` from the pedestrian traffic light with `turn green` from the car traffic light and vice versa. If we ensure their IDs are identical, the operator in Listing 4.2 is sufficient to synchronize both pairs of transitions.

B-COoL, a coordination framework, is the closest to our approach. It employs a DSL to define coordination at the language level. Coordination operators with the correct correspondence matching specify how all instances of particular models interact. This method is beneficial if all instances of a model coordinate similarly, but it is less effective if, for example, three finite state machines need to be coordinated differently. The main difference between B-COoL and our approach is that we are focused on formally verifying coordinated models. In contrast, B-COoL is designed for simulation and does not support formal verification. Moreover, our approach includes coordination with data and time, as outlined in Paper D, which B-COoL does not support. B-COoL highlights data exchange as a key area for improvement in future work [188].

4.2 Specialized Coordination Approaches

The approach in [104] describes consistency checking for sequence diagrams and statecharts by transforming them to the process algebra CSP [79]. Similarly, [197] checks consistency between sequence diagrams and state machines using Colored Petri Nets (CPN) [86]. The consistency check between sequence diagrams and state charts ensures that the interactions shown in the sequences align with the possible behaviors for each object as defined in the state charts.

These specialized approaches can be considered precursors to our method, as they adhere to the same methodology. To verify the consistency of behavioral models, they are transformed into a general formalism that supports formal verification, such as CSP, CPN, and graph transformation or rewriting logic, in our case. By employing this general formalism, one can examine properties of interest that span heterogeneous modeling languages [50].

4.3 Hybrid Approaches

Hybrid approaches are used to simulate cyber-physical systems. In a cyber-physical system, software and hardware components are deeply integrated. For example, an industrial control system uses physical sensors as inputs (hardware) to save data and then compute actions (software), which results in outputs that control physical machines or robots (hardware). The physical dynamics are often described using differential equations [63, 105]. The system state described by a differential equation evolves continuously over time, while purely software systems have a discrete state space (see chapter 3).

Our work does *not* aim to address the complexities inherent in hybrid systems. Nevertheless, because the challenges involved in coordinating systems are similar, it is

worthwhile to investigate them to a certain extent.

4.3.1 Ptolemy

Ptolemy builds on an actor-oriented model [4, 78] where each actor has input and output ports that can be connected. This is similar to Lingua Franca and MontiArc, which also have components with ports that can be connected. Furthermore, actors in Ptolemy can be hierarchical, meaning that one actor can be composed of multiple connected actors. The main idea behind Ptolemy is to employ a unified abstract syntax that can represent a variety of actor-oriented models. Then, complex systems can be built by hierarchically composing these models using different models of computation (MoCs). A MoC describes execution and communication semantics for connected actors, which makes these actors locally homogeneous in terms of semantics. Distinct MoCs can be hierarchically combined to allow for heterogeneity [105]. For instance, in the *Dataflow* MoC, an actor is triggered when input data becomes available and then consumes this input and produces output data using some computation. Other MoCs, for example, the *RendezVous* MoC allows for synchronous rather than asynchronous communication. A detailed overview of Ptolemy and its supported MoCs can be found in the Ptolemy book [158]. Furthermore, Ptolemy offers a tool implementation called Ptolemy II [159], which can be used to experiment with Ptolemy.

We classify Ptolemy as a coordination framework for hybrid systems. Our approach supports various MoCs available in Ptolemy where actors operate based on discrete states, such as *Dataflow*, *RendezVous*, and *Discrete Event*. However, we do not support the *Continuous Time* MoC, in which differential equations describe models. The *Continuous Time* MoC is typically used for co-simulation of cyber-physical systems.

4.3.2 DACCOSIM

DACCOSIM (Distributed Architecture for Controlled CO-SIMulation) employs the Functional Mock-up Interface (FMI) [138] as its component standard, which enables it to integrate any components that adhere to the FMI. Thus, each component defines its inputs and outputs, which are then coordinated by the master algorithm, i.e., the central orchestrator implemented in DACCOSIM [51, 59]. Each component is given by a Functional Mock-up Unit (FMU) represented by a metadata file defining its inputs and outputs, and a binary executable to run the component, typically written in C. Although FMI-based co-simulation approaches typically rely on a centralized orchestrator, DACCOSIM distinguishes itself by enabling distributed co-simulation [44, 59].

DACCOSIM and other co-simulation approaches employ a strategy similar to coordination languages, ADLs, and coordination frameworks in managing heterogeneous components. This approach involves defining a minimal interface for uniform interaction with each component. Designed to work with black-box components (binary executables), co-simulation preserves intellectual property by minimizing transparency. This *low* abstraction level highlights a focus on simulation rather than formal verification, which becomes particularly challenging in the absence of models for the constituent components.

"There are no solutions. There are only trade-offs."

— Thomas Sowell

CHAPTER 5

CONCLUSION

In this chapter, we start by summarizing the key contributions of this thesis from two perspectives. First, we revisit the research questions and provide direct answers, emphasizing the contributions made in the accompanying papers. Second, we summarize the contributions of each paper individually. Then, we evaluate our work and discuss limitations. Finally, we conclude the thesis and suggest directions for future research.

5.1 Research Questions revisited

Figure 5.1 illustrates the relationship between the research questions and the papers that address them, along with the extent to which each question is covered. While it was impossible to thoroughly answer all the research questions within the scope of my PhD project, each one was addressed substantially. Moreover, the contributions specific to BPMN (Paper C and Paper E) provide a possible roadmap for improving the general heterogeneous approach (Paper A, Paper B, and Paper D). In the subsequent paragraphs, we will discuss how these contributions address the research questions.

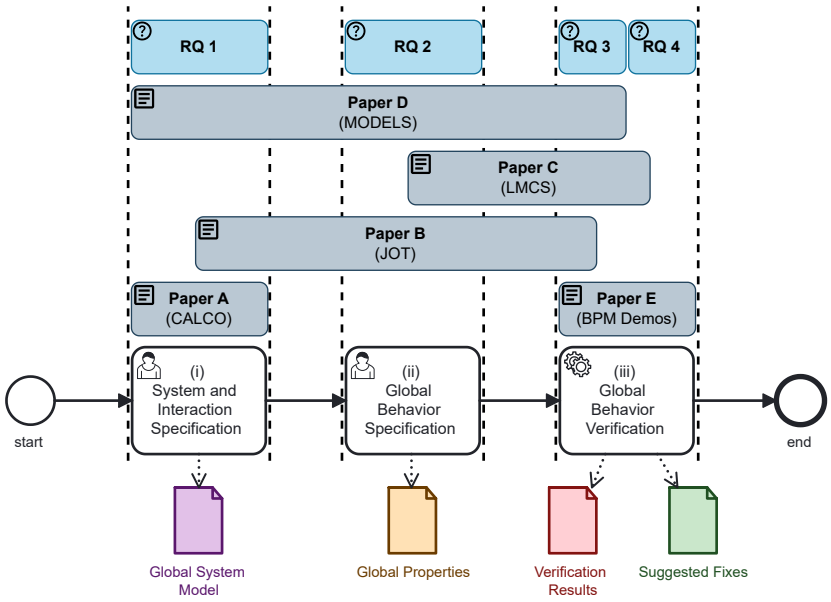


Fig. 5.1: Research questions and corresponding contributions

RQ 1 How can we specify *coordination* between heterogeneous behavioral models, including real-time features and data exchange?

Paper A addresses **RQ 1** by presenting an approach for coordinating heterogeneous coalgebras. This approach provides a theoretical foundation for understanding heterogeneous SOS rules, which form the basis for coordinating heterogeneous models. In earlier work, composition was limited to coalgebras of the same type. Paper B exemplifies this, providing a domain-specific language (DSL) to define coordination between heterogeneous models where the semantics for each model and the defined coordination are represented in a formal specification (for example, graph transformation). However, Paper B focuses exclusively on synchronous interactions and does not address real-time features or data exchange, which motivated the development of the new approach in Paper D. Paper D also uses a DSL to define coordination between heterogeneous systems and is similarly based on a formal specification (for example, rewriting logic). With the introduction of language adapters, which include data transformation functions and a general real-time approach, Paper D provides a more comprehensive response to **RQ 1** compared to Paper B. In addition, Paper D uses a broker architecture, i.e., a central component, to maintain separation between the models, whereas Paper B relies on merging heterogeneous rules to achieve coordination. In contrast to our approach described in Paper D, other methods are more intrusive and typically lack support for data exchange and real-time capabilities.

RQ 2 How can we *specify global behavioral requirements*, i.e., properties spanning heterogeneous models?

To specify properties that involve multiple heterogeneous models, atomic propositions are first defined for each model. These propositions are then integrated into a global property using Boolean and temporal operators. For defining atomic propositions, the *state structure* of each behavioral modeling language must be defined, as discussed in Paper B (snapshot metamodels) for heterogeneous scenarios and in Paper C for BPMN. To the best of our knowledge, no other existing approaches enable the specification of global behavioral properties without first converting each model into a standard modeling language, where atomic propositions are then defined.

Furthermore, modelers are often unfamiliar with temporal logic, and thus, there is often a knowledge gap between modeling and verifying the created models. In Paper C, we demonstrate how atomic propositions can be intuitively defined for BPMN through a graphical *concrete syntax* based on the underlying model. Insights from Paper C demonstrate that providing a *concrete syntax* for defining atomic propositions tailored to each modeling language in a heterogeneous environment is technically feasible. This approach helps to bridge the gap between modeling and verification.

RQ 3 How can we *verify global behavior* of heterogeneous models, including real-time features and data exchange?

Paper C demonstrates how to verify BPMN models by defining their formal semantics through graph transformations. To verify heterogeneous models, one must represent the semantics of all models within a unified formalism, as Paper B explains. While Paper B utilizes graph transformation and rewriting logic, benefiting from the availability of mature tools, our approach is not limited to these specific

formalisms. Furthermore, to address data exchange and real-time features, Paper D introduces several new concepts. We propose using a canonical data model as a shared intermediary format for data exchange. In addition, translation functions are provided to convert data between the data models of each language and the canonical data model. For real-time features, we build on the principles of Real-Time Maude [147] and adapt them to handle multiple heterogeneous models. Currently, we are not aware of any existing approaches that support the verification of global behavior in heterogeneous settings—even in the absence of data exchange—which has been identified as a key area for improvement in B-COoL [188].

RQ 4 How can we clearly *present violated properties* to a user and *suggest fixes*?

Presenting violated properties for heterogeneous systems poses a challenge because different parts may use distinct modeling languages. Consequently, counterexamples generated during verification must be translated back from the underlying formalism into the respective modeling languages that each part of the system uses. This issue was not addressed directly, but we try to stay as close as possible to the used languages when implementing their semantics, for example, in graph transformation systems (Paper B) and rewriting logic (Paper D). However, in Paper C and Paper E, we address the issue of presenting violated properties for BPMN. In Paper C, we highlight erroneous parts of the BPMN model based on predefined soundness properties. In Paper E, we go one step further and present an interactive visualization of the entire counterexample, allowing users to observe how a property is violated. The visualization is based on the *concrete syntax* introduced for atomic propositions in **RQ 2**. In future work, this visualization should be extended to heterogeneous behavioral models to fully address the first part of this research question. Paper E tackles a key challenge faced by BPMN users who are not familiar with formal analysis: *consumability*, that is, making property violations more understandable, as highlighted in [52]. We think this challenge persists—and may even become more pronounced—when transitioning from BPMN to a heterogeneous setting.

Proposing fixes for all possible temporal property violations is highly challenging, even when working within a single behavioral language. This complexity increases significantly in heterogeneous environments. Therefore, addressing this challenge is mainly left for future work. However, in Paper E, we demonstrate that automatic fixes can be suggested when restricted to specific predefined properties, such as BPMN soundness, and applied to a single behavioral language (BPMN). The developed tool analyzes BPMN soundness properties, visualizes violated properties, and suggests fixes applicable directly in the modeling environment, like *quick fixes* in Integrated Development Environments (IDEs).

5.2 Contributions

Figure 5.1 illustrates the relationship between the research questions, contributions from the papers, and the activities in the coordination and verification process. Earlier, we detailed how these contributions align with the research questions. In the following section, we will summarize the contributions of each paper individually. While some

overlap with previous discussions is expected, the focus here shifts from addressing the research questions to examining each paper's specific contributions, as shown in Figure 5.2.

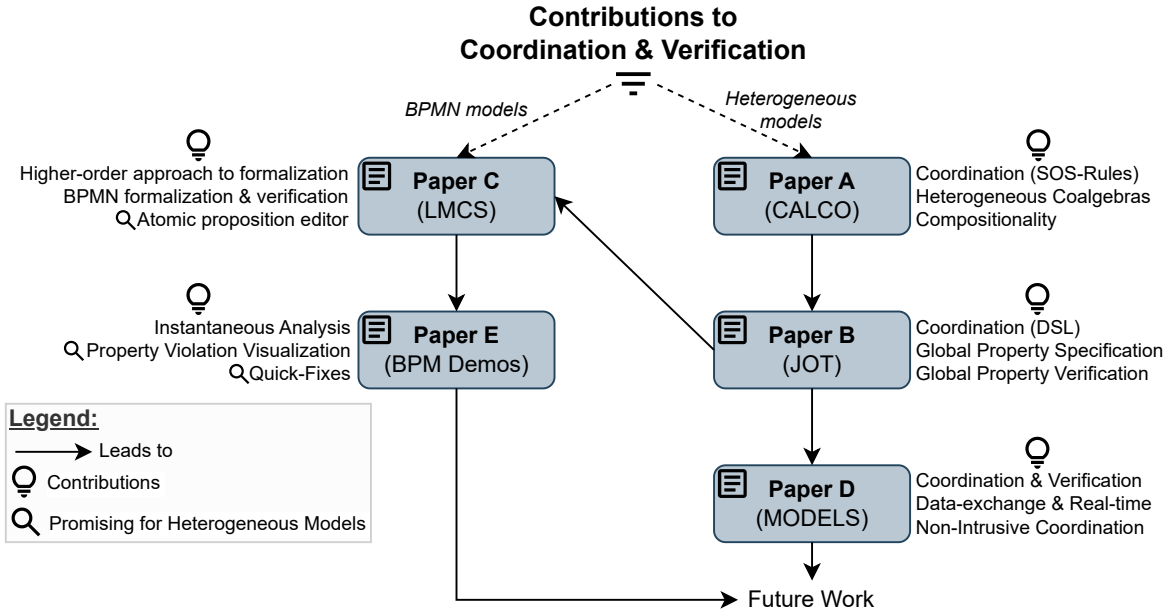


Fig. 5.2: Contributions summarized

The main contribution of **Paper A** [95] is the formal structure for describing the composition of *heterogeneously typed* coalgebras (SOS rules for coordination) and the proof for preserving observational equivalence in this many-sorted approach to enable *compositionality*.

Paper B [98] proposes a novel approach for behavioral consistency management, i.e., coordination and verification of heterogeneous systems. Before this work, there was no support for specifying and verifying global properties in heterogeneous systems. Our approach defines interactions (coordination) between different models using a domain-specific language (DSL). The DSL is based on the *state-changing elements* identified in each behavioral modeling language, which together form a minimal *behavioral interface*. The formalization using heterogeneous SOS rules in Paper A provides a theoretical underpinning for this DSL. Global properties are defined based on the *state structure* of each behavioral language (snapshot metamodels), which specifies how global states are arranged. These properties can be verified using any formalism capable of coping with the semantics of each behavioral model, realizing the defined coordination, and providing verification capabilities. Examples of such formalisms include graph-rewriting or term-rewriting, as discussed in Paper B.

Paper C [100] explains how we implement a complex modeling language (BPMN) in our approach to coordinate it as described in Paper B. BPMN is widely used in the industry and has sophisticated semantics that have not been fully formalized. By employing a novel higher-order transformation approach, we provide a comprehensive formalization of BPMN that covers the key elements commonly used in practice and supported by industrial process execution engines. Our formalization includes unique BPMN elements and offers broader coverage than most existing formalizations.

Moreover, we support defining and verifying behavioral properties, thereby supporting steps **(ii)** and **(iii)** of the overall process. Additionally, we provide a tool[1] that demonstrates the practical application of our approach. Unlike Paper B and Paper D, our tool focuses solely on BPMN, meaning it operates in a homogeneous setting rather than a heterogeneous one. However, like Paper E, it provides concrete evidence that our approach can handle complex modeling languages, informing and laying the groundwork for future enhancements to the general framework. Using concrete syntax to define atomic propositions was well-received and should be incorporated into the broader framework.

Paper D [101] directly builds upon the concepts from Paper B. The resulting coordination framework presented in Paper D supports the coordination and verification of heterogeneous behavioral systems. Notably, compared to Paper B, it introduces support for **data exchange** during coordination. In addition, it adds the capability to handle **real-time features**, ensuring a consistent progression of time across different models. The approach remains **non-intrusive** due to a linguistic extension and language adapters separating the behavioral languages. Heterogeneous models can only interact with each other through an intermediate broker, which is introduced to maintain the separation of the languages and corresponding models. This differs from our approach in Paper B, where we merge multiple rules from heterogeneous models to achieve coordination. Data exchange is facilitated by introducing a canonical data model, as outlined in **RQ 3**. Additionally, real-time functionality is enabled by adopting the principles of Real-Time Maude [147] to a heterogeneous environment. In summary, our approach distinguishes itself from prior work by combining the specification and verification of global properties with coordination mechanisms supporting data exchange and real-time functionality.

Paper E [102] makes significant contributions to step **(iii)** in the coordination process by showing how behavior validation can be made **performant**, **comprehensible**, and **fixable** for one specific language, namely BPMN. The developed BPMN Analyzer 2.0[2] not only analyzes real-world BPMN models instantaneously (defined as < 500 ms) but also highlights violations directly within the modeler and provides quick-fix suggestions for BPMN soundness properties. Such instantaneous analysis is uncommon among existing analysis tools, and our approach is the first to incorporate automated suggestions for resolving detected violations. BPMN practitioners also appreciate the tool, especially the end-to-end user experience it offers. The tool is implemented for BPMN, not the heterogeneous case, but it illustrates key ideas that can potentially be applied in a general setting. In contrast to Paper C, Paper E focuses solely on checking BPMN soundness properties, rather than supporting arbitrary temporal logic properties. However, limiting the scope allows us to implement the quick-fix suggestions and instantaneous analysis described earlier. Additionally, counterexamples showing property violations are visualized interactively through tokens.

Although Paper C and Paper E focus exclusively on verifying BPMN models, they present several ideas that show significant potential for application in heterogeneous models, as illustrated in Figure 5.2. These concepts provide a strong foundation for future research in heterogeneous scenarios. A detailed discussion of future work and

[1] You can watch a demonstration of the tool on YouTube and try it online.
[2] The BPMN Analyzer 2.0 is available online. A demonstration can be found on YouTube.

limitations is presented at the end of this chapter.

5.3 Validation

An important aspect of constructive technology research, such as software engineering, is the final evaluation of the results. This evaluation assesses whether the artifacts effectively address the original research questions and how they compare to existing solutions. We follow the distinction by [157] between *validation* and *evaluation*. Evaluation happens empirically in a real-world industrial context, whereas validation happens in a controllable academic context. We did not evaluate our contributions as they are still in the proof-of-concept stage. However, our contributions were validated academically in the following ways.

First, our contributions to the coalgebra theory in Paper A rely on mathematical proofs to ensure validity. These proofs have undergone peer review in addition to the verification by the authors.

Second, the feature model described in chapter 3 resulted from a meta-analysis of the literature following PRISMA [152]. We documented each step in the analysis and published our raw data [185], including the analysis code, enabling full reproducibility of our results. Naturally, some dependence on the authors' interpretation and comprehension of the examined approaches is inevitable.

Third, in Paper C, we claim extensive BPMN feature coverage due to our new higher-order transformation methodology, which is proven by providing our implementation, the BPMN Analyzer. Furthermore, we evaluate our approach regarding *scalability* and *performance* by benchmarking it against *realistic* process models from other contributions, models with increasing complexity, and models with increasing size. The validation shows that our approach performs well and can most likely handle large process models found in the industry. In addition, in Paper D, we use the same benchmarks to evaluate our claim of *instantaneous analysis* by the BPMN Analyzer 2.0. In both papers, we provide the BPMN models and step-by-step instructions to reproduce our benchmarks. In Paper D, we claim comprehensible control flow analysis that suggests fixes for BPMN models. We point out the improvements to the user interface and new possibilities of interaction, however we did not validate or evaluate the usability of the BPMN Analyzer 2.0 in a user study.

Finally, the coordination framework we propose includes unique features, as detailed in the previous contributions section, which offer improvements compared to other existing approaches. We categorize it in our feature model in chapter 3 and compare it to other approaches in chapter 4. The proof-of-concept implementations demonstrate the ability of the framework to address practical problems (feasibility). Concretely, we analyze safety for two use cases of traffic management systems. In Paper B, we check a system guiding traffic at a T-Junction, and in Paper D, we analyze a system for a road-rail crossing. We also perform repeatable experiments to assess performance and scalability in Paper B.

5.4 Limitations

Hybrid systems Our approach does not cover hybrid systems, such as cyber-physical systems with continuous states. Because these systems have non-discrete state spaces, formal verification is even more challenging, and we would need to generalize our approach further. It is crucial to strike a balance between generalization and practical applicability. In this thesis, we build a coordination framework that effectively handles software systems. Recent surveys on co-simulation, which enables simulation of hybrid systems, can be found in [63, 174]. In addition, we investigated specific co-simulation approaches in chapter 3.

Exchange of complex-typed data Our approach in Paper D only applies to data typed using basic data types. In practice, systems often exchange structured data by serializing it into text-based formats (e.g., XML, JSON) or binary formats (e.g., Protocol Buffers, Apache Thrift), which our method can support. However, directly exchanging structured data without serialization and handling its diverse representations across different systems provides an opportunity to align our work with MDSE research on structural models, such as in [91, 177, 178].

5.5 Future Work

Supporting multi-ary coordination Our current heterogeneous approaches in Paper D only support binary coordination relationships. Thus, if one wants to define, for example, broadcasting to a list of receivers, one must define multiple interactions and include an intermediary component that implements the broadcast. To make it more convenient for users, we aim to support multi-ary coordination patterns such as broadcast and publish/subscribe out-of-the-box. This syntactic sugar will be helpful since many patterns in system architecture, for example, scatter-gather [80], are using these coordination patterns. In addition, Paper B already supports synchronization of multiple models but lacks data exchange and other capabilities later introduced in Paper D.

Model repair: Proposing resolutions for violated global behavioral properties Proposing resolutions for violated global properties, like the quick fixes for BPMN models in Paper E, proposes an interesting challenge for future work. How can one identify which behavioral model within the multi-model needs adjustment? If the model likely containing the error is known, how should it be modified? When dealing with structural models, this is typically called *model repair* [20, 21, 114] or *consistency restoration* [36, 38]. To my knowledge, no research specifically addresses model repair for heterogeneous behavioral models. However, related work exists for certain modeling languages [30, 53] and formalisms [22].

Implementation of an End-To-End framework Apart from Paper E, which focuses specifically on BPMN, our current contributions are research prototypes and, as such, do not address all aspects. Therefore, we aim to develop a complete end-to-end framework

suitable for industrial applications, enabling us to validate its practical utility and gain insights from real-world usage. The framework must include graphical modeling environments for each heterogeneous language, the specified DSL for coordination, and a concrete syntax for defining the atomic propositions required for verification. A streamlined method for defining the semantics of a modeling language should be provided, enabling seamless integration into the framework, such as through adapters as described in Paper D. As described in Paper C and Paper D, users should not need to understand the underlying operational semantics based on graph transformations or rewriting logic.

Industrial case study We aim to validate our approach within an industrial model-driven software engineering context with three key aspects in mind. First, it is crucial to demonstrate that our method can effectively handle the heterogeneity present in real-world use cases. Second, we need to verify that our approach scales well with the large models commonly found in the industry. Third, it is essential to ensure that our approach is usable and understandable by software developers, not just researchers in model-driven software engineering.

5.6 Final remarks

This thesis embodies our belief that Model-Driven Software Engineering should emphasize behavioral models in addition to in-depth research on structural models. As mentioned in chapter 1, software systems comprise both structure *and* behavior. We argue that the predominant attention to structural models stems partly from the greater heterogeneity of behavioral models, making them more challenging to verify.

To summarize, this thesis presents a novel coordination framework that enables the specification and verification of global properties in software systems. In contrast, prior approaches supported only the coordination and simulation of global behavior without offering verification capabilities. In addition, our framework enables data exchange between heterogeneous models within a multi-model environment and supports real-time features, which are essential for representing modern software systems but were not supported in earlier approaches. The coordination framework can be explained using the coalgebraic formalism, but most importantly, we adopt the coalgebraic paradigm to align heterogeneous behavioral models. In addition, we conducted an in-depth analysis of BPMN model verification to demonstrate that our framework can handle complex behavioral languages and explore potential future directions, such as understanding and resolving violated properties.

This thesis marks an initial step towards a deeper understanding and effective use of behavioral multi-models in MDSE. The primary objective of this thesis is to help address the growing complexity of software systems, which have become essential components of modern daily life. Although analysis of individual behavioral models, such as our analysis of BPMN [102], has shown promising results, these methods have not yet been adapted for heterogeneous environments, which is the focus of this thesis.

Open Science The datasets and tool implementations presented in this thesis and the accompanying paper are publicly available [185]. We devoted considerable

effort to ensuring that the implementations are accessible and portable and that the benchmarks are reproducible. Our work has greatly benefited from publicly available implementations and results. We encourage fellow researchers to adopt open science practices by making their results accessible, reproducible, and available.

BIBLIOGRAPHY

[1] J.-R. Abrial. Formal methods in industry: Achievements, problems, future. In *Proceedings of the 28th International Conference on Software Engineering*, pages 761–768, Shanghai China, May 2006. ACM. 2.3.1

[2] R. Acerbis, A. Bongio, M. Brambilla, M. Tisi, S. Ceri, and E. Tosetti. Developing eBusiness Solutions with a Model Driven Approach: The Case of Acer EMEA. In L. Baresi, P. Fraternali, and G.-J. Houben, editors, *Web Engineering*, volume 4607, pages 539–544. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. 1.1, 1.1

[3] L. Aceto, W. Fokkink, and C. Verhoef. Chapter 3 - Structural operational semantics. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 197–292. Elsevier Science, Amsterdam, 2001. 2.1

[4] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan. 1997. 2.1, 4.3.1

[5] Alfresco Software lnc. Open Source Business Automation | Activiti. https://www.activiti.org/. 1.1

[6] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, Jan. 1997. 3.4

[7] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997. 3.1.3, 3.3.2, 3.4

[8] M. Amrani, R. Mittal, M. Goulão, V. Amaral, S. Guérin, S. Martínez, D. Blouin, A. Bhobe, and Y. Hallak. A Survey of Federative Approaches for Model Management in MBSE. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, pages 990–999, Linz Austria, Sept. 2024. ACM. 1.2

[9] C. André. Syntax and semantics of the clock constraint specification language (CCSL). Technical Report RR-6925, INRIA, 2009. 3.3.1, 4.1.3

[10] Apache Software Foundation. Apache Avro. https://avro.apache.org/. 1.1

[11] Apache Software Foundation. Apache Thrift. https://thrift.apache.org/. 1.1

[12] F. Arbab. The IWIM model for coordination of concurrent activities. In G. Goos, J. Hartmanis, J. Leeuwen, P. Ciancarini, and C. Hankin, editors, *Coordination Languages and Models*, volume 1061, pages 34–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. 3.1.2

[13] F. Arbab. What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science*, pages 11–22, 1998. 3.3.3

[14] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, June 2004. 3.1.2, 3.4

[15] F. Arbab, P. Ciancarini, and C. Hankin. Coordination languages for parallel programming. *Parallel Computing*, 24(7):989–1004, July 1998. 3.2

[16] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, Feb. 1993. 3.1.2, 3.4

[17] J.-C. Bach, A. Beugnard, J. Champeau, F. Dagnat, S. Guérin, and S. Martínez. 10 years of Model Federation with Openflexo: Challenges and Lessons Learned. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, pages 25–36, Linz Austria, Sept. 2024. ACM. 1.2

[18] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008. 2.1, 2.1, 2.3.3, 2.3.3, 2.4

[19] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, Apr. 2003. 3.4

[20] A. Barriga, R. Heldal, L. Iovino, M. Marthinsen, and A. Rutle. An extensible framework for customizable model repair. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 24–34, Virtual Event Canada, Oct. 2020. ACM. 5.5

[21] A. Barriga, A. Rutle, and R. Heldal. AI-powered model repair: An experience report—lessons learned, challenges, and opportunities. *Software and Systems Modeling*, 21(3):1135–1157, June 2022. 1.2, 5.5

[22] E. Bartocci, R. Grosu, P. Katsaros, C. R. Ramakrishnan, and S. A. Smolka. Model Repair for Probabilistic Systems. In P. A. Abdulla and K. R. M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605, pages 326–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 5.5

[23] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, May 2011. 3.4

[24] P. Blackburn, M. D. Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 1 edition, June 2001. 2.3.3

[25] S. Bliudze and J. Sifakis. The Algebra of Connectors—Structuring Interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, Oct. 2008. 3.4

[26] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels. Model checking flight control systems: The Airbus experience. In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 18–27, Vancouver, BC, Canada, 2009. IEEE. 2.3.1

[27] A. Boronat, A. Knapp, J. Meseguer, and M. Wirsing. What Is a Multi-modeling Language? In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques*, volume 5486, pages 71–87. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. 1.2

[28] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189, 1993. 2.3.1

[29] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Number 4 in Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, San Rafael, Calif., 2 edition, 2017. 1.1, 1, 1.1, 1.1, 1.1

[30] C.-H. Cai, J. Sun, and G. Dobbie. Automatic B-model repair using model checking and machine learning. *Automated Software Engineering*, 26(3):653–704, Sept. 2019. 5.5

[31] Camunda. The Universal Process Orchestrator. `https://camunda.com/`. 1.1

[32] B. Camus, V. Galtier, and M. Caujolle. Hybrid co-simulation of FMUs using DEV&DESS in MECSYCO. In *2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS)*, pages 1–8, 2016. 3.1.1, 3.3.2, 3.4

[33] B. Camus, T. Paris, J. Vaubourg, Y. Presse, C. Bourjot, L. Ciarletta, and V. Chevrier. Co-simulation of cyber-physical systems using a DEVS wrapping strategy in the MECSYCO middleware. *SIMULATION*, 94(12):1099–1127, Dec. 2018. 3.1.1, 3.3.2, 3.4

[34] N. Carriero and D. Gelernter. Linda in context. *Communications of The Acm*, 32(4):444–458, Apr. 1989. 3.1.2, 3.3.3, 3.4

[35] N. J. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20(4):633–655, Apr. 1994. 3.4

[36] J. Cheney, J. Gibbons, J. McKinna, and P. Stevens. On principles of Least Change and Least Surprise for bidirectional transformations. *The Journal of Object Technology*, 16(1):3:1, 2017. 5.5

[37] G. Ciatto, S. Mariani, M. Louvel, A. Omicini, and F. Zambonelli. Twenty Years of Coordination Technologies: State-of-the-Art and Perspectives. In G. Di Marzo Serugendo and M. Loreti, editors, *Coordination Models and Languages*, volume 10852, pages 51–80. Springer International Publishing, Cham, 2018. 3.1.2, 3.2, 3.1.2, 3.2

[38] A. Cicchetti, F. Ciccozzi, and A. Pierantonio. Multi-view approaches for software and system modelling: A systematic literature review. *Software and Systems Modeling*, 18(6):3207–3233, Dec. 2019. 5.5

[39] E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model Checking*. The Cyber-Physical Systems Series. The MIT Press, Cambridge, Massachusetts London, England, second edition edition, 2018. 2.1, 2.3.3, 2.3.3, 2.3.3

[40] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer International Publishing, Cham, 2018. 1, 2.3.1, 2.3.2, 2.3, 2.3.2, 2.3.3, 2.3.3, 2.3.3, 2.3.3, 2.3.4

[41] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, Aug. 2002. 2.4, 2.4, 2.4

[42] P. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 16–25, Schloss Velen, Germany, 1996. IEEE Comput. Soc. Press. 3.1.3, 3.2

[43] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. *Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach*, pages 163–245. World Scientific, Feb. 1997. 2.5

[44] C. Dad, J.-P. Tavella, and S. Vialle. Synthesis and feedback on the distribution and parallelization of FMI-CS-based co-simulations with the DACCOSIM platform. *Parallel Computing*, 106:102802, Sept. 2021. 3.3.2, 3.4, 4, 4.3.2

[45] J. Dahmann. High level architecture for simulation. In *Proceedings First International Workshop on Distributed Interactive Simulation and Real Time Applications*, pages 9–14, 1997. 3.1.1

[46] J. DeAntoni and F. Mallet. TimeSquare: Treat Your Models with Logical Time. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, C. A. Furia, and S. Nanz, editors, *Objects, Models, Components, Patterns*, volume 7304, pages 34–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 3.3.1

[47] E. W. Dijkstra. Chapter I: Notes on structured programming. In *Structured Programming*, pages 1–82. Academic Press Ltd., GBR, 1972. 2.3.1

[48] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach.*, pages 247–312. World Scientific, Feb. 1997. 2.5

[49] J. Eker, J. Janneck, E. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan. 2003. 1.2, 3.1.4, 3.3.2, 3.4, 4

[50] G. Engels, J. M. Küster, R. Heckel, and L. Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. *ACM SIGSOFT Software Engineering Notes*, 26(5):186–195, Sept. 2001. 4.2

[51] J. Évora Gómez, J. J. Hernández Cabrera, J.-P. Tavella, S. Vialle, E. Kremers, and L. Frayssinet. Daccosim NG: Co-simulation made simpler and faster. In *The 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, pages 785–794, Feb. 2019. 4.3.2

[52] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data & Knowledge Engineering*, 70(5):448–466, May 2011. 1.7.2, 5.1

[53] D. Fahland and W. M. Van Der Aalst. Model repair — aligning process models to reality. *Information Systems*, 47:220–243, Jan. 2015. 5.5

[54] A. Falcone and A. Garro. Distributed Co-Simulation of Complex Engineered Systems by Combining the High Level Architecture and Functional Mock-up Interface. *Simulation Modelling Practice and Theory*, 97:101967, Dec. 2019. 3.1.1

[55] J. L. Fiadeiro and A. Lopes. Semantics of architectural connectors. In G. Goos, J. Hartmanis, J. Van Leeuwen, M. Bidoit, and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214, pages 503–519. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. 3.4

[56] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, Aug. 1991. 3.3.1

[57] M. Firore and D. Turi. Semantics of name and value passing. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 93–104, Boston, MA, USA, 2001. IEEE Comput. Soc. 2.2

[58] J. Freund and B. Rücker. *Real-Life BPMN: Using BPMN and DMN to Analyze, Improve, and Automate Processes in Your Company*. Camunda, Berlin, 4th edition, 2019. 2.3.4

[59] V. Galtier, S. Vialle, C. Dad, J.-P. Tavella, J.-P. Lam-Yee-Mui, and G. Plessis. FMI-Based distributed multi-simulation with DACCOSIM. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, DEVS '15, pages 39–46, San Diego, CA, USA, 2015. Society for Computer Simulation International. 3.3.2, 3.3.3, 3.4, 4, 4.3.2

[60] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, Feb. 1992. 3.2

[61] GEMOC Initiative. The GEMOC Initiative. https://gemoc.org/BCOoL/, 2013. 4.1.3

[62] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, 14(1):15–40, Feb. 2012. 2.5

[63] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe. Co-Simulation: A Survey. *ACM Computing Surveys*, 51(3):1–33, May 2019. 3.1.1, 3.2, 3.3.1, 3.3.3, 4.3, 5.4

[64] C. Gomes, C. Thule, J. Deantoni, P. G. Larsen, and H. Vangheluwe. Co-simulation: The Past, Future, and Open Challenges. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems*, volume 11246, pages 504–520. Springer International Publishing, Cham, 2018. 3.2

[65] Google. Protocol Buffers. `https://protobuf.dev/`. 1.1

[66] G. Goos, J. Hartmanis, and J. Van Leeuwen, editors. *Coordination Models and Languages*, volume 2039, pages 33–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. 3.2

[67] A. Haber, J. O. Ringert, and B. Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems, 2014. 3.1.3, 3.3.1, 3.3.2, 3.4, 4, 4.1.2

[68] I. Hafner and N. Popper. On the terminology and structuring of co-simulation methods. In *Proceedings of the 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 67–76, Weßling Germany, Dec. 2017. ACM. 3.2

[69] I. Hafner and N. Popper. An Overview of the State of the Art in Co-Simulation and Related Methods. *SNE Simulation Notes Europe*, 31(4):185–200, Dec. 2021. 3.2

[70] C. Hardebolle and F. Boulanger. Exploring multi-paradigm modeling techniques. *SIMULATION*, 85(11-12):688–708, 2009. 1.3

[71] D. Harel and E. Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 246–257, USA, 1996. IEEE Computer Society. 1.1

[72] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin/Heidelberg, 2006. 2.5

[73] I. Hasuo, B. Jacobs, and A. Sokolova. The Microcosm Principle and Concurrency in Coalgebra. In R. Amadio, editor, *Foundations of Software Science and Computational Structures*, volume 4962, pages 246–260. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 2.2

[74] R. Heckel and G. Taentzer, editors. *Graph Transformation, Specifications, and Nets: In Memory of Hartmut Ehrig*, volume 10800 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2018. 2.5

[75] R. Heckel and G. Taentzer. *Graph Transformation for Software Engineers: With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer International Publishing, Cham, 2020. 2.5

[76] A. R. Hevner. A Three Cycle View of Design Science Research. *Scandinavian Journal of Information Systems*, 19(2):4, 2007. 1.7, 1.7.1, 1.6, 1.7.1

[77] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004. 1.7.1

[78] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. 4.3.1

[79] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978. 1.1, 4.2

[80] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. The Addison-Wesley Signature Series. Addison-Wesley, Boston, 2004. 5.5

[81] S. Hussain. Investigating Architecture Description Languages (ADLs) A Systematic Literature Review, 2013. 3.2

[82] itemis AG. Itemis CREATE | State Machine Tool - Lowcode Development. https://www.itemis.com/en/products/itemis-create/, Dec. 2024. 1.1

[83] B. Jacobs. The temporal logic of coalgebras via Galois algebras. *Mathematical Structures in Computer Science*, 12(6):875–903, Dec. 2002. 2.3.3

[84] B. Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge University Press, Cambridge, 2017. 2.2, 2.2, 2.2, 2.2, 2.2

[85] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use*. Springer, Berlin, 2nd ed edition, 1996. 1.1, 1.3, 2.3.4

[86] K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. 1.1, 2.3.4, 4.2

[87] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study:. Technical report, Defense Technical Information Center, Fort Belvoir, VA, Nov. 1990. 3, 3.2, 3.3

[88] H. Kastenberg and A. Rensink. Model Checking Dynamic States in GROOVE. In A. Valmari, editor, *Model Checking Software*, volume 3925, pages 299–305. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. 2.5

[89] M. Kick. Bialgebraic Modelling of Timed Processes. In G. Goos, J. Hartmanis, J. Van Leeuwen, P. Widmayer, S. Eidenbenz, F. Triguero, R. Morales, R. Conejo, and M. Hennessy, editors, *Automata, Languages and Programming*, volume 2380, pages 525–536. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. 2.2

[90] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering, 2007. 3.2

[91] H. Klare and J. Gleitze. Commonalities for Preserving Consistency of Multiple Models. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 371–378, Munich, Germany, Sept. 2019. IEEE. 1.2, 5.4

[92] B. Klin. Bialgebras for structural operational semantics: An introduction. *Theoretical Computer Science*, 412(38):5043–5069, Sept. 2011. 2.2, 2.2

[93] A. Knapp and T. Mossakowski. Multi-view Consistency in UML: A Survey. In R. Heckel and G. Taentzer, editors, *Graph Transformation, Specifications, and Nets*, volume 10800, pages 37–60. Springer International Publishing, Cham, 2018. 1.2

[94] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6, Vancouver, BC, Canada, May 2009. IEEE. 1.2

[95] H. König, U. Wolter, and T. Kräuter. Structural operational semantics for heterogeneously typed coalgebras. In P. Baldan and V. de Paiva, editors, *10th Conference on Algebra and Coalgebra in Computer Science (CALCO 2023)*, volume 270 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:17, Dagstuhl, Germany, Sept. 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 1.5, 1.7.2, 5.2

[96] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In W. van der Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, R. F. Paige, and B. Meyer, editors, *Objects, Components, Models and Patterns*, volume 11, pages 297–315. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 4.1.2

[97] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: A framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, Sept. 2010. 4.1.2

[98] T. Kräuter. Towards behavioral consistency in heterogeneous modeling scenarios. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 666–671, Fukuoka, Japan, Oct. 2021. IEEE. 5.2

[99] T. Kräuter, H. König, A. Rutle, Y. Lamo, and P. Stünkel. Behavioral consistency in multi-modeling. *The Journal of Object Technology*, 22(2):2:1, 2023. 1.5, 1.7.2, 2.1, 2.1

[100] T. Kräuter, A. Rutle, H. König, and Y. Lamo. A higher-order transformation approach to the formalization and analysis of BPMN using graph transformation systems. *Logical Methods in Computer Science*, Volume 20, Issue 4:12533, Oct. 2024. 1.5, 1.7.2, 5.2

[101] T. Kräuter, A. Rutle, Y. Lamo, H. König, and F. Durán. Towards the Coordination and Verification of Heterogeneous Systems with Data and Time, Submitted to the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2025), Apr. 2025. 1.5, 1.7.2, 5.2

[102] T. Kräuter, P. Stünkel, A. Rutle, Y. Lamo, and H. König. BPMN Analyzer 2.0: Instantaneous, Comprehensible, and Fixable Control Flow Analysis for Realistic BPMN Models. In *Best BPM Dissertation Award, Doctoral Consortium, and Demonstrations & Resources Forum Co-Located with 22nd International Conference on Business Process Management (BPM 2024), Krakow, Poland, September 1st to 6th, 2024.*, Sept. 2024. 1.5, 1.7.2, 5.2, 5.6

[103] C. Kupke and D. Pattinson. Coalgebraic semantics of modal logics: An overview. *Theoretical Computer Science*, 412(38):5070–5094, Sept. 2011. 1.5, 2.2, 2.3.3

[104] J. Küster and J. Stehr. Towards explicit behavioral consistency concepts in the UML. In *Proceedings of 2nd ICSE Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (Portland, USA)*, 2003. 4.2

[105] E. A. Lee. Disciplined heterogeneous modeling. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part II*, MODELS'10, pages 273–287, Berlin, Heidelberg, 2010. Springer-Verlag. 4.3, 4.3.1

[106] M. Levandowsky and D. Winter. Distance between Sets. *Nature*, 234(5323):34–35, Nov. 1971. 3.5.3

[107] G. Liboni. *Complex Systems Co-Simulation with the CoSim20 Framework : For Efficient and Accurate Distributed Co-Simulations*. Theses, Université Côte d'Azur, Apr. 2021. 3.1.1, 3.3.3, 3.4

[108] G. Liboni and J. Deantoni. CoSim20: An Integrated Development Environment for Accurate and Efficient Distributed Co-Simulations. In *Proceedings of the 2020 International Conference on Big Data in Management*, pages 76–83, Manchester United Kingdom, May 2020. ACM. 3.4

[109] J. Lin. The Lambda and the Kappa. *IEEE Internet Computing*, 21(5):60–66, 2017. 1

[110] Y. Lingling and Z. Wei. An Overview of Software Architecture Description Language and Evaluation Method. In J. Kacprzyk and G. Yang, editors, *Proceedings of the 2012 International Conference on Communication, Electronics and Automation Engineering*, volume 181, pages 895–901. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. 3.2

[111] M. Lohstroh. *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*. PhD thesis, EECS Department, University of California, Berkeley, Dec. 2020. 3.1.2, 3.3.1, 3.3.3, 3.4, 4, 4.1.1, 4.1.1

[112] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Transactions on Embedded Computing Systems*, 20(4):1–27, July 2021. 3.1.2, 3.3.2, 3.4, 4, 4.1.1

[113] M. Lohstroh, M. Schoeberl, A. Goens, A. Wasicek, C. Gill, M. Sirjani, and E. A. Lee. Actors Revisited for Time-Critical Systems. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–4, Las Vegas NV USA, June 2019. ACM. 4.1.1

[114] N. Macedo, T. Jorge, and A. Cunha. A Feature-Based Classification of Model Repair Approaches. *IEEE Transactions on Software Engineering*, 43(7):615–640, July 2017. 1.2, 5.5

[115] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In G. Goos, J. Hartmanis, J. Van Leeuwen, W. Schäfer, and P. Botella, editors, *Software Engineering — ESEC '95*, volume 989, pages 137–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. 3.1.3

[116] R. Majumdar. Challenges and Opportunities in Model Checking Large-scale Distributed Systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–2, Lisbon Portugal, May 2024. ACM. 2.3.1

[117] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, June 2013. 3.2

[118] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Transactions on Software Engineering and Methodology*, 18(4):1–56, July 2009. 3.1.2

[119] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007. 2.4, 2.4, 2.4, 2.4, 2.4, 3.1.4, 3.3.1

[120] S. T. March and G. F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, 1995. 1.7.1, 1.7.2, 1.7.2

[121] L. Z. Markosian, M. Mansouri-Samani, P. C. Mehlitz, and T. Pressburger. Program Model Checking Using Design-for-Verification: NASA Flight Software Case Study. In *2007 IEEE Aerospace Conference*, pages 1–9, Big Sky, MT, USA, 2007. IEEE. 2.3.1

[122] N. Martí-Oliet and J. Meseguer. Rewriting Logic as a Logical and Semantic Framework. *Electronic Notes in Theoretical Computer Science*, 4:190–225, 1996. 2.4

[123] Martin Fowler. Uml Mode. `https://martinfowler.com/bliki/UmlMode.html`, 2003. 1.1

[124] D. A. Mathaikutty, H. D. Patel, S. K. Shukla, and A. Jantsch. UMoC++: A C++-Based Multi-MoC Modeling Environment. In A. Vachoux, editor, *Applications of Specification and Design Languages for SoCs*, pages 115–130. Springer Netherlands, Dordrecht, 2006. 3.4

[125] M. Maximova, H. Giese, and C. Krause. Probabilistic timed graph transformation systems. *Journal of Logical and Algebraic Methods in Programming*, 101:110–131, Dec. 2018. 2.5

[126] N. Medvidovic. Moving Architectural Description from Under the Technology Lamppost. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, pages 2–3, Cavtat, Dubrovnik, Croatia, 2006. IEEE. 3.1.3

[127] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000. 3.1.3, 3.2, 3.3.1

[128] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In G. Goos, J. Hartmanis, J. van Leeuwen, M. Jazayeri, and H. Schauer, editors, *Software Engineering — ESEC/FSE'97*, volume 1301, pages 60–76. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. 3.1.3, 3.2

[129] Mendix Technology B.V. Mendix. https://www.mendix.com/, Dec. 2024. 1.1

[130] J. Meseguer and G. Roşu. The rewriting logic semantics project: A progress report. *Information and Computation*, 231:38–69, Oct. 2013. 2.4

[131] T. Mettler, M. Eurich, and R. Winter. On the Use of Experiments in Design Science Research: A Proposition of an Evaluation Framework. *Communications of the Association for Information Systems*, 34(10), 2014. 1.7.1

[132] Microsoft Corporation. Bond, Microsoft's cross-platform framework for working with schematized data -. https://microsoft.github.io/bond/. 1.1

[133] Microsoft Corporation. AI-Powered Low-Code Tools | Microsoft Power Platform. https://www.microsoft.com/en-us/power-platform, Feb. 2025. 1.1, 1.3

[134] Microsoft Corporation. Microsoft Power Automate – Process Automation Platform. https://www.microsoft.com/en-us/power-platform/products/power-automate, Feb. 2025. 1.1, 1.3

[135] R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer, Berlin, 1980. 1.1

[136] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge Univ. Press, Cambridge, 7. print edition, 2010. 1.1

[137] P. Mishra and N. Dutt. Architecture Description Languages. In *Customizable Embedded Processors*, pages 59–76. Elsevier, 2007. 3.2

[138] Modelisar. Functional Mock-up Interface Specification. https://fmi-standard.org/docs/3.0.1/, July 2023. 3.1.1, 4.3.2

[139] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernandez, B. Nordmoen, and M. Fritzsche. Where does model-driven engineering help? Experiences from three industrial cases. *Software & Systems Modeling*, 12(3):619–639, July 2013. 1.1

[140] M. Nagl. A tutorial and bibliographical survey on graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73, pages 70–126. Springer-Verlag, Berlin/Heidelberg, 1979. 2.5

[141] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 2015. 2.3.1

[142] L. J. B. Nixon, E. Simperl, R. Krummenacher, and F. Martin-Recuerda. Tuplespace-based computing for the Semantic Web: A survey of the state-of-the-art. *The Knowledge Engineering Review*, 23(2):181–212, June 2008. 3.1.2

[143] Object Management Group. Business Process Model and Notation (BPMN), Version 2.0.2. https://www.omg.org/spec/BPMN/, Dec. 2013. 1.1, 2.3.4

[144] Object Management Group. Unified Modeling Language, Version 2.5.1. https://www.omg.org/spec/UML, Dec. 2017. 1.1, 2.4

[145] C. Oliveira and M. Wermelinger. The CommUnity Workbench. *Science of Computer Programming*, 69(1-3):46–55, Dec. 2007. 3.4

[146] P. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20:161–196, Apr. 2007. 2.4

[147] P. C. Ölveczky. Real-Time Maude and Its Applications. In S. Escobar, editor, *Rewriting Logic and Its Applications*, volume 8663, pages 42–79. Springer International Publishing, Cham, 2014. 2.4, 5.1, 5.2

[148] P. C. Ölveczky. *Designing Reliable Distributed Systems*. Undergraduate Topics in Computer Science. Springer London, London, 2017. 2.4, 2.4, 2.4

[149] A. Omicini and M. Viroli. Coordination models and languages: From parallel computing to self-organisation. *The Knowledge Engineering Review*, 26(1):53–59, Feb. 2011. 3.1.2

[150] OutSystems- Software em Rede, S.A. OutSystems. https://www.outsystems.com/, Dec. 2024. 1.1

[151] M. Ozkaya and C. Kloukinas. Are We There Yet? Analyzing Architecture Description Languages for Formal Analysis, Usability, and Realizability. In *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, pages 177–184, Santander, Spain, Sept. 2013. IEEE. 3.1.3, 3.2

[152] M. J. Page, J. E. McKenzie, P. M. Bossuyt, I. Boutron, T. C. Hoffmann, C. D. Mulrow, L. Shamseer, J. M. Tetzlaff, E. A. Akl, S. E. Brennan, R. Chou, J. Glanville, J. M. Grimshaw, A. Hróbjartsson, M. M. Lalu, T. Li, E. W. Loder, E. Mayo-Wilson, S. McDonald, L. A. McGuinness, L. A. Stewart, J. Thomas, A. C. Tricco, V. A. Welch, P. Whiting, and D. Moher. The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. *Systematic Reviews*, 10(1):89, Dec. 2021. 3.2, 3.2, 3.4, 5.3

[153] R. K. Pandey. Architectural description languages (ADLs) vs UML: A review. *ACM SIGSOFT Software Engineering Notes*, 35(3):1–5, 2010. 3.1.3, 3.2

[154] G. A. Papadopoulos. Models and Technologies for the Coordination of Internet Agents: A Survey. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents*, pages 25–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. 3.2

[155] G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *Advances in Computers*, volume 46, pages 329–400. Elsevier, 1998. 3.2, 3.3.1

[156] G. A. Papadopoulos and F. Arbab. Modelling activities in information systems using the coordination language MANIFOLD. In *Proceedings of the 1998 ACM Symposium on Applied Computing - SAC '98*, pages 185–193, Atlanta, Georgia, United States, 1998. ACM Press. 3.1.2, 3.4

[157] K. Petersen, S. Vakkalanka, and L. Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, Aug. 2015. 3.2, 5.3

[158] C. Ptolemaeus, editor. *System Design, Modeling, and Simulation: Using Ptolemy II*. UC Berkeley EECS Dept, Berkeley, Calif, 1. ed., version 1.02 edition, 2014. 3.1.4, 3.3.1, 3.4, 4, 4.3.1

[159] Ptolemy Project. Ptolemy II Home Page. 4.3.1

[160] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062, pages 479–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. 2.5, 3.1.4

[161] A. Rensink. Nested Quantification in Graph Transformation Rules. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations*, Lecture Notes in Computer Science, pages 1–13, Berlin, Heidelberg, 2006. Springer. 2.5

[162] A. Rensink. Explicit state model checking for graph grammars. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, pages 114–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 2.5, 2.5

[163] A. Rensink. How Much Are Your Geraniums? Taking Graph Conditions Beyond First Order. In J.-P. Katoen, R. Langerak, and A. Rensink, editors, *ModelEd, TestEd, TrustEd*, volume 10500, pages 191–213. Springer International Publishing, Cham, 2017. 2.5

[164] A. Rensink and J.-H. Kuperus. Repotting the Geraniums: On Nested Graph Transformation Rules. *Electronic Communications of the EASST*, page Volume 18: Graph Transformation and Visual Modeling Techniques 2009, Sept. 2009. 2.5

[165] D. Rossi, G. Cabri, and E. Denti. Tuple-based Technologies for Coordination. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents*, pages 83–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. 3.1.2

[166] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1: Foundations*. WORLD SCIENTIFIC, Feb. 1997. 2.5

[167] B. Rücker. *Practical Process Automation: Orchestration and Integration in Microservices and Cloud Native Architectures*. O'Reilly, Beijing Boston Farnham Sebastopol Tokyo, first edition edition, 2021. 2.3.4, 2.5, 2.7

[168] J. Rutten. Universal coalgebra: A theory of systems. *Theoretical Computer Science*, 249(1):3–80, Oct. 2000. 2.2

[169] RWTH Aachen. MontiCore - Language Workbench | SE@RWTH. https://www.se-rwth.de/research/MontiCore/, Jan. 2025. 4.1.2

[170] I. Sander and A. Jantsch. System Modeling and Transformational Design Refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, Jan. 2004. 3.4

[171] I. Sander, A. Jantsch, and S.-H. Attarzadeh-Niaki. ForSyDe: System Design Using a Functional Language and Models of Computation. In S. Ha and J. Teich, editors, *Handbook of Hardware/Software Codesign*, pages 1–42. Springer Netherlands, Dordrecht, 2016. 3.4

[172] H. Schichl. Models and the History of Modeling. In P. M. Pardalos, D. W. Hearn, and J. Kallrath, editors, *Modeling Languages in Mathematical Optimization*, volume 88, pages 25–36. Springer US, Boston, MA, 2004. 1.1

[173] S. Schneider, M. Maximova, L. Sakizloglou, and H. Giese. Formal testing of timed graph transformation systems using metric temporal graph logic. *International Journal on Software Tools for Technology Transfer*, 23(3):411–488, June 2021. 2.5

[174] G. Schweiger, G. Engel, J.-P. Schöggl, I. Hafner, T. S. Nouidui, and C. Gomes. Co-Simulation - An Empirical Survey: Applications, Recent Developments and Future Challenges. *SNE Simulation Notes Europe*, 30(2):73–76, June 2020. 3.2, 5.4

[175] G. Schweiger, C. Gomes, G. Engel, I. Hafner, J. Schoeggl, A. Posch, and T. Nouidui. An empirical survey on co-simulation: Promising standards, challenges and research needs. *Simulation Modelling Practice and Theory*, 95:148–163, Sept. 2019. 3.2

[176] M. Seidl, M. Scholz, C. Huemer, and G. Kappel. *UML @ Classroom*. Undergraduate Topics in Computer Science. Springer International Publishing, Cham, 2015. 1.1, 1.2

[177] P. Stünkel. *A Framework for Multi-Model Consistency Management*. PhD thesis, Western Norway University of Applied Sciences, 2022. 1.2, 1.3, 1.4, 5.4

[178] P. Stünkel, H. König, Y. Lamo, and A. Rutle. Comprehensive Systems: A formal foundation for Multi-Model Consistency Management. *Formal Aspects of Computing*, July 2021. 1.2, 1.3, 1.4, 5.4

[179] P. Stünkel, H. König, A. Rutle, and Y. Lamo. Multi-Model Evolution through Model Repair. *The Journal of Object Technology*, 20(1):1:1, 2021. 1.2

[180] The Lingua Franca project. Intuitive concurrent programming in any language | Lingua Franca. https://www.lf-lang.org/, Jan. 2025. 4.1.1

[181] Tim Kräuter. BPMN Analyzer. https://bpmn-analyzer.wittyrock-9d6a3c00.northeurope.azurecontainerapps.io/, June 2023. 1.7.2

[182] Tim Kräuter. BPMN Analyzer Tool Demonstration. https://www.youtube.com/watch?v=MxXbNUl6IjE, June 2023. 1.7.2

[183] Tim Kräuter. BPMN Analyzer 2.0. https://timkraeuter.com/bpmn-analyzer-js/, May 2024. 1.7.2

[184] Tim Kräuter. BPMN Analyzer 2.0 Demonstration. https://youtu.be/Nv2W-hXNZYA, May 2024. 1.7.2

[185] Tim Kräuter. Thesis: Artifacts. https://github.com/timKraeuter/thesis-artifacts, Nov. 2024. 2.3.4, 2.4, 2.5, 3.2, 3.2, 3.4, 3.4, 3.5.3, 3.6, 4.1.1, 4.1.2, 5.3, 5.6

[186] R. Tolksdorf and R. Menezes. Using Swarm Intelligence in Linda Systems. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, A. Omicini, P. Petta, and J. Pitt, editors, *Engineering Societies in the Agents World IV*, volume 3071, pages 49–65. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. 3.1.2

[187] W. Torres, M. G. J. van den Brand, and A. Serebrenik. A systematic literature review of cross-domain model consistency checking by model management tools. *Software and Systems Modeling*, Oct. 2020. 1.2

[188] M. Vara Larsen. *BCOol : The Behavioral Coordination Operator Language*. PhD thesis, Université Nice Sophia Antipolis, Apr. 2016. 3.1.4, 3.2, 3.3.1, 3.4, 3.4, 4, 4.1.3, 4.1.3, 5.1

[189] M. E. Vara Larsen, J. Deantoni, B. Combemale, and F. Mallet. A Behavioral Coordination Operator Language (BCOoL). In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 186–195, Ottawa, ON, Canada, Sept. 2015. IEEE. 3.1.4, 3.3.1, 3.3.2, 3.3.3, 3.4, 3.4, 4, 4.1.3

[190] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini. From Field-Based Coordination to Aggregate Computing. In G. Di Marzo Serugendo

and M. Loreti, editors, *Coordination Models and Languages*, volume 10852, pages 252–279. Springer International Publishing, Cham, 2018. 3.2

[191] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini. From distributed coordination to field calculus and aggregate computing. *Journal of Logical and Algebraic Methods in Programming*, 109:100486, Dec. 2019. 3.1.2

[192] J. Whittle, J. Hutchinson, and M. Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, May 2014. 1.1, 1.1

[193] R. J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. 1.7.1

[194] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall PTR, USA, 1978. 1.1

[195] E. Woods and R. Hilliard. Architecture Description Languages in Practice Session Report. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 243–246, Pittsburgh, PA, USA, 2005. IEEE. 3.1.3

[196] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*, Boston, MA, Apr. 2009. USENIX Association. 2.3.1

[197] S. Yao and S. Shatz. Consistency Checking of UML Dynamic Models Based on Petri Net Techniques. In *2006 15th International Conference on Computing*, pages 289–297, Mexico city, Mexico, Nov. 2006. IEEE. 4.2

Part II

ARTICLES

PAPER A

STRUCTURAL OPERATIONAL SEMANTICS FOR HETEROGENEOUSLY TYPED COALGEBRAS

Harald König, Uwe Wolter, Tim Kräuter

In Proceedings of the Conference on Algebra and Coalgebra in Computer Science (CALCO), 2023, https://doi.org/10.4230/LIPIcs.CALCO.2023.7

Structural Operational Semantics for Heterogeneously Typed Coalgebras

Harald König
Fachhochschule für die Wirtschaft Hannover, Germany
Western Norway University of Applied Sciences, Bergen, Norway

Uwe Wolter
University of Bergen, Norway

Tim Kräuter
Western Norway University of Applied Sciences, Bergen, Norway

Abstract

Concurrently interacting components of a modular software architecture are heterogeneously structured behavioural models. We consider them as coalgebras based on different endofunctors. We formalize the composition of these coalgebras as specially tailored segments of distributive laws of the bialgebraic approach of Turi and Plotkin. The resulting categorical rules for structural operational semantics involve many-sorted algebraic specifications, which leads to a description of the components together with the composed system as a single holistic behavioural system. We evaluate our approach by showing that observational equivalence is a congruence with respect to the algebraic composition operation.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning

Keywords and phrases Coalgebra, Bialgebra, Structural operational semantics, Compositionality

Digital Object Identifier 10.4230/LIPIcs.CALCO.2023.7

Funding *Harald König*: The author thanks the University of Bergen for support of this project.

Acknowledgements The authors thank the anonymous referees for their helpful suggestions that have helped to improve this article.

1 Introduction

In a modular and component-based software architecture of a compound system the individual components interact concurrently. Categorically, these individual state-based components are modelled as *coalgebras*. However, in a landscape of multiple interacting systems these behavioural models are *heterogeneously* typed: There are deterministic or non-deterministic labelled transition systems as well as probabilistic systems, systems with or without termination, with or without output and so on, see [24], Chapter 3. Hence the coalgebras of the individual components are based on different endofunctors.

Reasoning about the correct behaviour of a compound system often requires establishing correctness of each local component and furthermore using theoretical means, which guarantee that global behaviour is determined by local behaviours. In [9], this modular method is called *compositionality* and a precise formulation of it requires the use of a framework, which captures the operational semantics of concurrent processes. Such a framework is given by transition rules of structural operational semantics (SOS). Conditional rules of the form

$$\frac{x \xrightarrow{a} x' \qquad y \xrightarrow{b} y'}{\mathsf{op}(x,y) \xrightarrow{c} \mathsf{op}(x',y')} \tag{1}$$

10th Conference on Algebra and Coalgebra in Computer Science (CALCO 2023).
Editors: Paolo Baldan and Valeria de Paiva; Article No. 7; pp. 7:1–7:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

generate systems, whose states are closed terms over an *algebraic* signature [1]. Well-known rule formats are GSOS[1] [3] and tyft/tyxt [8]. For some of these rule formats one can prove compositionality to hold, whereas counterexamples can be provided for other formats [8].

In this paper, we propose a formal structure, which describes the composition of *heterogeneously typed* coalgebras with the help of structural operational semantics. For this, it is important to provide a suitable rule format, which guarantees compositionality (and hopefully other similar requirements) in heterogeneous environments. Since we deal with supposedly arbitrarily varying behavioural specifications, we need more general rule formats, which cannot be expected to be homogeneous like GSOS or tyft/tyxt. An adequate generalization of transition rules in the context of coalgebraic specifications are *natural transformations* between functors, whose domain and codomain reflect the transition from $n(\geq 2)$ local systems to one compound system, i.e., functors of type $\mathcal{SET}^n \to \mathcal{SET}$. We will show that these so-called *interaction laws* (see Def. 12) can be embedded into distributive laws

$$\lambda : \vec{\Sigma}\vec{\mathcal{B}} \Rightarrow \vec{\mathcal{B}}\vec{\Sigma}$$

for suitable endofunctors $\vec{\Sigma}$ and $\vec{\mathcal{B}}$. Distributive laws are part of a *bialgebraic* approach, which has been described in [14], but was originally proposed by Turi and Plotkin [27]. Here $\vec{B}$ (and also $\vec{\Sigma}$) is a $\mathcal{SET}^{n+1}$-endofunctor, which simultaneously covers the behaviours of the n heterogeneously typed coalgebras *and* a specification of the compound system, which has to comprise the commonalities of the local system behaviours. The algebraic syntax functor $\vec{\Sigma}$ contains the operation(s), which realize(s) the transition from the local components (input of the operation) to the global view of the composed system (output). We evaluate our approach by proving compositionality to hold for interaction laws.

Whereas in process algebras like CCS or CSP[2] this transfer of observational indistinguishability during syntactical build-up of process terms has to be guaranteed [8], we rather want compositionality, when individual software components are composed into a global compound network. Whereas [14] circumscribes compositionality as observational equivalence (w.r.t. final semantics) being a congruence (i.e. the coinductive extension is an algebra homomorphism), we propose a slightly adapted definition tailored to the specific situation of heterogeneously typed interacting systems.

Our work was inspired by practical scenarios, where the coupling of behavioural models with other executable models like test runners or event injectors is of crucial importance [19]. Furthermore, recently, systematic approaches to co-simulation for large-scale system assessment have gained popularity [20]. Here, a typical scenario is the interaction with probabilistic systems [2, 26], which requires a concrete language for their interaction [5, 19]. While these DSLs[3] are already well-established, they lack theoretical underpinning in the form of transition rules to reason about correctness properties.

Hence, we answer the main question

How can we apply (parts of) the bialgebraic theory to understand the interaction of heterogeneously typed behavioural components?

by providing the following contributions and novelties:

- A proof for the preservation of observational equivalence, when n local components are based on *different* behavioural specifications $\mathcal{B}_1, \ldots, \mathcal{B}_n$, by embedding interaction laws into distributive laws.

[1] *General Structured Operational Semantics*
[2] Calculus of Communicating Systems [21], Communicating Sequential Processes [10]
[3] Domain Specific Languages

- An ensemble of n separated individual components together with the specification of its composition is formalised in one holistic *many-sorted* approach, i.e., as coalgebras for an endofunctor $\vec{\mathcal{B}} : \mathcal{SET}^{n+1} \to \mathcal{SET}^{n+1}$.

The paper is organized as follows: Sect. 2 clarifies notation, Sect. 3 presents the general setting based on practical scenarios as well as a motivating example, Sect. 4 recapitulates the survey [14] in some detail to make the content complete and comprehensible, and Sect. 5 presents the above mentioned novelties in detail: The linkage of the definitions of Interaction Law (in Def.12) and Congruence (adapted to the heterogeneous case in Def. 17) yield an adequate definition of compositionality and we can obtain our main statements: Theorem 21 proves compositionality to hold for interaction laws and Corollary 22 adapts the statement of the theorem to practical needs.

2 Basic Notation

We use the following notations: $\mathcal{SET}$ is the category of sets and total mappings. For two sets A and X we write X^A for the set of all total maps from A to X. A special set is 1, which denotes any singleton set, e.g. $(1 + X)^A$ is the set of all partial maps from A to X.

For functors we will use calligraphic letters like $\mathcal{F}$, $\mathcal{G}$, and, especially, letter $\mathcal{B}$ for behavioural and greek letter Σ for algebraic specifications. Categories are denoted $\mathbb{C}$ or $\mathbb{D}$, an application of a functor $\mathcal{F} : \mathbb{C} \to \mathbb{D}$ will be written $\mathcal{F}(X)$ for $X \in |\mathbb{C}|$ (or short $X \in \mathbb{C}$), the collection of objects of $\mathbb{C}$, whereas an application of $\mathcal{F}$ to a morphism $\alpha : X \to Y$ does not use parentheses: $\mathcal{F}\alpha : \mathcal{F}(X) \to \mathcal{F}(Y)$. Composition of functors $\mathcal{F}$ and $\mathcal{G}$ (if it is possible) is always written $\mathcal{GF}$ ($\mathcal{G}$ applied after $\mathcal{F}$). Special functors are $\mathcal{ID}_{\mathbb{C}}$, the identity functor on $\mathbb{C}$, and $\wp_{fin}$, the powerset functor assigning to a set the set of its finite subsets.

It is often convenient to give a definition (on objects) of a functor without explicitly naming its formal parameters, e.g. a functor $\mathcal{B}$ mapping a set X to the set $(1+X)^A$ (see above) is often denoted $\mathcal{B} = (1 + \_)^A$. Furthermore, when we give the complete definition of functors $\mathcal{F}$, we often combine object and morphism mapping by writing $X \xrightarrow{f} Y \;\mapsto\; \mathcal{F}(X) \xrightarrow{\mathcal{F}f} \mathcal{F}(Y)$.

As usual, a natural transformation ν between functors $\mathcal{F}$ and $\mathcal{G}$ with common domain and codomain, written $\nu : \mathcal{F} \Rightarrow \mathcal{G}$, is a family $(\nu_X : \mathcal{F}(X) \to \mathcal{G}(X))_{X \in |\mathbb{C}|}$ compatible with morphism mapping. For appropriate functors $\mathcal{H}$ and $\mathcal{H}'$ we denote with $\mathcal{H}\nu$ the family $(\mathcal{H}\nu_X : \mathcal{HF}(X) \to \mathcal{HG}(X))_{X \in |\mathbb{C}|}$ and with $\nu_{\mathcal{H}'}$ the family $(\nu_{\mathcal{H}'(X)} : \mathcal{FH}'(X) \to \mathcal{GH}'(X))_{X \in |\mathbb{C}|}$.

For an endofunctor $\mathcal{B} : \mathbb{C} \to \mathbb{C}$ a $\mathcal{B}$-coalgebra is a $\mathbb{C}$-morphism $X \xrightarrow{\alpha} \mathcal{B}(X)$, called the structure map and written (X, α) or - if X is clear from the context - just α. A coalgebra morphism from (X, α) to (Y, β) is a $\mathbb{C}$-morphism $f : X \to Y$ such that $\beta \circ f = \mathcal{B}f \circ \alpha$. Instead of f we sometimes write $(f, \mathcal{B}f)$ to stress the fact that commutativity involves $\mathcal{B}f$, as well. The resulting category of all coalgebras for $\mathcal{B} : \mathbb{C} \to \mathbb{C}$ will be denoted $\mathcal{B}$-$\mathcal{C}oalg$. If it admits a final object (Z, ζ) and if $(X, \alpha) \in \mathcal{B}$-$\mathcal{C}oalg$, we denote with $u_\alpha : X \to Z$ the *coinductive extension* of α, i.e. the unique $\mathcal{B}$-$\mathcal{C}oalg$-morphism into the final object.

Likewise for an endofunctor $\Sigma : \mathbb{C} \to \mathbb{C}$ a Σ-algebra is a $\mathbb{C}$-morphism $\Sigma(X) \xrightarrow{a} X$ written (a, X). An algebra morphism from (a, X) to (b, Y) is a $\mathbb{C}$-morphism $f : X \to Y$ such that $b \circ \Sigma f = f \circ a$. Instead of f we sometimes write $(\Sigma f, f)$. The resulting category of all algebras will be denoted Σ-$\mathcal{A}lg$.

For morphisms in combination with cartesian products, we use the following notations: If $f : A \to B$ and $g : A \to B'$, then $\langle f, g\rangle : A \to B \times B'$ denotes the uniquely determined resulting morphism. Likewise for $g : A' \to B'$, $f \times g : A \times A' \to B \times B'$.

3 General Setting and Example

Multiple interacting components of software architectures collectively realize the requirements of business domains. Describing the interactions between these systems and checking their global behavioural consistency is a general, well-known challenge in software engineering [4]. To address this challenge, model-driven software engineering utilizes abstract representations of the constituting systems and their interactions. Such a setting thus consists of an ensemble of *heterogeneously structured* components, which must guarantee the desired global behaviour.

In the sequel, we will speak of *local* or *individual* components, which are assembled into a *global* or *compound* system. As in [24], "system" is also used as a superordinate term for all kinds of artifacts, whether they are composite or not.

Using a general and formal coordination language for the interaction of behavioural components in the form of transition rules requires agreement on key concepts of behavioural systems. It turns out that the concepts "State" and (observational) "State Change" are common to almost all behavioural specifications, cf. the introductory remarks of [12]. Coalgebras (X, α) for some endofunctor $\mathcal{B} : \mathbb{C} \to \mathbb{C}$ on some category $\mathbb{C}$ comprise exactly these concepts: The structure map α assigns to each x in the state space X the observable causality exhibited in state x. The different natures of causalities (behaviour) are specified by different endofunctors $\mathcal{B}$.

Towards a formal underpinning for the described setting, we need to understand how aligning individual components by specifying their interactions on the one hand, and automatic generation (computation) of global execution behaviour of the compound system, on the other hand, are carried out. For this, we assume n behavioural specifications $\mathcal{B}_i : \mathcal{SET} \to \mathcal{SET}$ to be given for some $n \geq 2$ and fix individual behavioural systems $(S_i, \alpha_i) \in \mathcal{B}_i\text{-}\mathcal{C}oalg$.

As an example, we refer to the use case depicted in Fig. 1, where an instance of a `T-Junction`-Controller regulates the interaction of three `TrafficLights` `A`, `B`, and `C`. The T-junction controller (component (S_1, α_1)) and the behaviour of one traffic light (e.g., component (S_2, α_2)) are shown in the top and the bottom left part of Fig. 1. The resulting compound system is hinted at in the bottom right part. The operation, which takes as input the local components and "generates" the semantics of the compound system is visualized by arrows between the systems (blue in a colour display).

Whereas each traffic light is specified as a labelled transition system, the `TJunction` may be modelled as a BPMN[4]-model. The BPMN model specifies different phases to handle ($P1$ and $P2$). They are shown in the BPMN model and also in the two different snapshots of the compound system. The interaction with approaching vehicles may be modelled with a third formalism, e.g., a probabilistic transition system, which simulates exponentially distributed arrivals of buses or cars at one of the traffic lights. Aligning individual components by means of coordination languages, cf. [5], requires specifying coordination points (communication channels), e.g., if a request e of some approaching bus triggers the switch to phase 2 in the `TJunction` controller (an observation o), this transition must synchronize with input $i =$ `turnRed` of traffic light `A` and `C`. Moreover, `B` must simultaneously turn green. These synchronisations can be formalized with *synchronization algebras*, cf. [22], in this simple case, a partial map $\varphi : O \times I \to Act$, where O is the set of outputs of the controller like throw events or service calls in automatic tasks, I is the set of possible inputs to the respective traffic light, and Act is the set of observable actions of the compound system.

[4] Business Process Model and Notation

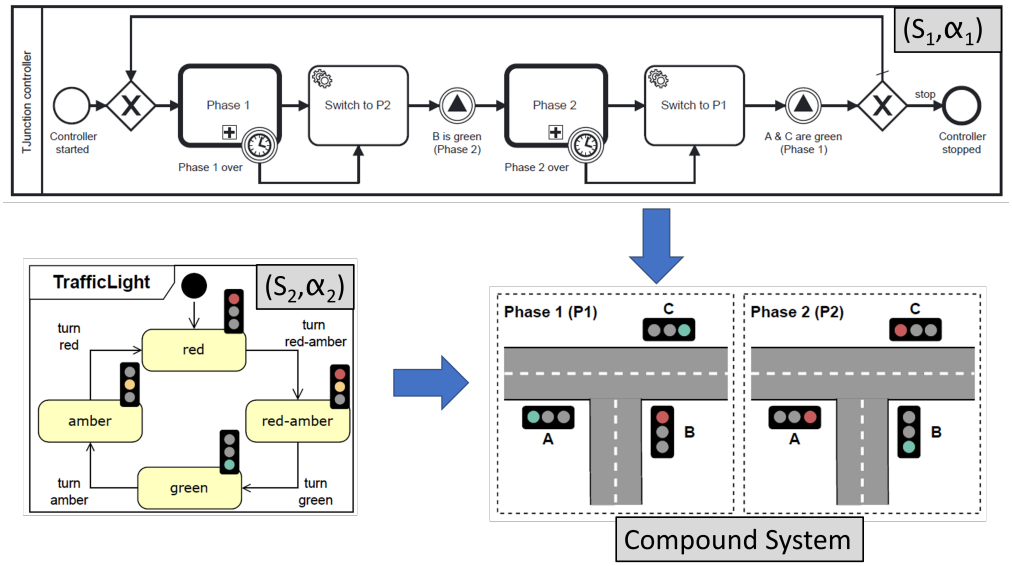


Figure 1 TJunction traffic control system and its individual components.

The global execution behaviour can be described by *premises* (transitions of the local components) and *conclusions* (the resulting actions taken by the compound system). If, for instance, $x \xrightarrow{e/o} x'$ in the BPMN-model and $y \xrightarrow{i} y'$ are possible, then $z \xrightarrow{\varphi(o,i)} z'$ is a global interaction of the components. Note that we obtain a respective conditional rule, but with different formats in its premises and conclusions: There is the Mealy-like notation in the first premise specifying output o in the BPMN process, when event e occurs, whereas the second premise specifies that the labelled transition system behaves like y', if, in state y, input i occurred. Furthermore, the compound system may be non-deterministic, such that the conclusion reads, "The system *may* behave like z', if in state z, action $\varphi(o,i)$ was performed".

Formally the interaction operation, which takes as input n states of the local components and outputs a state of the compound system, is based on an n-ary operation symbol $\mathtt{op} = \mathtt{interact} : s_1 s_2 \cdots s_n \to s_{n+1}$ of a suitable algebraic signature Σ, where sorts $s_1, s_2, ...$ reflect the structurally separated but interacting local components, and the compound system is based on a new behavioural specification $\mathcal{B}$ and requires a new sort $s_{n+1} \notin \{s_1, ..., s_n\}$.

We summarise the transfer from practical concepts to the bialgebraic formalism:

1. The individual components are based on behavioural endofunctors $\mathcal{B}_1, ..., \mathcal{B}_n$ and the compound system's behaviour is specified by another endofunctor $\mathcal{B}$. The individual components are coalgebras $(S_i, \alpha_i) \in \mathcal{B}_i\text{-}\mathcal{C}oalg$, from which the states of the compound system $(S, \alpha) \in \mathcal{B}\text{-}\mathcal{C}oalg$ arise as output of an application of an n-ary algebraic operation $\mathtt{op}$ which has as input the states of the individual components.
2. The semantics of the compound system is formalized by SOS rules of the form

$$\frac{x_1 \xrightarrow{E_1} x_1' \ \ldots \ x_n \xrightarrow{E_n} x_n'}{\mathtt{op}(x_1, \ldots, x_n) \xrightarrow{F} \mathtt{op}(x_1', \ldots, x_n')}$$

where E_i and F are differently structured terms over the coordination points, and x_i, x_i' are states of the individual component (S_i, α_i) for all $i \in \{1, ..., n\}$.

4 Background: Distributive Laws and Bialgebras

In this section, we recapitulate parts of [14] which are necessary to make the content of Sect. 5 complete and comprehensible. As in [18], we extend behavioural specifications $\mathcal{B}$ only to copointed coalgebras (see Def. 2 below), i.e. we consider the assignment $X \mapsto X \times \mathcal{B}(X)$ instead of $\mathcal{B}(X)$ in order to be able to use current states in the formulation of the conclusion of SOS rules. However, we do not generalise it further, i.e. we neither make use of free extensions for the algebraic specification functor as in the original work [27] nor cofree extensions of the behaviour functor, cf. [14].

Let $\mathbb{C}$ be an arbitrary category with finite products. The classical theory works with one fixed behavioural specification $\mathcal{B} : \mathbb{C} \to \mathbb{C}$ and an algebraic specification $\Sigma : \mathbb{C} \to \mathbb{C}$ which usually specifies the syntactical assembly of process terms (such as prefixing, alternative, and parallel, as well as interaction). In contrast to that, in Sect. 5, we will use Σ for the assembly of the compound system from the local components. Furthermore, let's define the functor

$$\mathcal{H} : \left\{ \begin{array}{ccc} \mathbb{C} & \to & \mathbb{C} \\ X \xrightarrow{f} Y & \mapsto & X \times \mathcal{B}(X) \xrightarrow{f \times \mathcal{B}f} Y \times \mathcal{B}(Y) \ . \end{array} \right. \tag{2}$$

Pairs $(\mathcal{H} : \mathbb{C} \to \mathbb{C}, \varepsilon : \mathcal{H} \Rightarrow ID_{\mathbb{C}})$ are usually called *copointed* functors in the literature, e.g. [14], i.e. $\mathcal{H}$ comes equipped with a comonadic "counit" ε. In our particular definition $\mathcal{H}$ is accompanied with counit $\pi_1 : \mathcal{H} \Rightarrow \mathcal{ID}_{\mathbb{C}}$, where $\pi_1 = ((\pi_1)_X : X \times \mathcal{B}(X) \to X)$ is the componentwise first projection. We will use only these special copointed functors.

▶ **Definition 1** (Distributive Law over $\mathcal{H}$). *A* Distributive Law *of Σ over $\mathcal{H}$ is a natural transformation*

$$\lambda : \Sigma\mathcal{H} \Rightarrow \mathcal{H}\Sigma$$

which is compatible with the counit, i.e. such that $(\pi_1)_\Sigma \circ \lambda = \Sigma\pi_1 : \Sigma\mathcal{H} \Rightarrow \Sigma$. ⌟

The extension from the original behavioural specification functor $\mathcal{B}$ to $\mathcal{H}$ also requires to consider special coalgebras for $\mathcal{H}$ [18]:

▶ **Definition 2** (Copointed $\mathcal{H}$-Coalgebra). *Let $\mathcal{H}$ be given as above. The category of copointed $\mathcal{H}$-coalgebras, written $\mathcal{H}$-$\mathcal{C}oalg_{co}$, is the full subcategory of $\mathcal{H}$-$\mathcal{C}oalg$ with those objects (X, α) satisfying $(\pi_1)_X \circ \alpha = id_X$.* ⌟

▶ **Proposition 3** (Copointed $\mathcal{H}$-Coalgebras are $\mathcal{B}$-Coalgebras). *The assignment $(X, \alpha) \mapsto (X, (\pi_2)_X \circ \alpha)$ extends to an isomorphism between categories $\mathcal{H}$-$\mathcal{C}oalg_{co}$ and $\mathcal{B}$-$\mathcal{C}oalg$.* ⌟

There is a canonical assignment from distributive laws over $\mathcal{H}$ to natural transformations

$$\rho : \Sigma\mathcal{H} \Rightarrow \mathcal{B}\Sigma \tag{3}$$

given by $\lambda \mapsto (\pi_2)_\Sigma \circ \lambda$. Using counit compatibility from Def. 1, the assignment

$$\rho \mapsto \langle \Sigma\pi_1, \rho \rangle \tag{4}$$

turns out to be inverse to the former, see Theorem 10 in [18]. Thus

▶ **Proposition 4** (Equivalent Representation of Distributive Laws). *The assignments (3) and (4) yield a bijection between distributive laws over $\mathcal{H}$ and natural transformations $\rho : \Sigma\mathcal{H} \Rightarrow \mathcal{B}\Sigma$.* ⌟

Note that natural transformations as in (3) are special cases of GSOS laws, where the syntax functor Σ is replaced by its free extension Σ^* in the codomain of ρ, thus enabling arbitrary terms in the target of the SOS rule conclusion.

Because λ is a natural transformation,

$$\Sigma_\lambda : \begin{cases} \mathcal{H}\text{-}\mathcal{C}oalg_{co} & \to & \mathcal{H}\text{-}\mathcal{C}oalg_{co} \\ (X,h) & \mapsto & (\Sigma(X), \lambda_X \circ \Sigma h) \end{cases}$$

and its dual construction

$$\mathcal{H}^\lambda : \begin{cases} \Sigma\text{-}\mathcal{A}lg & \to & \Sigma\text{-}\mathcal{A}lg \\ (g,X) & \mapsto & (\mathcal{H}g \circ \lambda_X, \mathcal{H}(X)) \end{cases}$$

extend to endofunctors, where the first indeed maps to $\mathcal{H}\text{-}\mathcal{C}oalg_{co}$ by the compatibility of counits in Def. 1. Σ_λ applied to a coalgebra yields behaviour of algebraically composed states and will play a major role in Sect. 5. $\mathcal{H}^\lambda$ will be used only in the present section.

For a distributive law λ, there is a new category, which yields a combination of operational and denotational models w.r.t. functors $\mathcal{B}$ (and thus $\mathcal{H}$) and Σ:

▶ **Definition 5** (Category of λ-Bialgebras). *Let $\lambda : \Sigma\mathcal{H} \Rightarrow \mathcal{H}\Sigma$ be a distributive law according to Def.1. The category λ-$\mathcal{B}ialg$ has objects arrow-pairs $\Sigma(X) \xrightarrow{g} X \xrightarrow{h} \mathcal{H}(X)$ with copointed (X,h) and for which*

$$\mathcal{H}g \circ \lambda_X \circ \Sigma h = h \circ g \tag{5}$$

Morphisms are those $f : X \to Y$, which are simultaneously Σ-$\mathcal{A}lg$- and $\mathcal{H}\text{-}\mathcal{C}oalg_{co}$-morphisms.

Using (5), one obtains

▶ **Proposition 6** ([14], Prop. 12). *There are the isomorphisms*

$$\Sigma_\lambda\text{-}\mathcal{A}lg \cong \lambda\text{-}\mathcal{B}ialg \cong \mathcal{H}^\lambda\text{-}\mathcal{C}oalg$$

where e.g. the second one is based on the assignment

$$(\ \Sigma(X) \xrightarrow{g} X \xrightarrow{h} \mathcal{H}(X)\ ) \mapsto ((g,X) \overset{(\Sigma h, h)}{\to} \mathcal{H}^\lambda(g,X))$$

on objects of the respective categories. ⌟

A consequence of this fact is the following proposition, for which we include a proof, because we need parts of it in Sect. 5:

▶ **Proposition 7** (Initial and Final Bialgebras, [14], Sect. 4.3). *If Σ admits an initial algebra (a,A) and if $\mathcal{H}\text{-}\mathcal{C}oalg_{co}$ has the final (copointed) coalgebra (Z,ζ), then the former uniquely extends to an initial object $\Sigma(A) \xrightarrow{a} A \xrightarrow{h_\lambda} \mathcal{H}(A)$ of λ-$\mathcal{B}ialg$ and the latter uniquely extends to a final object $\Sigma(Z) \xrightarrow{g^\lambda} Z \xrightarrow{\zeta} \mathcal{H}(Z)$ of λ-$\mathcal{B}ialg$.*

Proof. By Prop. 6 we can look for an initial object in $\mathcal{H}^\lambda\text{-}\mathcal{C}oalg$. But for any endofunctor $\overline{\mathcal{H}} : \mathbb{D} \to \mathbb{D}$ the carrier of the initial object in $\overline{\mathcal{H}}\text{-}\mathcal{C}oalg$ is just the initial object in $\mathbb{D}$, if it exists. Hence for $\overline{\mathcal{H}} = \mathcal{H}^\lambda$ and $\mathbb{D} = \Sigma\text{-}\mathcal{A}lg$, we obtain the initial $\mathcal{H}^\lambda\text{-}\mathcal{C}oalg$-object $a \overset{(\Sigma h_\lambda, h_\lambda)}{\longrightarrow} \mathcal{H}^\lambda a$, where $h_\lambda : A \to \mathcal{H}(A)$ is the unique Σ-$\mathcal{A}lg$-morphism out of the initial object. By the definition of $\mathcal{H}^\lambda$ this yields the commutative diagram

$$\begin{array}{ccc} \Sigma(A) & \xrightarrow{\quad a \quad} & A \\ {\scriptstyle \Sigma h_\lambda}\downarrow & & \downarrow{\scriptstyle h_\lambda} \\ \Sigma\mathcal{H}(A) & \xrightarrow[\mathcal{H}a \circ \lambda_A]{} & \mathcal{H}(A) \end{array} \tag{6}$$

turning $\Sigma(A) \xrightarrow{a} A \xrightarrow{h_\lambda} \mathcal{H}(A)$ into a λ-$\mathcal{B}ialg$-object (because h_λ is copointed by the following Prop. 8) but also the *initial* λ-$\mathcal{B}ialg$-object due to the assignment given in Prop. 6. The unique extension of the final object is dually obtained yielding the final λ-bialgebra $\Sigma(Z) \xrightarrow{g^\lambda} Z \xrightarrow{\zeta} \mathcal{H}(Z)$ for the unique $\mathcal{H}$-$\mathcal{C}oalg_{co}$-morphism g^λ to the final object. ◀

Using counit compatibility of π_1 in Def. 1 and initiality of (a, A), one also obtains

▶ **Proposition 8** (Copointedness of h_λ)**.** *h_λ is a copointed $\mathcal{H}$-coalgebra.* ⌟

The initial and final bialgebras from the proof of Prop. 7 yield a unique arrow $f : A \to Z$ in the commutative diagram

$$\begin{array}{ccccc} \Sigma(A) & \xrightarrow{a} & A & \xrightarrow{h_\lambda} & \mathcal{H}(A) \\ {\scriptstyle \Sigma f}\downarrow & & \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle \mathcal{H}f} \\ \Sigma(Z) & \xrightarrow{g_\lambda} & Z & \xrightarrow{\zeta} & \mathcal{H}(Z) \end{array} \tag{7}$$

f is simultaneously the coinductive extension of h_λ, i.e. its behavioural semantics, and the inductive extension of g^λ, i.e. the evaluation of Σ-terms in (g_λ, Z). The former statement is the important one. It gives the key statement of this section:

▶ **Observation 9.** The coinductive extension of copointed h_λ is an algebra homomorphism from the initial Σ-algebra. ⌟

5 From Local Components to Compound Systems Coalgebraically

5.1 The Theoretical Setting

Let $(S_1, \alpha_1) \in \mathcal{B}_1$-$\mathcal{C}oalg, \ldots, (S_n, \alpha_n) \in \mathcal{B}_n$-$\mathcal{C}oalg$ be n individual, local components and $\mathcal{B}$ a behavioural specification for the compound system, see item 1 in the summary on page 5. Related local components' interactions are based on an n-ary operation symbol `op` $:=$ `interact` and coordination points of $\mathcal{B}_1, \ldots, \mathcal{B}_n$ together with a synchronisation algebra[5] φ establish transition rules, cf. item 2 of the summary.

▶ **Example 10** (See Sect. 3)**.** A BPMN model can be encoded with the functor

$$\mathcal{B}_1 = (1 + O \times \_)^E,$$

where E are state-changing events like a timer event in the TJunction Controller, cf. Fig. 1. The set O defines outputs, e.g., `SwitchToP2` $\in O$, which must be synchronized with a call to a traffic light to turn red. Let $(S_1, \alpha_1) \in \mathcal{B}_1$-$\mathcal{C}oalg$ be such a component. Traffic lights are deterministic labelled transition systems based on

$$\mathcal{B}_2 = (1 + \_)^I,$$

where, for instance, $I = \{$`A.turnRed`, `A.turnGreen`, $\ldots\}$ is the input set I of traffic light `A`. Let $(S_2, \alpha_2) \in \mathcal{B}_2$-$\mathcal{C}oalg$ be such a component.

[5] For arbitrary n, these synchronisation descriptions will no longer be binary.

We define $Act := O + I + \{\tau\}$ and expect the compound system to change state depending on the used coordination points[6]. For this, we extend the involved sets by an idle action $*$, i.e. $X_* := X + \{*\}$ for $X \in \{O, I, Act\}$, cf. [22]. As usual, a transition with $*$ from state x to x' is possible, if and only if $x = x'$. We assume a synchronisation algebra $\varphi : O_* \times I_* \to Act_*$ for the synchronisation of a BPMN model with one traffic light. In the above example, we might have $\varphi(\texttt{SwitchToP2}, \texttt{A.turnRed}) = \tau$ (modelling a silent synchronisation), whereas some other value combinations like $\varphi(\texttt{SwitchToP2}, \texttt{A.turnGreen})$ are undefined. Whenever an output o is *uncoupled*, i.e., whenever the first component can evolve independently from the second for a transition with o, we let φ be undefined for pairs (o, i) for all $i \in I$, and define $\varphi(o, *) := o$. This is true, for example, if o is an outgoing signal like "B is green", which does not change the state of any traffic light, cf. Fig. 1. Similarly φ is undefined for (o, i) for all $o \in O$ and $\varphi(*, i) = i$ for uncoupled i. Finally, $\varphi(o, i) = * \iff o = i = *$, cf.[22].

The underlying algebraic signature will have three sorts: s_1 and s_2 for the states of the two local components and s_3 for the compound system. Because resulting transitions can be silent for different coordinations and hence result in non-determinism of the compound system, we define

$$\mathcal{B} := \wp_{fin}(Act \times \_) \qquad \lrcorner$$

The original work of [27] shows a one-to-one correspondence between sets of GSOS laws and natural transformations. We will show how we can follow this approach along the above-stated example. When we mention SOS rules, we exemplarily use notations in the context of our examples. Furthermore, whenever we write down $\varphi(o, i)$, we automatically assume this value to be defined.

The family of SOS-rules

$$\left(\frac{x \xrightarrow{e/o} x' \qquad y \xrightarrow{i} y'}{op(x, y) \xrightarrow{\varphi(o,i)} op(x', y')} \right)_{o \in O_*, i \in I_*} \tag{8}$$

describes the operational semantics of the compound system as a *heterogenous* interaction law. E.g., the controller's command to make traffic light A change to red is the law for $b := \texttt{SwitchToP2}$ and $i = \texttt{A.turnRed}$.

In the example, the state space S of the compound system must take into consideration the original state spaces by pairing S_1 and S_2, i.e., in this example

$$S = S_1 \times S_2. \tag{9}$$

Of course, in the general case, S can depend arbitrarily on the state spaces $S_1, ..., S_n$ of the local components.

5.2 Interaction Laws and Induced Coalgebra

In this section, we formalize the construction of the compound system from the local components, if its operational semantics is given as an SOS rule like in (8). This rule does not depend on concrete state spaces, hence it can be seen as an interaction law between

[6] As usual, τ models silent (unobservable) transitions.

systems of arbitrary state spaces X and Y. We claim that it can be encoded as a map, which decomposes into two factors, the first reflecting the premises given by the transitions of α_1 and α_2, and a second factor $\rho_{X,Y}$, which reflects the conclusions:

$$X \times Y \xrightarrow{\langle id,\alpha_1\rangle \times \langle id,\alpha_2\rangle} X \times \mathcal{B}_1(X) \times Y \times \mathcal{B}_2(Y) \xrightarrow{\rho_{X,Y}} \mathcal{B}(X \times Y) \tag{10}$$

We first define $\rho_{X,Y}$ in the context of our example:

▶ **Example 11** (Example 10 ctd)**.** Recall that we call $o \in O$ *coupled*, if there is $i \in I$ such that $\varphi(o,i)$ is defined and vice versa for $i \in I$. Otherwise, it is called *uncoupled*. Then, for $\mathcal{B}_1 = (1 + O \times \_)^E, \mathcal{B}_2 = (1 + \_)^I$, and $\mathcal{B} = \wp_{fin}(Act \times \_)$, we can define

$$\begin{array}{rcl} \rho_{X,Y}(x, f_1, y, f_2) & = & \{(\varphi(o,i),(x',y')) \mid o \neq * \neq i, (o,x') \in f_1(E), y' = f_2(i)\} \\ & \cup & \{(o,(x',y)) \mid (o,x') \in f_1(E), o \text{ uncoupled}\} \\ & \cup & \{(i,(x,y')) \mid y' = f_2(i), i \text{ uncoupled}\} \end{array}$$

where $f_1 : E \to 1 + O \times X$ an $f_2 : I \to 1 + Y$. It is easy to see that $(\rho_{X,Y})_{(X,Y)\in|\mathcal{SET}|^2}$ is natural in its parameters X and Y. ⌟

As in the classical theory, natural transformations as in Example 10 can now be used to *define* SOS-rules. For this let's define the copointed versions $\mathcal{H}_i := \mathcal{SET} \to \mathcal{SET}$ of the functors $\mathcal{B}_i$ as in (2) for $i \in \{1, ..., n\}$. The special assignment $(S_1, S_2) \mapsto S_1 \times S_2$ from (9) extends to a functor $\Sigma : \mathcal{SET}^2 \to \mathcal{SET}$ which yields natural transformation $\rho : \Sigma(\mathcal{H}_1 \times \mathcal{H}_2) \Rightarrow \mathcal{B}\Sigma : \mathcal{SET}^2 \to \mathcal{SET}$ in Example 11. However, we don't want to exclude additional dependencies, when constructing states of the compound system. E.g. additional supervising components or intermediate components like message queues may let the overall state space differ from the pure cartesian product of the local state spaces. Hence, we are interested in an arbitrary functor $\Sigma : \mathcal{SET}^n \to \mathcal{SET}$ for some $n \geq 2$ and correspondingly adapted natural transformations. Thus the appropriate definition in our context is

▶ **Definition 12** (Interaction Law)**.** *Let $\Sigma : \mathcal{SET}^n \to \mathcal{SET}$ be an arbitrary functor, $\mathcal{B}_1, \ldots, \mathcal{B}_n$, and $\mathcal{B}$ be $\mathcal{SET}$-endofunctors, and functors $\mathcal{H}_i : \mathcal{SET} \to \mathcal{SET}$ be defined as in (2), i.e. $\mathcal{H}_i(X) = X \times \mathcal{B}_i(X)$ for all $X \in \mathcal{SET}$ and all $i \in \{1, \ldots, n\}$. An* interaction law *is a natural transformation*

$$\rho : \Sigma(\mathcal{H}_1 \times \cdots \times \mathcal{H}_n) \Rightarrow \mathcal{B}\Sigma : \mathcal{SET}^n \to \mathcal{SET}.$$ ⌟

Similarly to the definition of Σ_λ in Sect. 4, this yields an assignment

$$\Sigma_\rho : \begin{cases} \mathcal{B}_1\text{-}\mathcal{C}oalg \times \cdots \times \mathcal{B}_n\text{-}\mathcal{C}oalg \longrightarrow \mathcal{B}\text{-}\mathcal{C}oalg \\ ((S_1,\alpha_1),\ldots,(S_n,\alpha_n)) \mapsto (\Sigma(S_1,...,S_n), \rho_{S_1,\ldots,S_n} \circ \Sigma(\langle id_{S_1},\alpha_1\rangle,\ldots,\langle id_{S_n},\alpha_n\rangle)) \end{cases} \tag{11}$$

which becomes a functor, because ρ is a natural transformation. Any n-tuple $(f_1, ..., f_n)$ with f_i a $\mathcal{B}_i$-$\mathcal{C}oalg$-morphism is mapped by Σ_ρ to the $\mathcal{B}$-$\mathcal{C}oalg$-morphism $\Sigma(f_1, ..., f_n)$.

▶ **Definition 13** (ρ-Induced Coalgebra)**.** *Given $(S_1,\alpha_1) \in \mathcal{B}_1\text{-}\mathcal{C}oalg, \ldots, (S_n,\alpha_n) \in \mathcal{B}_n\text{-}\mathcal{C}oalg$ and an interaction law ρ, the $\mathcal{B}$-coalgbra $\Sigma_\rho((S_1,\alpha_1),\ldots,(S_n,\alpha_n))$ is called the* ρ-induced coalgebra *of $(S_1,\alpha_1),\ldots,(S_n,\alpha_n)$. If the carrier sets are clear from the context, we just write $\Sigma_\rho(\alpha_1, ..., \alpha_n)$ for the ρ-induced coalgebra.* ⌟

Hence, the ρ-induced coalgebra is the compound system arising from the local components, when an SOS rule like (8), which is reflected in interaction law ρ, is applied.

▶ **Example 14** (Example 10 ctd)**.** In the example, we obtain the desired compound system, a $\mathcal{B}$-coalgebra with state space $S_1 \times S_2$ behaving as specified by the local components and the SOS-laws from (8). ⌟

5.3 Compositionality

Verification of correctness of composed systems should be guaranteed, if its components are already correct. Moreover, semantics preserving refactorings of local components should also preserve the semantics of the compound system. These behavioural correctness issues are often based on observational equivalence, hence we want observational equivalence to be preserved after the construction of the compound system from the local components. In this section, we formally define these aspects in the context of our setting.

▶ **Definition 15** (Observational Equivalence). *Let $\mathcal{F} : \mathcal{SET} \to \mathcal{SET}$, such that $\mathcal{F}$-$\mathcal{C}oalg$ admits a final object (Z, ζ). Let $(X, \alpha) \in \mathcal{F}$-$\mathcal{C}oalg$ and $u_\alpha : X \to Z$ be its coinductive extension. Two states $x, x' \in X$ are said to be* observationally equivalent*, written $x \sim_\alpha x'$, if (x, x') is contained in the kernel relation $ker(u_\alpha)$, i.e. if $u_\alpha(x) = u_\alpha(x')$.* ⌟

For future use, we state the following proposition, which easily follows, because for any $\mathcal{F}$-$\mathcal{C}oalg$-morphism $f : (X, \alpha) \to (Y, \beta)$, the coinductive extension satisfies $u_\alpha = u_\beta \circ f$:

▶ **Proposition 16** (Observational Equivalence). *With the same ingredients as in Def. 15, two states $x_1, x_2 \in X$ are observationally equivalent, if there is an $\mathcal{F}$-$\mathcal{C}oalg$-morphism $f : (X, \alpha) \to (Y, \beta)$ such that $f(x_1) = f(x_2)$.*[7] ⌟

Let $(u_{\alpha_i})_{i \in \{1,\dots,n\}}$ be the coinductive extensions of our local components,

$$\sim_i = ker(u_{\alpha_i}), \tag{12}$$

and some operation $\mathtt{op} : S_1 \times \cdots \times S_n \to A$ be given for some state set A of some $\mathcal{B}$-coalgebra. Furthermore, let $\sim$ be the kernel relation of its coinductive extension, then preservation of observational equivalence under $\mathtt{op}$ means

$$\forall i \in \{1, ..., n\} : x_i \sim_i x_i' \Rightarrow \mathtt{op}(x_1, ..., x_n) \sim \mathtt{op}(x_1', ..., x_n'), \tag{13}$$

i.e. observational equivalence is a compatible w.r.t. operation $\mathtt{op}$. It is well-known that a general definition of congruence on an algebra $a : \mathcal{F}(A) \to A$ for an endofunctor $\mathcal{F} : \mathbb{C} \to \mathbb{C}$ is as follows: A monomorphism $R \rightarrowtail^{\langle \pi_1, \pi_2 \rangle} A \times A$ is a congruence on a, if there is an algebra $r : \mathcal{F}(R) \to R$, for which the diagram

$$\begin{array}{ccccc} \mathcal{F}(A) & \xleftarrow{\mathcal{F}(\pi_1)} & \mathcal{F}(R) & \xrightarrow{\mathcal{F}(\pi_2)} & \mathcal{F}(A) \\ \downarrow{\scriptstyle a} & & \downarrow{\scriptstyle r} & & \downarrow{\scriptstyle a} \\ A & \xleftarrow[\pi_1]{} & R & \xrightarrow[\pi_2]{} & A \end{array} \tag{14}$$

commutes, cf. Theorem 3.3.5. in [12].

However, in the case of separated heterogeneously typed state sets of the local systems, a general definition of congruence must be based on the above-defined functor $\Sigma : \mathcal{SET}^n \to \mathcal{SET}$.

[7] Note that this proposition can even better be taken as the definition for observational equivalence, because it does not depend on the existence of a final object. We found it, however, more demonstrative to use Def. 15 for it.

▶ **Definition 17** (a-compatibility). *Let $A_1, ..., A_n, A$ be sets and*

$$a : \Sigma(A_1, ..., A_n) \to A$$

be a map. Furthermore let $(R_i \subseteq A_i \times A_i)_{i\in\{1,...,n\}}$ and $R \subseteq A \times A$ be a collection of $n+1$ binary relations with projections $\pi_1^i, \pi_2^i : R_i \to A_i$ for all i and $\pi_1, \pi_2 : R \to A$. The relation tuple $(R_1, ..., R_n, R)$ is said to be a-compatible*, if there is a map r, such that the following diagram commutes:*

$$\begin{array}{ccccc}
\Sigma(A_1,...,A_n) & \xleftarrow{\Sigma(\pi_1^1,...,\pi_1^n)} & \Sigma(R_1,...,R_n) & \xrightarrow{\Sigma(\pi_2^1,...,\pi_2^n)} & \Sigma(A_1,...,A_n) \\
\downarrow{\scriptstyle a} & & \downarrow{\scriptstyle r} & & \downarrow{\scriptstyle a} \\
A & \xleftarrow[\pi_1]{} & R & \xrightarrow[\pi_2]{} & A
\end{array}$$

⌟

▶ **Example 18** (op-compatibility). Let $R_i = \sim_i$ and $R = \sim$, cf. (12), then it is easy to see that in the case $\Sigma(X_1, ..., X_n) = X_1 \times \cdots \times X_n$ op-compatibility yields (13). ⌟

▶ **Observation 19.** *a-compatibility of $(R_1, ..., R_n, R)$ can thus be read as an implication:* If *pairs (a_i, a_i') are related via R_i,* then *a-images of corresponding elements of the set $\Sigma(A_1, ..., A_n)$ are related as well.* ⌟

Of course, the meaning of the term "corresponding" depends on the action of Σ.

It is not self-evident that observational equivalence is compatible with the syntactic structure of process terms in transition rules, see the counterexamples in [8] or violations of compositionality in the context of the π-calculus [21], Chapt. 12.4. However, in our setting, we can prove that interaction laws preserve observational equivalence. Note that this is almost evident in the above example, where $n = 2$ and $\Sigma(X, Y) = X \times Y$, because the image of the pair of coinductive extensions $u_{\alpha_1} : S_1 \to ...$ and $u_{\alpha_2} : S_2 \to ...$ of functor Σ_ρ is the $\mathcal{B}$-coalgebra-morphism $u = u_{\alpha_1} \times u_{\alpha_2}$, for which $((x_1, x_2), (x_1', x_2')) \in ker(u_{\alpha_1} \times u_{\alpha_2})$, if $(x_1, x_1') \in ker(u_{\alpha_1})$ and $(x_2, x_2') \in ker(u_{\alpha_2})$, which yields the desired result by Prop. 16.

The proof idea for the general case is to keep the local state spaces and the state space for the compound system separated as systems in their own right by assigning different sorts of the underlying algebraic specification to them and then apply (7) (i.e observation 9). For this, we must formalise the whole setting of system components and their interaction in one holistic *many-sorted* approach as follows. Recall that we assume $(S_1, \alpha_1) \in \mathcal{B}_1\text{-}\mathcal{C}oalg, \ldots, (S_n, \alpha_n) \in \mathcal{B}_n\text{-}\mathcal{C}oalg$ to be n individual, local components, then we define the endofunctor

$$\vec{\Sigma} : \left\{ \begin{array}{rcl} \mathcal{SET}^{n+1} & \to & \mathcal{SET}^{n+1} \\ (X_1, ..., X_n, X_{n+1}) & \mapsto & (S_1, ..., S_n, \Sigma(X_1, ..., X_n)) \end{array} \right.$$

with $\vec{\Sigma}(h_1, ..., h_n, h_{n+1}) := (\underbrace{id, ..., id}_{n \text{ times}}, \Sigma(h_1, \ldots, h_n))$ on function tuples. Intuitively, we define an algebraic signature with sorts $s_1, ..., s_n, s_{n+1}$ and "constants" of sort s_i the elements of S_i (for $1 \le i \le n$), as well as operation symbols with codomain s_{n+1}. Thus the term algebra has carrier sets $S_1, ..., S_n$, whereas the carrier of sort s_{n+1} comprises all terms arising from a single application of an operation symbol. We obtain

▶ **Proposition 20** (Initial Object of $\vec{\Sigma}$). *$\vec{\Sigma}$-$\mathcal{A}lg$ possesses an initial object with carrier $\mathbf{0} := (S_1, ..., S_n, \Sigma(S_1, ..., S_n)) = \vec{\Sigma}(\mathbf{0})$ and structure map $id_{\mathbf{0}} : \vec{\Sigma}(\mathbf{0}) \to \mathbf{0}$.*

Proof. Given a $\vec{\Sigma}$-algebra $(f_1, \ldots, f_n, f_{n+1}) : \vec{\Sigma}(X_1, \ldots, X_n, X_{n+1}) \to (X_1, \ldots, X_n, X_{n+1})$, it is easy to see that

$$\mathbf{0} \xrightarrow{(f_1,\ldots,f_n,f_{n+1}\circ\Sigma(f_1,\ldots,f_n))} (X_1, \ldots, X_n, X_{n+1})$$

establishes the unique algebra homomorphism from $id_{\mathbf{0}}$ to the given algebra. ◀

▶ **Theorem 21** (Interaction Laws preserve Observational Equivalence). *Let $\mathcal{B}_1, \ldots, \mathcal{B}_n$, and $\mathcal{B}$ be $n+1$ $\mathcal{SET}$-endofunctors, such that all corresponding categories of coalgebras admit a final coalgebra. Let $(S_1, \alpha_1) \in \mathcal{B}_1\text{-}\mathcal{C}oalg, \ldots, (S_n, \alpha_n) \in \mathcal{B}_n\text{-}\mathcal{C}oalg$ and $u_{\alpha_1}, ..., u_{\alpha_n}$ be their coinductive extensions. For functor $\Sigma : \mathcal{SET}^n \to \mathcal{SET}$ let an interaction law ρ be given as in Def. 12, and $\Sigma_\rho(\alpha_1, ..., \alpha_n)$ be the ρ-induced $\mathcal{B}$-coalgebra, cf. (11) and Def. 13, together with its coinductive extension u. Then the family $(ker(u_{\alpha_1}), ..., ker(u_{\alpha_n}), ker(u))$ of kernel relations is $id_{\Sigma(S_1,\ldots,S_n)}$-compatible.*

Thus by observation 19: If pairs (x_i, x_i') are observationally equivalent w.r.t. α_i, then corresponding elements in the set $\Sigma(S_1, ..., S_n)$ are observationally equivalent w.r.t. the ρ-induced coalgebra $\Sigma_\rho(\alpha_1, ..., \alpha_n)$. Thus observational equivalence carries over from the local components to the compound system.

Proof. To make reading easier, we give the proof of Theorem 21 for the special case $n = 2$. It easily carries over to the general case. Let $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}$ be the copointed versions of $\mathcal{B}_1, \mathcal{B}_2$, and $\mathcal{B}$ as in (2) and with this

$$\vec{\mathcal{H}} := \mathcal{H}_1 \times \mathcal{H}_2 \times \mathcal{H} : \mathcal{SET}^3 \to \mathcal{SET}^3$$

Let $\vec{\mathcal{B}} := \mathcal{B}_1 \times \mathcal{B}_2 \times \mathcal{B} : \mathcal{SET}^3 \to \mathcal{SET}^3$, then we have obtained the setting of Sect. 4 with $\mathbb{C} = \mathcal{SET}^3$, $\mathcal{H} := \vec{\mathcal{H}}$, and $\mathcal{B} := \vec{\mathcal{B}}$. Furthermore, we define

$$\vec{\rho} := (\alpha_1, \alpha_2, \rho) : \vec{\Sigma}\vec{\mathcal{H}} \Rightarrow \vec{\mathcal{B}}\vec{\Sigma} : \mathcal{SET}^3 \to \mathcal{SET}^3. \tag{15}$$

which is a natural transformation, because $(\alpha_{1,X_1} = \alpha_1)_{X_1 \in |\mathcal{SET}|}$ and $(\alpha_{2,X_2} = \alpha_2)_{X_2 \in |\mathcal{SET}|}$ are independent of their parameters X_1, X_2, resp. By Prop. 4 it corresponds to a distributive law $\vec{\lambda} : \vec{\Sigma}\vec{\mathcal{H}} \Rightarrow \vec{\mathcal{H}}\vec{\Sigma}$ of $\vec{\Sigma}$ over $\vec{\mathcal{H}}$, where by (4)

$$\vec{\lambda} = \langle \vec{\Sigma}\pi_1, \vec{\rho} \rangle. \tag{16}$$

Following the notation of Prop. 20, (6) becomes

$$\begin{array}{ccc} \vec{\Sigma}(\mathbf{0}) & = & \mathbf{0} \\ {\scriptstyle \vec{\Sigma}h_{\vec{\lambda}}}\downarrow & & \downarrow{\scriptstyle h_{\vec{\lambda}}} \\ \vec{\Sigma}\vec{\mathcal{H}}(\mathbf{0}) & \xrightarrow[\vec{\lambda}_{\mathbf{0}}]{} & \vec{\mathcal{H}}(\mathbf{0}) \end{array} \tag{17}$$

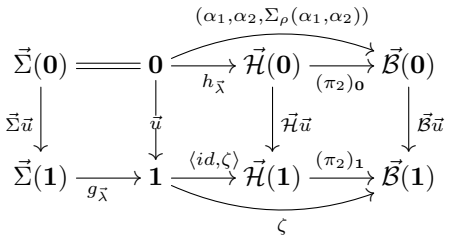
The first component $h^1_{\vec{\lambda}}$ of $h_{\vec{\lambda}}$ is the composition of the first components of the left and bottom arrow: $h^1_{\vec{\lambda}} = \vec{\lambda}^1_{\mathbf{0}} \circ (\vec{\Sigma}h_{\vec{\lambda}})^1 = \langle id_{S_1}, \alpha_1 \rangle \circ id_{S_1}$, because $\vec{\Sigma}$ is constant in the first component and similarly for the second component, hence

$$(h^1_{\vec{\lambda}}, h^2_{\vec{\lambda}}) = (\langle id_{S_1}, \alpha_1 \rangle, \langle id_{S_1}, \alpha_2 \rangle). \tag{18}$$

Thus, the third component of $\vec{\Sigma}h_{\vec{\lambda}}$ equals $\Sigma(\langle id_{S_1}, \alpha_1 \rangle, \langle id_{S_2}, \alpha_2 \rangle)$. By (16) the third component of $\vec{\lambda}_{\mathbf{0}}$ is the pair of the third component of $(\vec{\Sigma}\pi_1)_{\mathbf{0}}$ and $\rho_{(S_1,S_2)}$, hence

$$h^3_{\vec{\lambda}} = \langle id_{\Sigma(S_1,S_2)}, \rho_{S_1,S_2} \circ \Sigma(\langle id_{S_1}, \alpha_1 \rangle, \langle id_{S_2}, \alpha_2 \rangle) \rangle = \langle id_{\Sigma(S_1,S_2)}, \Sigma_\rho(\alpha_1, \alpha_2) \rangle \tag{19}$$

by Def. 13. Let $(\mathbf{1}, \zeta)$ be the final $\vec{\mathcal{B}}$-coalgebra (which exists, because it is taken componentwise), then by Prop. 3 $(\mathbf{1}, \langle id, \zeta \rangle)$ is final in $\vec{\mathcal{H}}\text{-}\mathcal{C}oalg_{co}$ and (7) is reflected in the left two squares in



(20)

with $\vec{u} = (u_{\alpha_1}, u_{\alpha_2}, u)$, see observation 9. By (15), (18), and (19) the triangle in the top right commutes.

Thus the coinductive extension $\vec{u}$ of the $\vec{B}$-coalgebra $(\alpha_1, \alpha_2, \Sigma_\rho(\alpha_1, \alpha_2))$ is a $\vec{\Sigma}$-algebra homomorphism and it is well-known that this makes $\vec{u}$'s kernel relation a congruence in the sense of (14) for $\mathcal{F} := \vec{\Sigma}$, see [12], Sect. 3.2., where $A = (S_1, S_2, \Sigma(S_1, S_2)) = \mathcal{F}(A)$, $R = (ker(u_{\alpha_1}), ker(u_{\alpha_2}), ker(u))$, hence $\mathcal{F}(R) = (S_1, S_2, \Sigma(ker(u_{\alpha_1}), ker(u_{\alpha_2}))$. Considering the third components only, shows that the family of kernel relations $(R_1 := ker(u_{\alpha_1}), R_2 := ker(u_{\alpha_2}), R := ker(u))$ is $id_{\Sigma(S_1,S_2)}$-compatible as desired. ◀

From Theorem 21 we also obtain

▶ **Corollary 22** (Sufficient Criterion for Compositionality). *Let $\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{B}$ and $\Sigma : \mathcal{SET}^n \to \mathcal{SET}$ be given as above. Let for all $i \in \{1, ..., n\}$ the $\mathcal{SET}$-endofunctors $\mathcal{H}_i$ and $\mathcal{H}$ be given as in (2), then compositionality holds for the heterogeneous scenario, if the computation of the compound system can be described by a natural transformation*

$$\rho : \Sigma(\mathcal{H}_1 \times \cdots \times \mathcal{H}_n) \Rightarrow \mathcal{B}\Sigma : \mathcal{SET}^n \to \mathcal{SET}.$$ ⌟

6 Related Work

Practical approaches. The general idea of transforming different behavioural formalisms to a single semantic domain in order to reason about crosscutting concerns is nothing new [6]. We mention only a few approaches: [17] developed consistency checking for sequence diagrams and statecharts based on CSP, while Petri nets were used for the same scenario in [29]. Nevertheless, all approaches utilize fixed types of transition systems and no common framework, which can capture all possible types of transition structures. In recent years, *co-simulation* of coupled heterogeneous systems has become popular and there is already a plethora of work on that topic [7]. In particular [5] tackles the problem of coordinating different models using a dedicated coordination language. However, the majority of these approaches lack theoretical underpinnings, and, to the best of our knowledge, co-simulated comprehensive behaviour has not been formulated coalgebraically.

SOS Framework, Distributive Laws and Compositionality. All important variants of SOS rules are described in [1] and we took most of its coalgebraic abstraction from the original work [27], further elaborated in [14], especially for copointed functors in [18], and probably formulated in the most general way in [12]. All important variations of distributive laws and connected aspects of compositionality are surveyed in Chapter 8 of [14]. Moreover, compositionality in the bialgebraic approach is a facet of the microcosm principle: The

behavior of a composed system involves an outer operator on $\mathcal{B}$-$\mathcal{C}oalg$, the composition of behaviors is an inner operator on the final object of $\mathcal{B}$-$\mathcal{C}oalg$, see [9], where the compositionality property is derived from a formalization of the microcosm principle for Lawvere theories.

Heterogeneity appears whenever different behavioral paradigms shall be combined. One of the first examples are hybrid systems, which combine discrete and continuous dynamics [11]. However, reasoning about operational semantics of arbitrary heterogeneously typed transition structures is usually treated by common abstractions of the different systems: E.g. the coordination of a Mealy machine and a probabilistic system can be investigated by reducing both systems to labelled transition systems and formulating interactions with LTS-based SOS rules. A different approach, which is closer to ours, is described in [13], where the combination of two distributive laws based on different behavioral specifications is investigated: So-called *heterogeneous* transition systems simultaneously carry two different coalgebraic stuctures $\mathcal{B}$ and $\mathcal{B}'$ and behavioural descriptions are based on natural transformations of the form $\Sigma(\mathcal{B} \times \mathcal{B}') \Rightarrow (\mathcal{B} \times \mathcal{B}')\Sigma$. However, the authors do not pick up the holistic view of our approach and do not investigate compositionality.

Categorically, heterogeneity leads to the general theory of *(co-)institutions*. [23] proves three different types of logics for coalgebras to be institutions. Another approach are parametrized endofunctors as comprehensive behavioural specifications, where the overall structure can be studied in terms of cofibrations [16]. [28] investigates co-institutions purely dual to classical institutions [25].

7 Future Work

We investigated the synchronisation of n local components to obtain a compound system. The idea was to introduce $n + 1$ sorts, which reflects the fact that the resulting compound system is obtained *in one step* from the locals. That excludes step-by-step synchronisation, i.e. the assembly of some components to an intermediate composed system, which in a later step is combined with other components, before the resulting global operational semantics is reached. The challenge in future work is to cope with an unsteady number of sorts for the arising intermediate systems. Similarly, our approach cannot directly be applied to asynchronous communications via intermediate components like message queues, object spaces, etc. It is a goal to derive formal underpinnings also in these cases.

Moreover, it is worth thinking about other types of extensions or refinements of local components and how they cause an impact on the composed system. If, for instance, a local system is conservatively extended [1], then we can ask the question whether the compound system is also conservatively extended. Furthermore, it is an open question, whether extensive refinements of the local systems and their interaction specifications can still be handled with interaction laws.

Finally, if additional system properties are imposed on the local behavioural models by modal logic formulae, the question arises, whether the use of co-forgetful functors in the translation of these formulae to the compound system [15] matches the framework proposed in the present paper. Altogether, the goal is to extend the first iteration of our work and, in future steps, develop more insight into the topic.

References

1 Luca Aceto, Wan J. Fokkink, and Chris Verhoef. Structural operational semantics. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 197–292. North-Holland / Elsevier, 2001. doi:10.1016/b978-044482830-9/50021-7.

2 Falk Bartels. On generalised coinduction and probabilistic specification formats: distributive laws in coalgebraic modelling. *Academisch Proefschrift, Vrije Universiteit Amsterdam*, 2004.

3 Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. In Jeanne Ferrante and Peter Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 229–239. ACM Press, 1988. doi:10.1145/73560.73580.

4 Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207, 2017.

5 Julien Deantoni. Modeling the behavioral semantics of heterogeneous languages and their coordination. In *2016 Architecture-Centric Virtual Integration (ACVI)*, pages 12–18. IEEE, 2016.

6 Gregor Engels, Jochen Malte Küster, Reiko Heckel, and Luuk Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In A Min Tjoa and Volker Gruhn, editors, *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, pages 186–195. ACM, 2001. doi:10.1145/503209.503235.

7 Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: a survey. *ACM Computing Surveys (CSUR)*, 51(3):1–33, 2018.

8 Jan Friso Groote and Frits Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and computation*, 100(2):202–260, 1992.

9 Ichiro Hasuo, Bart Jacobs, and Ana Sokolova. The microcosm principle and concurrency in coalgebra. In *International Conference on Foundations of Software Science and Computational Structures*, pages 246–260. Springer, 2008.

10 Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

11 Bart Jacobs. Object-oriented hybrid systems of coalgebras plus monoid actions. *Theoretical Computer Science*, 239(1):41–95, 2000.

12 Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2016. doi:10.1017/CBO9781316823187.

13 Marco Kick, John Power, and Alex Simpson. Coalgebraic semantics for timed processes. *Information and Computation*, 204(4):588–609, 2006.

14 Bartek Klin. Bialgebras for structural operational semantics: An introduction. *Theor. Comput. Sci.*, 412(38):5043–5069, 2011. doi:10.1016/j.tcs.2011.03.023.

15 Harald König and Uwe Wolter. Consistency of heterogeneously typed behavioural models: A coalgebraic approach. In Yamine Aït Ameur and Florin Craciun, editors, *Theoretical Aspects of Software Engineering - 16th International Symposium, TASE 2022, Cluj-Napoca, Romania, July 8-10, 2022, Proceedings*, volume 13299 of *Lecture Notes in Computer Science*, pages 308–325. Springer, 2022. doi:10.1007/978-3-031-10363-6_21.

16 Alexander Kurz and Dirk Pattinson. *Coalgebras and modal logic for parameterised endofunctors*. Centrum voor Wiskunde en Informatica, 2000.

17 Jochen Malte Küster. Towards Inconsistency Handling of Object-Oriented Behavioral Models. *Electron. Notes Theor. Comput. Sci.*, 109:57–69, 2004. doi:10.1016/j.entcs.2004.02.056.

18 Marina Lenisa, John Power, and Hiroshi Watanabe. Category theory for operational semantics. *Theor. Comput. Sci.*, 327(1-2):135–154, 2004. doi:10.1016/j.tcs.2004.07.024.

19 Dorian Leroy, Erwan Bousse, Manuel Wimmer, Tanja Mayerhofer, Benoit Combemale, and Wieland Schwinger. Behavioral interfaces for executable DSLs. *Software and Systems Modeling*, 19(4):1015–1043, 2020.

20 Giovanni Liboni and Julien Deantoni. Cosim20: An integrated development environment for accurate and efficient distributed co-simulations. In *Proceedings of the 2020 International Conference on Big Data in Management*, pages 76–83, 2020.

21 Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

22 Mogens Nielsen and Glynn Winskel. Models for concurrency. In *Handbook of Logic in Computer Science, Volume 4*. Oxford Science Publications, 1995.

23 Dirk Pattinson. Translating logics for coalgebras. In *International Workshop on Algebraic Development Techniques*, pages 393–408. Springer, 2002.

24 Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000. doi:10.1016/S0304-3975(00)00056-6.

25 Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2012. doi:10.1007/978-3-642-17336-3.

26 Ana Sokolova. Probabilistic systems coalgebraically: A survey. *Theoretical Computer Science*, 412(38):5095–5110, 2011.

27 Daniele Turi and Gordon Plotkin. Towards a mathematical operational semantics. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 280–291. IEEE, 1997.

28 U. Wolter. (Co)Institutions for Coalgebras. Reports in Informatics 415, Dep. of Informatics, University of Bergen, 2016.

29 Shuzhen Yao and Sol M Shatz. Consistency checking of UML dynamic models based on Petri net techniques. In *2006 15th International Conference on Computing*, pages 289–297. IEEE, 2006.

PAPER B

BEHAVIORAL CONSISTENCY IN MULTI-MODELING

Tim Kräuter, Harald König, Adrian Rutle, Yngve Lamo, Patrick Stünkel

Journal of Object Technology, Vol. 22, Issue 2, 2023, https://doi.org/10.5381/jot.2023.22.2.a9

JOT
JOURNAL OF
OBJECT TECHNOLOGY

Behavioral consistency in multi-modeling

Tim Kräuter*, Harald König†*, Adrian Rutle*, Yngve Lamo*, and Patrick Stünkel‡
*Western Norway University of Applied Sciences, Norway
†University of Applied Sciences FHDW Hannover, Germany
‡Haukeland Universitetssykehus, Norway

ABSTRACT Multiple heterogeneous interacting systems are needed to realize the requirements of complex domains. Describing the interactions between these systems and checking their global behavioral consistency is a general, well-known challenge in software engineering. To address this challenge, model-driven software engineering utilizes abstract representations of the constituting systems and their interactions, resulting in a *multi-model* representing the overall system. In such a multi-modeling setting, global consistency requirements must be satisfied by a set of heterogeneously typed models to guarantee a desired *global behavior*. In this paper, we propose a novel approach for behavioral consistency management of heterogeneous multi-models. The approach introduces a workflow in which we (i) define which behavioral models in the multi-model *may interact*, (ii) specify consistency requirements as *global behavioral properties*, (iii) align the individual models by specifying *how they interact*, (iv) generate a formal specification of the *global behavior*, and finally, (v) *check* the global behavioral properties, which should be satisfied by the multi-model. Our approach is decoupled from the particular formalism used in the generated formal specification, and we currently support graph transformations (Groove) and rewriting logic (Maude).

KEYWORDS Global behavioral consistency, Consistency verification, Multi-modeling, Heterogeneous models, Graph transformation, Rewriting Logic.

1. Introduction

Model-Driven Engineering (MDE) addresses the increasing complexity of software systems by employing models to describe the different aspects of the system. In this way, MDE promotes a clear separation of concerns and raises the abstraction level throughout the entire development process (France & Rumpe 2007). These models are then used to generate portions of the system, leading to increased productivity, and reduced errors (Brambilla et al. 2017). As multiple interacting systems are needed to realize the requirements of complex domains, a set of corresponding models would be needed to represent these systems and their interactions. Such a collection of interrelated models is called a *multi-model* (Boronat et al. 2009), which is usually heterogeneous, meaning it consists of models conforming to different modeling languages. Models in a multi-model contradicting each other can lead to problems during development, system generation, and system execution. Consequently, continuous multi-model consistency management during the development process is a significant issue for multi-models (Spanoudakis & Zisman 2001; Cicchetti et al. 2019).

Recent research describes methods to check the structural consistency of a multi-model (Stünkel et al. 2021; Klare & Gleitze 2019). Structural models, like UML class diagrams, describe structural aspects of systems, i.e., domain concepts and relations between these concepts. This is usually referred to as the denotational semantics of the software system, as it only describes the set of valid instances or states of the system. Structural models in a multi-model often contain related information. Thus, current approaches define so-called *commonalities* to explicate these relationships and keep the information consistent. Afterward, these commonalities can be used to get a comprehensive view of the global system, for example, by *merging* all models into a global view. Structural consistency can then be verified using this global view.

Nevertheless, approaches to multi-model consistency management must also include a means to maintain *behavioral con-*

JOT reference format:
Tim Kräuter, Harald König, Adrian Rutle, Yngve Lamo, and Patrick Stünkel. *Behavioral consistency in multi-modeling*. Journal of Object Technology. Vol. 22, No. 2, 2023. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2023.22.2.a9

sistency since behavioral models, like Business Process Modeling Notation (BPMN) models, are associated with execution semantics describing dynamic aspects of the system (Object Management Group 2013). For example, multi-models consisting of different interacting behavioral models are used when modeling embedded and cyber-physical systems (Vara Larsen et al. 2015).

Several approaches exist for checking the consistency of specific pairs of behavioral models. For example, consistency checking for sequence diagrams and statecharts was implemented using Petri nets (Yao & Shatz 2006) and Communicating Sequential Processes (CSP) (Küster & Stehr 2003). Moreover, the co-simulation field tackles the simulation of interacting models by composing their individual simulations. To facilitate composing simulations, the Functional Mock-up Interface (FMI) standard for simulation models and simulation tools such as Ptolemy (J. Eker et al. 2003) were developed. However, there is no approach to define and *check consistency* of *arbitrary many* behavioral models.

We propose a novel approach for consistency management of heterogeneous multi-models, which allows us to define and check *global* behavioral properties. Our approach facilitates specifying *interactions* between multiple potentially heterogeneous behavioral models, which are used to generate a specification of the global behavior. The approach is decoupled from the particular formalism used in the specification, and currently, we can generate specifications in two different formalisms. The generation of the global behavior specification is *fully automatic* and results in Graph Transformation (GT) rules (or, respectively, term rewriting rules) executable in Groove (Maude). Afterward, we can use the built-in verification mechanisms in Groove (Maude) to check the previously defined global behavioral properties.

Our approach is based on two fundamental concepts: *state* and *state-changing elements*. The state structure of each participating behavioral language must be explicitly defined to infer how global states are structured. Furthermore, state-changing elements must be identified in each participating behavioral language. Thus, state-changing elements serve as a minimal *behavioral interface* to uniformly define interactions for heterogeneous models. Our approach applies to behavioral formalisms where these two concepts can be found, which is the case for most formalisms with discrete state variables (see related work in section 6).

The proposed approach partly resembles the state of the art approaches for structural consistency. Interactions correspond to commonalities as both add necessary inter-model information. Then, a global representation of the system's behavior/structure is constructed by composing the individual models using the interaction/commonality information. Generating a global behavior specification is similar to merging structural models into a global view. To achieve a global view for structural models, one introduces a base language in which individual models and the global view can be represented. Likewise, generating our global behavior specification is based on the two fundamental concepts of *state* and *state-changing elements*. In summary, we adapt the three steps for structural consistency management: Alignment, Verification, and Reconciliation as proposed in (Stünkel et al. 2021) to a multi-model containing behavioral models.

This contribution builds on our previous publication (Kräuter 2021). However, we refined the behavioral consistency management workflow into five steps and introduced new key concepts such as state, state-changing elements, interactions, and behavioral relationships.

The remainder of this paper is structured as follows. We introduce a simplified use case (section 2) before explaining our behavioral consistency management approach in detail (section 3). Afterward, we show how we can use the GT toolset Groove to check behavioral consistency (section 4). Furthermore, we discuss the potential limitations of our approach in section 5. Finally, we examine related work in section 6 and conclude in section 7.

2. Use Case

This section motivates our approach with a simplified use case in which a traffic management system is developed to guide the traffic at a T-Junction with three traffic lights. The traffic management system should control the traffic by switching between the two traffic phases highlighted in figure 1. In addition, it must fulfill the following two requirements. First, it must guarantee safe traffic by correctly changing the three traffic lights A, B, and C. Second, it should prioritize arriving buses, i.e., switch the traffic lights quicker than usual to let an approaching bus pass (early green). This so-called bus priority signal is a widely implemented technique to improve service and reduce delays in public transport.

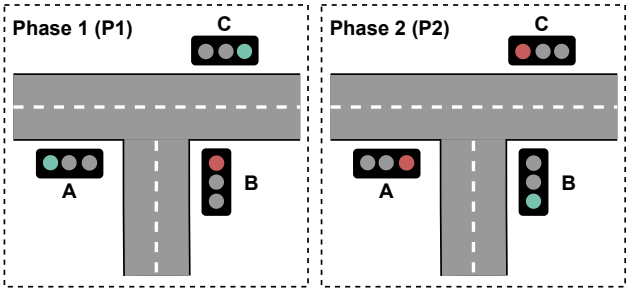


Figure 1 Traffic phases of a T-Junction

To develop the behavior of the traffic management system, we follow an MDE approach. First, we model the behavior of a traffic light as a Unified Modeling Language (UML) state machine. Then we use BPMN to model the different traffic phases of the T-Junction, including the prioritization of approaching buses.

Using different behavioral modeling languages in the use case has two reasons. First, two software development teams might work on the system in parallel but prefer different modeling languages. Second, each team can choose the most appropriate modeling language for defining their part of the system. In this use case, the behavior of a traffic light and a T-Junction differs significantly in complexity and requirements, resulting in

the use of two different behavioral modeling languages, namely UML state machines, and BPMN.

The behavior of a traffic light is straightforward since it uses only three colors to guide the traffic. Figure 2 shows a typical traffic light that switches from red to red-amber, green, amber, and back to red. The start state of the traffic light in figure 2 is red but can be any of the four possible states.

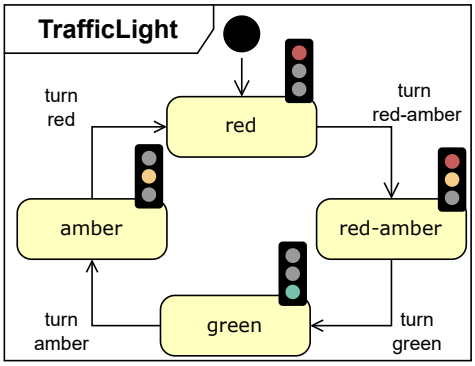


Figure 2 Traffic light state machine model

However, the T-Junction's behavior is more complex since it should coordinate the three traffic lights and communicate with approaching buses to implement bus priority. Consequently, we are using BPMN to model this aspect of the system's behavior and utilize BPMN message and signal events to implement the communication with approaching buses.

We model two processes, one for the T-Junction and one for the Bus. Each process is modeled in its BPMN *pool*. A pool is depicted as a horizontal lane with a name on the left. Message flows (arrows with dashed lines) are only allowed between two different pools.

Figure 3 shows how a possible controller for a T-Junction behaves in the traffic management system. When a TJunction controller is started, we assume that the traffic lights are showing the colors according to phase 1 (see figure 1). Thus, the controller enters a subprocess called phase 1 (see top right in figure 4), which we describe together with the subprocess called phase 2 later. However, when a fixed amount of time has passed, the subprocess is interrupted by the attached timer boundary event. Then, the controller executes the next activity and switches to phase 2. The controller will pass a throwing signal event before entering a subprocess for phase 2 and repeat the same steps. This signal event represents a broadcast to all buses waiting for traffic light B to become green. After switching back from phase 2 to phase 1 and signaling that traffic lights A and C are green, the controller can stop or execute the described steps again. Typically, the controller does not stop, indicated by the default sequence flow going back to the beginning of the process.

Figure 4 shows the communication of a bus with the subprocess phase 1. The BPMN model and communication for phase 2 of the controller can be defined accordingly.

The phase 1 model uses an event-based gateway to respond to two different kinds of messages. First, the traffic light status can be requested, which is answered by sending a message declaring that the traffic lights A and C are green while B is red. Moreover, early green for traffic light B can be requested. This request ends the subprocess, and the controller immediately switches to phase 2 (see figure 3), which results in the traffic light B turning green.

The bottom of Figure 4 shows the controller for a bus parameterized with direction B. It will first request the traffic light status to determine if traffic light B is green. If it is green, the bus can pass the junction. However, if it is red, the bus requests to change B to green and waits for a signal that the controller has changed the traffic light. After receiving the signal, the bus passes the junction. A BPMN model for a bus controller parameterized with the direction A or C looks nearly identical. In addition, the bus controller communicates with the phase 2 subprocess, which we only hint at in figure 4. The phase 2 subprocess has the same structure as the phase 1 subprocess but reports that A and C are red, while B is green. Similarly, it terminates if green is requested for A or C. The full model and all other models are available in (Kräuter 2023).

Having developed behavioral models for the system, we want to check the previously stated *safe traffic* requirement while buses are prioritized. We can lower the overall development cost if we find bugs related to these requirements as early as possible during system development. However, the traffic light model is currently not related to the T-Junction and bus models while the T-Junction is supposed to control the traffic lights, for example, when it switches between the two traffic phases. In addition, the system has to manage multiple slightly different instances of the behavioral models. For example, there are three traffic lights at one T-Junction starting in different states—i.e., showing different colors—and buses approaching the T-Junction from one of the three directions. Consequently, we need a model of the system to allow us to define interactions between the models and configure instances of the behavioral models contained in the multi-model.

The resulting model called the *System Relationship Model (SRM)* is shown in figure 5 using a graph-based syntax. It contains one node for each behavioral model and arrows to depict behavioral relationships, leading to possible interactions. In addition, it contains enumerations to parameterize the behavioral models. A TJunction has three associated TrafficLights, A, B, and C, and a set of currently approaching Buses. A TrafficLight has four possible TrafficLightStates and an attribute to define its startState. A Bus has a direction that indicates which TrafficLight of the T-Junction it is approaching.

Finally, using the SRM, we can define a test configuration of our traffic management system to check its requirements. Figure 6 depicts the test system configuration as an instance of the SRM. First, it contains three instances of the traffic light behavioral model, representing the three traffic lights, A, B, and C. Second, it contains an instance of the T-Junction behavioral model connected to the three traffic lights and two instances of the bus behavioral model. Thus, the test system configuration describes a system that controls one T-Junction with three traffic lights and two buses approaching from directions A and B.

First, we would like to check the safe traffic requirement. Since we only want to check system conformance concerning

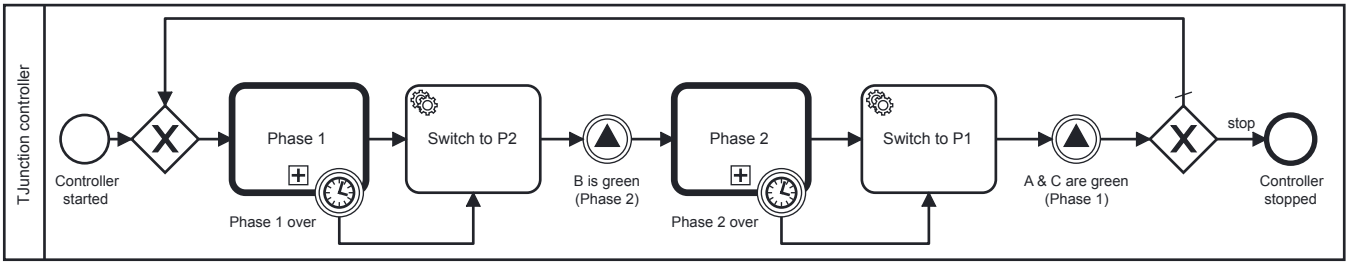


Figure 3 Model for a TJunction controller

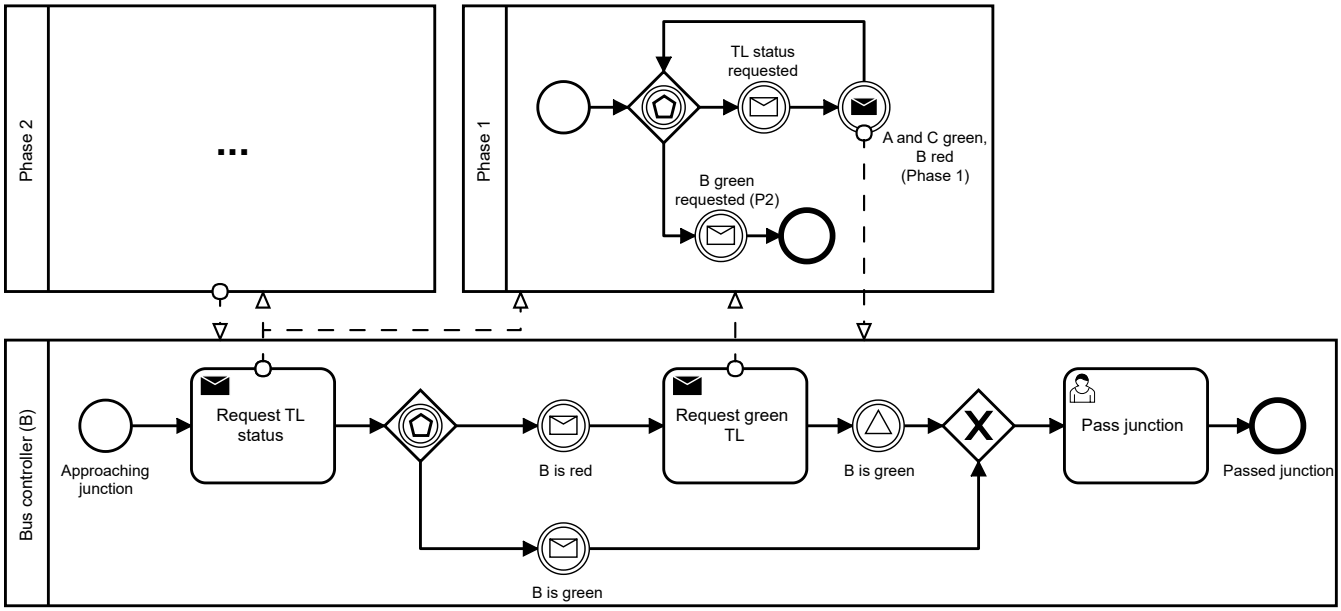


Figure 4 Model for a bus with direction B and its communication with a T-Junction

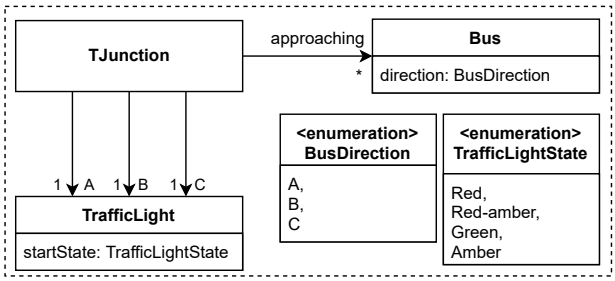


Figure 5 System relationship model of the traffic management system

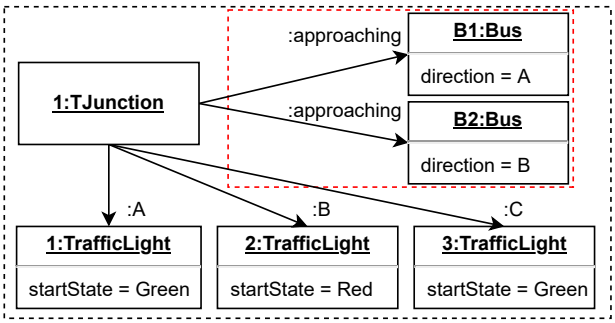


Figure 6 Test system configuration

the two traffic phases, we do not need to include the two buses depicted in the red dotted square in figure 6 in the analysis. We cannot simply assert that the system is either in phase 1 or phase 2 since there are intermediate states during the transition between the two phases, which are allowed. By consulting figure 2, we can, for example, expect a state in which traffic lights A and C are amber, and traffic light B is red-amber before reaching phase 2. However, we can define *safe traffic* as the absence of *unsafe traffic*, which is easier to define.

For the T-Junction, unsafe traffic occurs if traffic light A is green or amber and traffic light B is green or amber simultaneously. In addition, the same state combinations are forbidden for traffic lights B and C. Unsafe traffic occurs only in these situations since green and amber mean that cars are allowed to pass, while red (red-amber) means cars are not (not yet, respectively) allowed to pass. We can formalize the consistency requirements

as safety properties in Linear Temporal Logic (LTL), i.e., states that should never be reached. The resulting global properties (1) and (2) are the following, assuming the existence of atomic propositions for each traffic light state.

$$\Box\neg((A_{green} \vee A_{amber}) \wedge (B_{green} \vee B_{amber})) \quad (1)$$
$$\Box\neg((C_{green} \vee C_{amber}) \wedge (B_{green} \vee B_{amber})) \quad (2)$$

If we include buses B1 and B2 in the system, we want to check that they cannot pass when their traffic light is red or red-amber. Concretely, this means the Pass Junction activity should not execute while the corresponding traffic light is red or red-amber. We formalize these requirements again by using LTL safety properties (3) and (4), where the atomic proposition $B1_{passing}$ and $B2_{passing}$ represent that Pass Junction (see figure 4) has started but not finished yet.

$$\Box\neg(B1_{passing} \wedge (A_{red} \vee A_{red-amber})) \quad (3)$$
$$\Box\neg(B2_{passing} \wedge (B_{red} \vee B_{red-amber})) \quad (4)$$

However, to check the global properties, we must execute the system with the behavior specified in the behavioral models according to the test configuration. This is not straightforward since the multi-model of the use case consists of a SRM relating two heterogeneously typed behavioral models. In addition, the system configuration instantiates the traffic light and bus behavioral models multiple times with different parameters. Furthermore, we face the problem that the models are not independent. For example, the T-Junction controller must decide when the traffic lights A, B, and C switch states. Thus, if we were to run the models independently in parallel, the properties would be violated.

A multi-model is behaviorally consistent if it satisfies all of its behavioral properties. A behavioral property is given in temporal logic, for example, LTL in the use case, and is characterized as *local* if it constrains only one model and as *global* if it spans two or more models in a multi-model. Furthermore, global properties depend on the system configuration, i.e., the instance of the SRM used. In the remainder of this paper, we will describe our approach to address behavioral consistency in multi-modeling and apply it to this use case.

3. Behavioral consistency management

Figure 7 depicts our approach to behavioral consistency management as a BPMN diagram.

Our approach consists of five steps.

1. We define a *System Relationship Model (SRM)* describing which behavioral models may interact.
2. We specify consistency requirements as global behavioral properties for the SRM.
3. We define *interactions* between the behavioral models using the SRM.
4. We automatically generate a specification of the specified global behavior using the interactions and the SRM.
5. Given a system configuration, we check the global behavioral properties using the generated specification.

The first three steps are marked as manual and must be completed to use our approach in a given use case. However, the last two steps are automated and reusable in any use case. In the following sections, we will describe each step in detail, highlighting what a modeler must repeatedly do for each use case and what must only be done once for each participating behavioral language. In addition, figure 18 at the end of the paper gives an overview of all the new concepts and how they are applied to the use case.

3.1. Define the system relationship model

As mentioned, a set of behavioral models might be used to describe the behavior of a software system. Each model conforms to its metamodel, corresponding to the behavioral language used to specify the model. The metamodel ensures that models specified in the corresponding languages are well-defined and machine-readable. This is crucial when automating parts of the consistency checking.

The use case utilizes state machine and BPMN models, which conform, respectively, to the metamodels of state machines (see figure 8) and BPMN (see figure 9). The metamodel of state machines is defined by a UML class diagram. In addition, the clouds depict the concrete syntax that we use to denote the models conforming to the metamodel. The traffic light model in figure 2 uses this concrete syntax.

A StateMachine has a startState and transitions, whereas each Transition connects two States. The states of a state machine are not explicitly modeled but can be derived from the transitions of a state machine. Furthermore, isolated states are not allowed.

The metamodel for BPMN (see figure 9) is defined analogously to the one of state machines. A BPMN Process contains a set of FlowNodes connected by SequenceFlows. FlowNodes and SequenceFlows are FlowElements, inheriting an id and a name. A FlowNode can be an Activity, Gateway, or Event. All special activities, gateways, and events are defined in the BPMN specification (Object Management Group 2013).

Behavioral models *interact* to realize the global system behavior. A SRM describes which behavioral models exist in the system and whether they are behaviorally related, i.e., they may interact during execution. In our approach, we define a system relationship metamodel to specify these relationships formally. The constructed SRM is use-case specific, while the metamodels for the participating languages must only be defined once.

We are using a graph-based syntax to define SRMs (see figure 5), where each node corresponds to a behavioral model (typed by a BehavioralMetamodel), while each arrow corresponds to a BehavioralRelationship (see concrete syntax depicted in clouds). For example, the SRM for the use case (see figure 5) has three behavioral relationships from TJunction to TrafficLight since a TJunction controller interacts with three

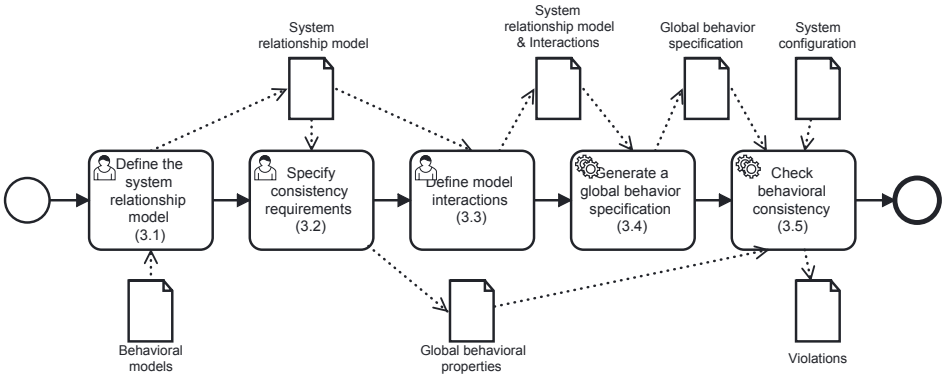


Figure 7 Behavioral consistency management workflow

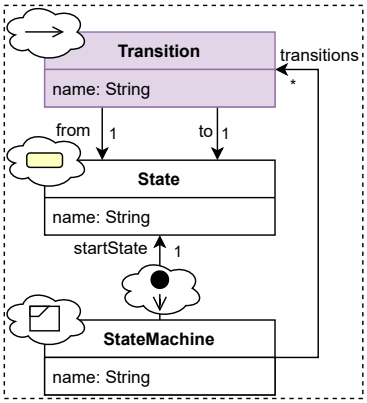


Figure 8 Finite state machine metamodel

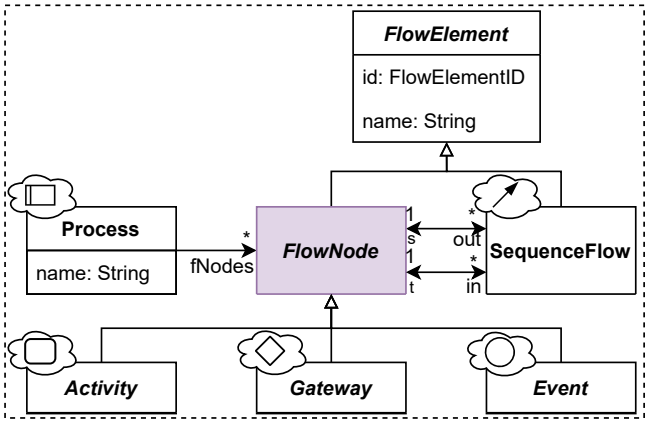


Figure 9 Simplified BPMN metamodel (Object Management Group 2013)

different traffic lights A, B, and C. Furthermore, there is a behavioral relationship from TJunction to Bus because we want to check the safety properties (3) and (4). To summarize, behavioral relationships define *which* behavioral models *may* interact, while the interactions in the next step of the workflow describe *how* they interact.

In addition, we allow enumerations and attributes in SRMs. These may be used as parameters, e.g., to define the start state in a state machine (see figure 5). Different instances of the SRM can be used to analyze the global behavior of *different* system configurations by changing the parameters.

3.2. Specify consistency requirements

In this step, we specify behavioral consistency requirements as global behavioral properties. These properties are defined using temporal logic, for example, LTL as in the use case. Theoretically, any temporal logic can be used together with our approach, however, in practice, the underlying system which runs the generated specifications must support it. Furthermore, we agree with (Meyers et al. 2014) that modelers are usually unfamiliar with temporal logic. Lifting property specification to the domain-specific level is a promising idea that fits our approach. Nevertheless, for now, a modeler must define temporal logic properties.

Temporal logic properties are built upon a set of atomic propositions which are either true or false in a given state. Each behavioral language (e.g., the BPMN) specification defines how these states are represented. Furthermore, the transitions between these states depend on the semantics of the behavioral models. In our approach, the first fundamental concept is to make *state* structure explicit. We will call the models for the state structure *snapshot metamodels* and specify them using class diagrams.

For example, in the use case, we define snapshot metamodels for state machines and BPMN. A state machine is in one state at a time, as shown in the snapshot metamodel on the left of figure 10. We are reusing the concrete syntax elements from the state machine metamodel (see figure 8) for the snapshot metamodel. In addition, each snapshot metamodel has a root element in our approach, highlighted in light blue.

The snapshot metamodel for BPMN is based on a Token distribution as described in the BPMN specifications (Object Management Group 2013) (see on the right of figure 10). The root element ProcessSnapshot has tokens and subprocesses. A Token indicates where it is located in the BPMN model using its position attribute. A valid position is the id of a FlowElement (see figure 9). Also, for the snapshot metamodel of BPMN we

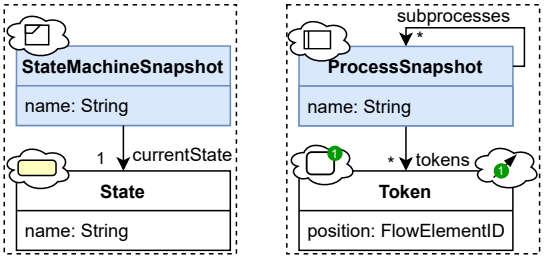


Figure 10 FSM and BPMN snapshot metamodels

reuse the concrete syntax of the BPMN metamodel. In addition, Tokens are highlighted with green bubbles in the middle of sequence flows and the top right of an activity.

Each instance of a snapshot metamodel can represent an atomic proposition since a system can either be in the state specified by this instance or not. Since instances of snapshot metamodels are essentially graphs, they can be matched to a given system state uniformly. Thus, using the snapshot metamodels to create atomic propositions, we can specify local behavioral properties for any modeling language. However, to specify global behavioral properties, we must combine the information of the SRM with the snapshot metamodels. Each behavioral model is typed by a behavioral metamodel, which has a snapshot metamodel describing its state structure. Thus, we know how states of behavioral models are represented when they are instantiated. For example, figure 11 shows how to specify the atomic propositions A_{green} and $B1_{passing}$, used in the global properties in section 2. Snapshot links connect instances of behavioral models with root element instances of the corresponding snapshot metamodel.

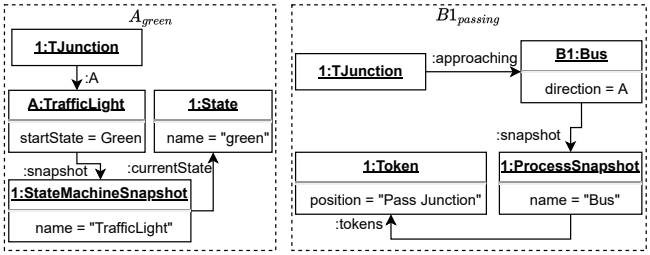


Figure 11 Atomic propositions A_{green} and $B1_{passing}$

To make formulating atomic propositions less cumbersome, one can use the concrete syntax of the individual snapshot metamodels. For example, figure 12 shows the same atomic propositions as figure 11 but uses the introduced concrete syntax.

With the defined atomic propositions as ingredients, one can use temporal logic to define global behavioral properties such as the properties (1)-(4) in section 2. It is worth noting that the defined atomic propositions are *model-specific*, meaning they exactly fit the given multi-model. Thus, property definition (including atomic propositions) is done for each use case, while snapshot metamodels for behavioral languages must only be defined once.

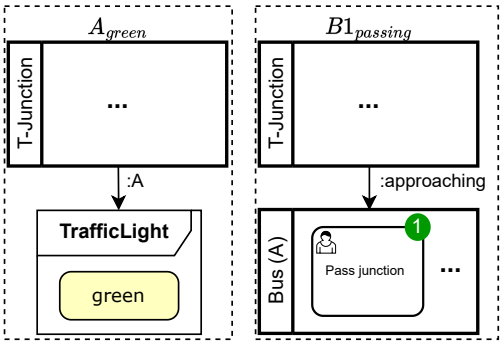


Figure 12 Concrete syntax for A_{green} and $B1_{passing}$

3.3. Define model interactions

In contrast to structural multi-modeling, we call behavioral inter-model relationships *interactions* since they carry behavioral meaning while commonalities carry structural meaning (Stünkel et al. 2021; Klare & Gleitze 2019). To specify interactions between different behavioral models, we define an interaction language given by the system relationship metamodel in figure 13. In addition, we introduce the second fundamental concept of *state-changing elements*. Each BehavioralMetamodel specifies a set of StateChangingElements. For example, a state machine defines states and transitions, but only the transitions describe how the states in a state machine change. Thus, the transitions are the state-changing elements of a state machine (highlighted in purple, see figure 8). Similarly, the flow nodes are the state-changing elements of a BPMN process (highlighted in purple, see figure 9)[1]. The interaction of behavioral models is only possible through instances of state-changing elements. Thus, state-changing elements function as a minimal *behavioral interface* to define interactions for heterogeneous behavioral languages uniformly.

Our approach is based on the requirement that state-changing elements can be identified in metamodels for any used behavioral formalism. This requirement is not difficult to meet since behavioral modeling languages with a discrete state must have some *observable* construct to describe state changes. Inspecting other behavioral languages, such as Petri Nets or activity diagrams, shows that identifying state-changing elements (transitions and activity nodes, respectively) is unproblematic.

An Interaction has a sender, a set of receivers, and a type. Currently, there is only the synchronous InteractionType. However, more interaction types, for example, asynchronous interactions or interactions with message passing, could be added in the future. We model the role of sender and receiver in interactions to accommodate these interaction types in the future. The two roles are not needed for synchronous interactions and are hidden from the modeler in the concrete syntax used later in listing 1. Furthermore, asynchronous interactions can be modeled using two synchronous interactions with an additional behavioral model, such as a queue. Each InteractionReceiver references one BehavioralRelationship and one StateChangingElement.

[1] One exception is the event-based gateway, which is not part of the state-changing elements.

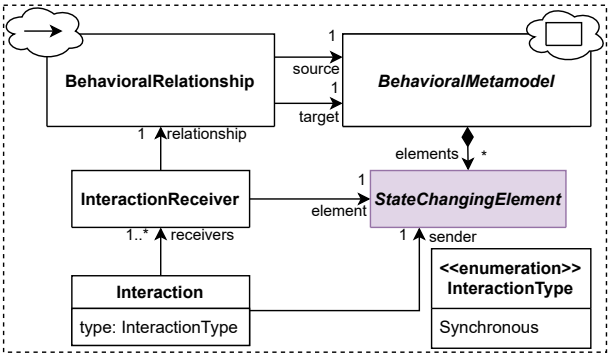


Figure 13 System relationship metamodel

Adding new interaction types only impacts steps 3.3 and 3.4 in our approach (see figure 7). Concretely, a new interaction type has to be added to the enumeration in the system relationship metamodel and must be accounted for in the concrete syntax for interactions. Then, the semantics of the new interaction type must be added to the generation of the global behavior specification in step 3.4.

The sender and the elements of the InteractionReceivers describe a state change for their behavioral models. By connecting them with a synchronous interaction, we define simultaneous state changes in one atomic step. Consequently, an interaction defines a synchronization between behavioral models. Generally, models behave in a distributed independent fashion until they reach a state-changing element that is part of an interaction. To execute these state changes, the models must then interact, i.e., synchronize as described. In addition, one can only define interactions for state-changing elements if a behavioral relationship connects their behavioral models (see constraint (5)). We use "." in constraints to navigate along associations. For example, i.receivers means following all receivers links for an Interaction object, resulting in a set of InteractionReceiver objects.

$$\begin{aligned} &\forall i \in Interaction : \forall r \in i.receivers : \\ &\quad r.element \in r.relationship.target.elements \wedge \\ &\quad i.sender \in r.relationship.source.elements \end{aligned} \tag{5}$$

We allow the definition of as many interactions as desired. Two interactions are not allowed to share state-changing elements (see constraint (6)). If such a situation occurs, it must be resolved by the modeler by deleting one of the interactions or merging the two interactions into one.

$$\begin{aligned} &\forall i_1, i_2 \in Interaction : \\ &\quad (i_1.receivers.element \cup i_1.sender) \cap \\ &\quad (i_2.receivers.element \cup i_2.sender) = \emptyset \end{aligned} \tag{6}$$

In the use case, the TJunction controller interacts with the three traffic lights, **A**, **B**, and **C**. Listing 1 defines two interactions synchronizing TJunction controller and the traffic lights, using a textual Domain-Specific Language (DSL). The **synchronize** keyword specifies the InteractionType to be synchronous.

Each interaction first defines the *sender* state-changing element. Then *receivers* are defined by navigating along an arrow (BehavioralRelationship) to other BehavioralModels in the SRM and then specifying a StateChangingElement. All navigation along BehavioralRelationships starts from the BehavioralModel containing the *sender*, so constraint (5) is satisfied.

```
1 synchronize(TJunction.Switch_to_P1,
2              TJunction.A.turn_green,
3              TJunction.B.turn_red,
4              TJunction.C.turn_green)
5
6 synchronize(TJunction.Switch_to_P2,
7              TJunction.A.turn_red,
8              TJunction.B.turn_green,
9              TJunction.C.turn_red)
```

Listing 1 Interactions for the use case

We can explain the interactions as follows. The first interaction defines that the task Switch to P1 and three other state-changing elements synchronize. Furthermore, line 2 specifies that one of the synchronization receivers is the element *turn green* connected by the relationship *A*. Similarly, two other transitions of the traffic lights *B* and *C* are specified in the following two lines. Thus, the interaction defines a synchronization of a task and three traffic light transitions. The second interaction defines a synchronization for the task Switch to P2 and three traffic light transitions.

To summarize, we use the system relationship metamodel to define the relations between the behavioral models in a multi-model. Thus, the inter-relations between behavioral models in a multi-model are given by interactions and their behavioral relationships. Interactions are specific to each use case, but identifying state-changing elements must only be done once for each behavioral language.

3.4. Generate a global behavior specification

Using the SRM and snapshot metamodels, we can represent the global states of the system. However, we still need a formal specification of the global behavior to check the defined properties. The specification of the global behavior used in our approach must fulfill the following three requirements:

1. The specification must implement the semantics of each behavioral model.
2. The specification must realize the defined interactions between the behavioral models.
3. The specification semantics must allow the checking of behavioral properties for a given system configuration.

Thus, a specification in any formalism fulfilling these three requirements can be used in our approach. Consequently, one can experiment with different formalisms, for example, GTs, rewriting logic, state machines, Petri nets, or process algebras, without changing the general framework. One can then pick the most suitable formalism for the modeling scenario at hand regarding, for example, the performance of consistency checking. In section 4, we describe how we generate specifications for the GT toolset Groove.

To summarize, we generate a specification of the global system behavior. This generation takes the models and interactions as input and is fully automated to allow frequent model changes.

3.5. Check behavioral consistency

In this step, for a given system configuration, we use the generated specification of the global behavior to check the consistency. A system configuration is an instance of the SRM and is automatically translated into the formalism used in the specification. We then check the defined properties using the specification and the system configuration. This step is fully automated, such that it can be executed as many times as needed for different system configurations and properties while using the same specification.

Finally, if a consistency requirement is violated, a counterexample will be presented. We can only show counterexamples if the concrete tool, executing the generated specifications, provides them. However, most modern tools, including Groove and Maude, will provide a counterexample. Adopting the same concrete syntax to visualize the counterexample as for the atomic propositions should be ideal for helping user understanding. Uncovered inconsistencies can lead to a consistency restoration process, which is crucial but out of the scope of this paper. We describe consistency checking for the use case and its result at the end of the next section.

4. Specification of the global behavior

This section describes how GTs can be used as one possible formalism for behavioral consistency management. We utilize the Groove tool set to run the generated specifications, i.e., GT systems (Rensink 2004) for the use case and discuss the results. The successful implementation serves as a *proof of concept* for our approach. We have chosen to use the GT formalism to describe our approach because GTs provide a visual representation and allow for a clearer understanding. As an alternative to GT and Groove, we describe in (Kräuter 2023) our implementation using rewriting logic and Maude.

Both implementations utilize a *global* Higher-Order model Transformation (HOT) from the behavioral models and their interactions to GT rules (Groove) or term rewriting rules (Maude). Since the results of the Model Transformation (MT) can be regarded as MTs themselves, we say the MT is *higher-order* (Tisi et al. 2009). In the following, we describe the HOT to generate GT specifications for Groove. The HOT works similarly when generating a term-rewriting specification for Maude.

The HOT can be decomposed into two steps. The **first step** to generate a specification of the global behavior is to create GT rules for each behavioral model contained in the multi-model. Each set of rules must describe the behavior of the given behavioral model by manipulating instances of the snapshot metamodels. For example, a rule for a transition in a state machine changes the current state of a state machine snapshot from the source to the target of the transition.

Thus, we need *local* HOTs for each behavioral modeling language producing rules. Each local HOT only has to be implemented once by a language engineer. The HOT, metamodel, and snapshot metamodel for a behavioral language can be shared together, for example, as a plugin, such that they can be reused in any future setting the language is needed. In addition, each local HOT must keep traces of the generated rules. Concretely, it has to save which rules originated from which state-changing elements in the behavioral model. Returning to the state machine example, we must know which transition results in which rule. In general, multiple rules may be associated with one state-changing element of a behavioral model. For example, a receive task in a BPMN process is represented by two rules since it starts and then waits for an incoming message before finishing.

The **second step** is to modify the generated rules to reflect the defined interactions. Interactions define the synchronization of systems, which we encode by merging the individual rules into rules describing the global behavior. In the following section, we describe these two steps in detail using the use case as an example.

4.1. Groove specification

A GT system consists of a set of GT rules of the form $L \to R$, where the graph L is called the left-hand side and the graph R is called the right-hand side of the rule. Nodes/edges in R but not in L are added by a rule, while nodes/edges in L and R are preserved, and a rule deletes nodes/edges that are in L but not in R. Applying a GT system to a given graph, one obtains a state space where each state is a graph, and each transition is a rule application. A formal description of GT systems can, for example, be found in (Ehrig et al. 2006).

We generate typed GT systems, where the merge of the SRM and the snapshot metamodels is the type graph (Kräuter 2023). Individual rules and rules changing the global state conform to this type graph. Interactions result in global rules which change multiple parts of the global state. A global GT rule is calculated by taking the sum of all left-hand sides and right-hand sides of the individual GT rules (Baldan et al. 1999, Definition 3.2.7). Together with a system configuration, i.e., a start graph, we can obtain an executable formal specification of the global behavior. Consequently, this can be used to check behavioral consistency.

We will now explain how the GT rules are generated for the use case. To apply our approach to the use case, we need to define local HOTs from state machines and BPMN processes to GT rules.

4.1.1. State machine semantics

The local HOT to generate GT rules for finite state machines is straightforward. Each transition leads to a GT rule. For example, figure 14 shows the GT rules for the transitions turn green and turn red of the traffic light model. It uses the concrete syntax introduced in figure 10 to depict state machine snapshots and their current states. We depict a GT rule by showing the graph L on the left, R on the right, and a named white arrow from L to R.

Using a traffic light snapshot with the state red as a start graph, we generate the same state space in Groove as the traffic light state machine describes. The generated GT system and the other GT systems of the use case, including further instructions regarding execution and consistency checking, can be found in

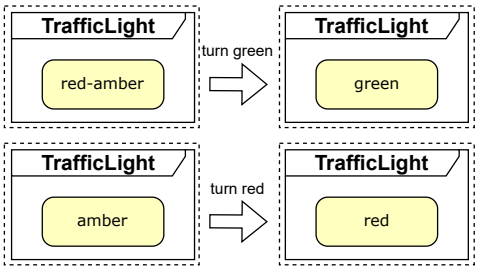


Figure 14 GT rules for *turn green* and *turn red*

(Kräuter 2023).

4.1.2. BPMN semantics The local HOT to generate GT rules for BPMN processes is challenging, and we are currently only supporting a subset of the BPMN semantics (see (Kräuter 2023)). Generally, we construct one or more rules for each flow node, i.e., type of state-changing element in a BPMN model. Furthermore, we created a comprehensive test suite to ensure the correctness of our HOT (Kräuter 2023). Figure 15 shows the GT rules for the task Switch to P1 of the TJunction controller. It uses the concrete syntax introduced in figure 10 to represent process snapshots containing tokens.

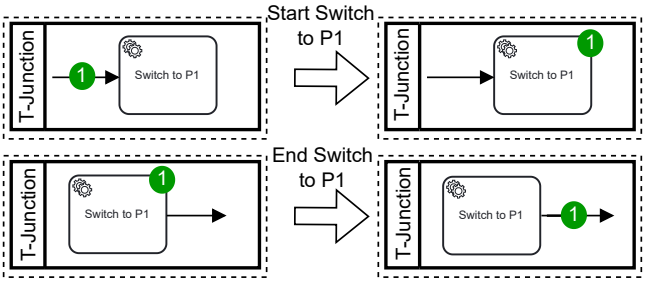


Figure 15 Example GT rules for the TJunction controller

Due to limited space, we only show the Switch to P1 rules but the artifacts of this paper (Kräuter 2023) contain the full GT system and a wiki explaining the HOT in detail. To summarize, we can generate GT systems that implement the behavioral semantics of BPMN.

4.1.3. Realizing interactions To realize interactions between the behavioral models we merge the previously generated rules as follows:

1. Rules generated from state-changing elements that are *not* part of interactions remain unchanged and are added to the global rule set.
2. For each interaction, we do the following:
   (a) Find the corresponding rule[2] P_0 for the sender of the interaction and find the rules $P_1, P_2, \ldots P_n$ for the receiver state-changing elements using the saved traces.
   (b) Create a *global rule* for the rules $P_0, P_1, \ldots, P_n$, which applies all of them at once, i.e., synchronizes the state changes of the behavioral models.
   (c) For each receiver of the interaction, instantiate the corresponding behavioral relationship from the behavioral model in P_0 to the behavioral model in P_i, for $1 \leq i \leq n$, and add it to the global rule. Thus, only behaviorally related models may interact, i.e., change their state simultaneously.

[2] If one state-changing element results in more than one rule, one can define a strategy to pick the appropriate rule.

The defined interactions change the rules for switching to phases 1 and 2. Figure 16 shows the resulting rule for switching to phase 1. We decided that the interactions between the traffic lights and the TJunction controller should synchronize with the end of the task, not the start. Thus, the rule was constructed using the individual rules *turn green* and *turn red* for traffic lights (see figure 14) and the rule *End Switch to P1* (see figure 15). Exactly these rules were used since they are generated from the state-changing elements specified in the first interaction (see listing 1).

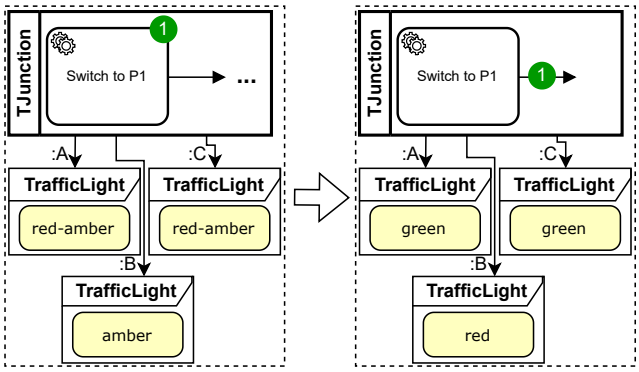


Figure 16 Global GT rule to switch to phase 1

The rule changes all traffic lights simultaneously and finishes the task. The corresponding individual rules no longer exist, so synchronization of the behavior is guaranteed. A similar rule exists for switching to phase 2, resulting from the other interaction. All other rules are left untouched while constructing the global GT system.

4.1.4. Check behavioral consistency Finally, we can generate the global state space of the system using the global GT system, which can be found in (Kräuter 2023). To check the requirements formalized by properties 1-4, we must also encode the used atomic propositions. These are specified as *graph conditions* in Groove. A graph condition in Groove is a rule that does not change elements but can be used as an atomic proposition in model checking.

4.2. Behavioral consistency in the use case

Running the obtained GT specification shows that properties (1) and (2) hold, while properties (3) and (4) do *not* hold (see artifacts in (Kräuter 2023)). The counterexamples for properties 3 and 4 show an unexpected race condition that must be handled: after the TJunction controller signals that the traffic light A

is green, bus B1 can advance to the Pass Junction activity. However, simultaneously, the TJunction controller can enter the subprocess for the next phase, which can be interrupted by the associated timer event. This can happen before bus B1 passes the junction, resulting in a violation of the consistency requirement.

The modelers have different options to handle the detected inconsistencies. One option is to keep the models unchanged and pay special attention to the found race condition during system implementation. This can be an acceptable solution since the Pass Junction activity is also modeled as a user activity, i.e., the bus driver decides when to cross the T-Junction. Furthermore, tolerating inconsistencies can be a viable option in MDE (Weidmann et al. 2021). Another option is to change the models to resolve the inconsistency. If buses are autonomous, this is the preferred option. For example, the T-Junction controller could wait for the bus to pass before changing the traffic lights again.

5. Discussion

In this section, we discuss two potential limitations of our approach: supporting new modeling languages and state space explosion.

5.1. Support for new modeling languages

To support a new modeling language in our approach one must do the following tasks: describe the state structure in a snapshot metamodel, identify state-changing elements in the language's metamodel, and implement a HOT to the chosen formalism. A plugin containing these three artifacts can then be reused.

Implementing a HOT that correctly implements a new modeling language's semantics takes time and effort. We learned that when implementing HOTs to the same underlying formalism, one gets accustomed to the formalism and naturally builds a framework to generate and test specifications in this formalism. Thus, the work needed to implement new languages decreases over time, but one must always understand the semantics of the new language to map them to the chosen formalism.

These tasks are typically done by a language engineer who may be a different person than the modeler who uses our consistency management approach. Generally, we assume two roles: language engineer and modeler. The modeler is in control of the use case and executes the manual tasks in Figure 7, while the language engineer is responsible for supporting the modeling languages which are used in the multi-model.

5.2. State space explosion

State space explosion is one of the predominant issues when applying model checking to complex systems, which are often found in real-world applications. We tested our approach using the generated GT system in Groove described earlier and the alternative implementation using rewriting logic with Maude. As one can see in table 1 and table 2, our approach is not immune to state space explosion. The tables show the states, transitions (rewrites), and average runtime of a full state space exploration in the Groove (Maude) specification. Four scenarios were benchmarked, starting with the use case multi-model without approaching buses and then increasing the number of buses up to three. Generally, the Maude exploration time is lower despite larger state spaces due to technical differences in implementing the BPMN semantics.

To calculate the average runtime, we used the hyperfine benchmarking tool (Peter 2022) (version 1.15.0), which ran state space exploration for each scenario ten times. Timing evaluations were done with an AMD Ryzen 7700X processor and 32 GB of RAM running Maude version 3.1 (inside the Windows Subsystem for Linux) and Groove version 5.8.1. A description of how to run the benchmarks is available in (Kräuter 2023).

Use case	States	Transitions	Exploration time
No buses	168	438	~1.201 ms
1 bus	2.888	10.046	~1.647 ms
2 buses	27.880	121.554	~4.087 ms
3 buses	195.336	1.028.340	~25.137 ms

Table 1 State space exploration in Groove

Use case	States	Rewrites	Exploration time
No buses	168	664	~28 ms
1 bus	3.304	19.958	~120 ms
2 buses	35.280	279.776	~1.279 ms
3 buses	260.176	2.522.582	~12.599 ms

Table 2 State space exploration in Maude

As in our use case, one is generally not interested in a full state space exploration as in table 1 and table 2 but rather in the validity of a set of behavioral properties. Checking a property specified in LTL does not necessarily lead to a full state space exploration. Furthermore, not every property is concerned with all behavioral models. For example, properties (1) and (2) of the use case do not involve buses and can be checked on smaller state spaces, not including approaching buses. In addition, checking a set of properties can be run in parallel. If some properties are computationally expensive, they can be run on dedicated hardware, for example, during a continuous integration pipeline once a day in case a behavioral model or an interaction changes. Thus, checking the consistency of a multi-model can be seen as an additional test during MDE, which can be run locally but, in addition, is a vital part of continuous integration.

The performance of the Maude LTL model checker is comparable to the popular SPIN model checker (S. Eker et al. 2004) and thus is proven to be competitive. However, one could use different techniques to mitigate the state space explosion problem further. One technique is to abstract models further such that they only contain information relevant to the set of properties to be checked. Thus, minimal models are synchronized,

leading to smaller state spaces. However, finding correct minimal models might not be trivial.

Partial-order reduction is a well-known and effective technique to mitigate the state space explosion problem. It is currently not implemented in the Maude LTL model-checker, but there is promising work to integrate partial-order reduction into the model-checker (Farzan & Meseguer 2007), showing substantial state space reductions. The potential to reduce the state space using this technique is huge, especially for model-checking concurrent systems (Clarke et al. 2018). Our approach analyzes concurrent systems with some interaction, and thus model-checking would greatly benefit from partial-order reduction. In our opinion, partial-order reduction must be implemented to analyze models from real-world applications.

6. Related work

We organized related work into two sections. First, we discuss ad-hoc and general solutions related to the problem of behavioral consistency. Then, we relate our work to the fields of Multi-Paradigm Modeling (MPM) and co-simulation.

6.1. Ad-hoc and general solutions

The general idea of transforming different behavioral formalisms to a single formalism to reason about cross-cutting concerns is not new, see, e.g., (Engels et al. 2001). For example, (Küster & Stehr 2003) developed consistency checking for sequence diagrams and statecharts based on CSP, while (Yao & Shatz 2006) used Petri nets for the same scenario. Nevertheless, these approaches only resemble ad-hoc solutions to specific combinations of two languages. They do not consider the general problem of behavioral consistency in a heterogeneous modeling scenario.

(Kienzle et al. 2019) proposes a unifying framework for the homogeneous model composition of structural and behavioral models. To combine behavioral models, they use Event structures as an underlying formalism and show how different homogeneous behavioral models can be combined. In addition, to express behavioral relationships between different models they create causal relationships, which are used during model composition. Generally, their approach is compatible with ours since we do not mandate a specific formalism. Thus, Event structures could be considered, where interactions are realized using causal relationships. Since their research challenges and resulting work items align with our work, we see our work following the same line of research. In addition, they cover the same class of behavioral formalisms as our approach (see DTDS and DEVS in the next subsection).

(Vara Larsen et al. 2015) propose the coordination framework B-COoL. In B-COoL a modeler defines behavioral interfaces for each behavioral language, which are then used to specify interactions between models. To execute the models with the specified interactions, they transform them into Clock Constraint Specification Language (CCSL) models. Their work results in plugins for GEMOC studio, which support running and debugging the models.

Both approaches (Kienzle et al. 2019; Vara Larsen et al. 2015) are similar to our approach since both have mechanisms to define interactions between behavioral models and generate a global behavioral specification based on these interactions. Furthermore, the generation of the global behavior is achieved by transforming models to a certain formalism like the HOT in our approach. However, the generated specification does not allow to check global behavioral properties since they do not make the state structure (snapshot metamodels in our approach) of the participating models explicit. In our approach, we define the two fundamental concepts (*state* and *state-changing elements*) to check behavioral consistency when heterogeneous models interact.

6.2. Multi-paradigm modeling and co-simulation

MPM is based on the idea to model every aspect and part of a system using the most appropriate modeling formalism(s) (Amrani et al. 2021). Thus, MPM often leads to multi-modeling scenarios where models conforming to different modeling formalisms are used.

Different behavioral formalisms can be classified using two criteria: *time* and *values of state variables* (Wainer 2009). Figure 17 shows the classification where time and state variables can either be *discrete* or *continuous*.

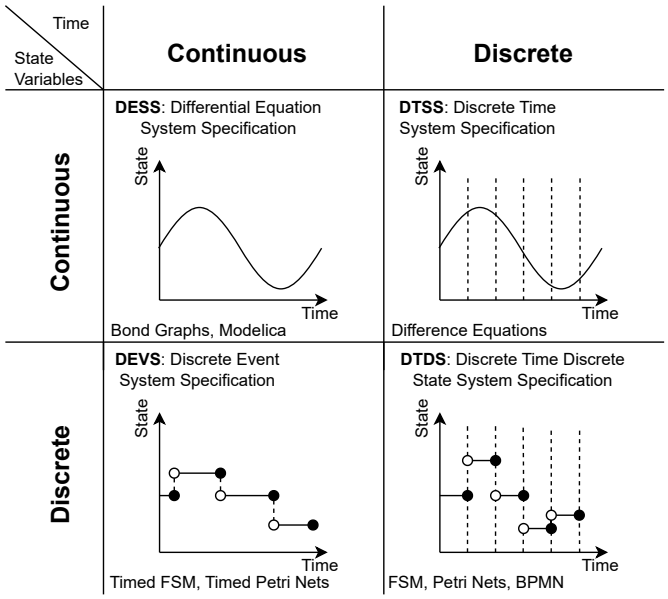


Figure 17 Classification of behavioral formalisms according to the nature of the *time* and *state variables* (adopted from (Wainer 2009; Amrani et al. 2021))

So far we have seen Discrete Time Discrete State System Specifications (DTDSs) like Finite State Machine (FSM) and BPMN in this work, where time and state variables are discrete. Furthermore, our approach can potentially be applied to Discrete Event System Specifications (DEVSs) since the underlying formalisms support non-discrete time but verification capabilities for DEVSs are more limited. However, we do not support formalisms with continuous state variables (top part of figure 17), since our central concept of *state-changing elements* would need to be changed. We think that supporting DEVS

formalisms is a good trade-off between the usefulness and complexity of our approach, since DEVS covers a wide variety of formalisms (Vangheluwe et al. 2002).

Using different modeling formalisms as is the case when following the MPM paradigm makes system simulation challenging. Co-Simulation aims to solve this challenge by composing the simulation of a system's parts into a global simulation (Gomes et al. 2019).

For example, (J. Eker et al. 2003) propose an actor-oriented co-simulation approach, where each system part is represented as an actor. An actor can communicate through its interfaces with other actors. Their approach is implemented in the tool *Ptolemy* and supports continuous time and state variables. Furthermore, the FMI[3] is a co-simulation standard to exchange executable systems parts, so-called Functional Mock-up Units (FMUs). Each FMU, similar to the actors in Ptolemy, comes with an XML model to describe its interface, for example, the FMU's exposed variables. FMUs support continuous time and state variables and are widely used in the industry.

However, with co-simulations, one can only simulate systems not check global behavioral properties.

7. Conclusion and future work

Our work represents a formalization of behavioral consistency management in a heterogeneous modeling scenario, facilitating the formulation and checking of *global* properties. Previous work either only dealt with the behavioral consistency between specific pairs of models or focused on the simulation in a heterogeneous scenario but lacked checking global properties.

Our approach is based on two fundamental concepts: *state* and *state-changing elements*. The state structure of each participating behavioral language must be explicitly defined to infer how global states are structured. Furthermore, state-changing elements serve as a minimal behavioral interface to uniformly define interactions for heterogeneous models. These two fundamental concepts can be found in most, if not every DEVS formalism. Thus, our approach can support behavioral formalisms with discrete state variables.

In future work, we plan to extend our implementation to support more behavioral modeling languages such as activity diagrams, hierarchical state machines, and the π-calculus. In addition, we aim to apply the approach to real-life industrial case studies. Furthermore, two systems often exchange data while interacting, for example, using name-passing or messaging. The exchanged data then greatly influences the future behavior of the systems. Thus, adding data transfers to interactions between heterogeneous models is an important issue left for future work. Finally, if consistency violations are found, consistency restoration must be achieved. We leave consistency restoration of behavioral models as a problem for future work.

Acknowledgments

We want to thank the anonymous reviewers for their valuable comments and helpful suggestions.

[3] https://fmi-standard.org/

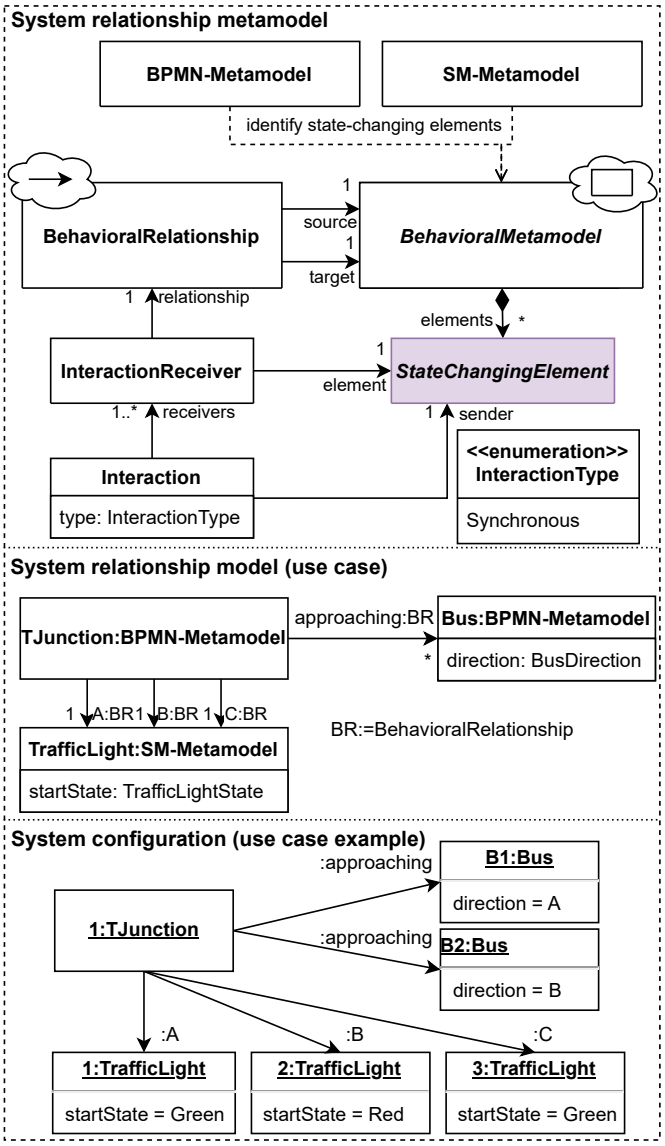


Figure 18 Overview of all concepts and their usage

References

Amrani, M., Blouin, D., Heinrich, R., Rensink, A., Vangheluwe, H., & Wortmann, A. (2021, June). Multi-paradigm modelling for cyber–physical systems: A descriptive framework. *Software and Systems Modeling*, *20*(3), 611–639. doi: 10.1007/s10270-021-00876-z

Baldan, P., Corradini, A., Montanari, U., Rossi, F., Ehrig, H., & Löwe, M. (1999, August). Concurrent semantics of algebraic graph transformations. In *Handbook of Graph Grammars and Computing by Graph Transformation* (Vol. 3, pp. 107–188). World Scientific. doi: 10.1142/9789812814951_0003

Boronat, A., Knapp, A., Meseguer, J., & Wirsing, M. (2009). What Is a Multi-modeling Language? In A. Corradini & U. Montanari (Eds.), *Recent Trends in Algebraic Development Techniques* (Vol. 5486, pp. 71–87). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-03429

-9_6

Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-driven software engineering in practice* (Second ed.) (No. 4). San Rafael, Calif.: Morgan & Claypool Publishers.

Cicchetti, A., Ciccozzi, F., & Pierantonio, A. (2019, December). Multi-view approaches for software and system modelling: A systematic literature review. *Software and Systems Modeling*, *18*(6), 3207–3233. doi: 10.1007/s10270-018-00713-w

Clarke, E. M., Henzinger, T. A., Veith, H., & Bloem, R. (Eds.). (2018). *Handbook of Model Checking*. Cham: Springer International Publishing. doi: 10.1007/978-3-319-10575-8

Ehrig, H., Ehrig, K., Prange, U., & Taentzer, G. (2006). *Fundamentals of algebraic graph transformation (monographs in theoretical computer science. An EATCS series)*. Berlin, Heidelberg: Springer-Verlag.

Eker, J., Janneck, J., Lee, E., Jie Liu, Xiaojun Liu, Ludvig, J., … Yuhong Xiong (2003, January). Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, *91*(1), 127–144. doi: 10.1109/JPROC.2002.805829

Eker, S., Meseguer, J., & Sridharanarayanan, A. (2004, April). The Maude LTL Model Checker. *Electronic Notes in Theoretical Computer Science*, *71*, 162–187. doi: 10.1016/S1571-0661(05)82534-4

Engels, G., Küster, J. M., Heckel, R., & Groenewegen, L. (2001, September). A methodology for specifying and analyzing consistency of object-oriented behavioral models. *ACM SIGSOFT Software Engineering Notes*, *26*(5), 186–195. doi: 10.1145/503271.503235

Farzan, A., & Meseguer, J. (2007, July). Partial Order Reduction for Rewriting Semantics of Programming Languages. *Electronic Notes in Theoretical Computer Science*, *176*(4), 61–78. doi: 10.1016/j.entcs.2007.06.008

France, R., & Rumpe, B. (2007, May). Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering (FOSE '07)* (pp. 37–54). Minneapolis, MN, USA: IEEE. doi: 10.1109/FOSE.2007.14

Gomes, C., Thule, C., Broman, D., Larsen, P. G., & Vangheluwe, H. (2019, May). Co-Simulation: A Survey. *ACM Computing Surveys*, *51*(3), 1–33. doi: 10.1145/3179993

Kienzle, J., Mussbacher, G., Combemale, B., & Deantoni, J. (2019, October). A unifying framework for homogeneous model composition. *Software & Systems Modeling*, *18*(5), 3005–3023. doi: 10.1007/s10270-018-00707-8

Klare, H., & Gleitze, J. (2019, September). Commonalities for Preserving Consistency of Multiple Models. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)* (pp. 371–378). Munich, Germany: IEEE. doi: 10.1109/MODELS-C.2019.00058

Kräuter, T. (2021). Towards behavioral consistency in heterogeneous modeling scenarios. In *2021 ACM/IEEE international conference on model driven engineering languages and systems companion (MODELS-C)*. doi: 10.1109/MODELS-C53483.2021.00107

Kräuter, T. (2023, February). *Artifacts ECMFA-2023*. https://github.com/timKraeuter/ECMFA-2023.

Küster, J., & Stehr, J. (2003). Towards explicit behavioral consistency concepts in the UML. In *Proceedings of 2nd ICSE workshop on scenarios and state machines: Models, algorithms, and tools (portland, USA)*.

Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., & Wimmer, M. (2014). ProMoBox: A Framework for Generating Domain-Specific Property Languages. In B. Combemale, D. J. Pearce, O. Barais, & J. J. Vinju (Eds.), *Software Language Engineering* (Vol. 8706, pp. 1–20). Cham: Springer International Publishing. doi: 10.1007/978-3-319-11245-9_1

Object Management Group. (2013, December). *Business Process Model and Notation (BPMN), Version 2.0.2*. https://www.omg.org/spec/BPMN/.

Peter, D. (2022). *Hyperfine*. GitHub.

Rensink, A. (2004). The GROOVE Simulator: A Tool for State Space Generation. In T. Kanade et al. (Eds.), *Applications of Graph Transformations with Industrial Relevance* (Vol. 3062, pp. 479–485). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-540-25959-6_40

Spanoudakis, G., & Zisman, A. (2001, December). Inconsistency Management in Software Engineering: Survey and Open Research Issues. In *Handbook of Software Engineering and Knowledge Engineering* (pp. 329–380). World Scientific. doi: 10.1142/9789812389718_0015

Stünkel, P., König, H., Lamo, Y., & Rutle, A. (2021, July). Comprehensive Systems: A formal foundation for Multi-Model Consistency Management. *Formal Aspects of Computing*. doi: 10.1007/s00165-021-00555-2

Tisi, M., Jouault, F., Fraternali, P., Ceri, S., & Bézivin, J. (2009). On the Use of Higher-Order Model Transformations. In R. F. Paige, A. Hartman, & A. Rensink (Eds.), *Model Driven Architecture - Foundations and Applications* (Vol. 5562, pp. 18–33). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-02674-4_3

Vangheluwe, H., de Lara, J., & Mosterman, P. (2002). An introduction to multi-paradigm modelling and simulation..

Vara Larsen, M. E., DeAntoni, J., Combemale, B., & Mallet, F. (2015, September). A Behavioral Coordination Operator Language (BCOoL). In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 186–195). Ottawa, ON, Canada: IEEE. doi: 10.1109/MODELS.2015.7338249

Wainer, G. A. (2009). *Discrete-event modeling and simulation: A practitioner's approach*. Boca Raton: CRC Press.

Weidmann, N., Kannan, S., & Anjorin, A. (2021, June). Tolerance in Model-Driven Engineering: A Systematic Literature Review with Model-Driven Tool Support. *arXiv:2106.01063 [cs]*.

Yao, S., & Shatz, S. (2006, November). Consistency Checking of UML Dynamic Models Based on Petri Net Techniques. In *2006 15th International Conference on Computing* (pp. 289–297). Mexico city, Mexico: IEEE. doi: 10.1109/CIC.2006.32

About the authors

Tim Kräuter is a Ph.D. student at the Western Norway University of Applied Sciences, Bergen, Norway. His primary research is on integrating heterogeneous behavioral models in multi-model-driven software engineering. Previously he worked as a software developer and acquired a master of science in Information Engineering at the University of Applied Sciences, FHDW Hannover, Germany. You can contact the author at tkra@hvl.no or visit https://timkraeuter.com/.

Harald König is a professor for Computer Science at the University of Applied Sciences, FHDW Hannover, Germany, and an Adjunct Professor at the Department of Computer Science, Electrical Engineering and Mathematical Sciences at the Western Norway University of Applied Sciences, Bergen, Norway. Before entering academia, he worked at SAP in Walldorf and received his Ph.D. in pure Mathematics from Leibniz University in Hannover, Germany. You can contact the author at harald.koenig@fhdw.de.

Adrian Rutle is a professor at the Department of Computer science, Electrical engineering and Mathematical sciences at the Western Norway University of Applied Sciences, Bergen, Norway. Rutle's main interest is applying theoretical results from the field of model-driven software engineering to practical domains and his main expertise is in the development of formal modeling frameworks for domain-specific modeling languages, graph-based logic for reasoning about static and dynamic properties of models, and the use of model transformations for the definition of the semantics of modeling languages. His recent research focuses on multilevel modeling, model repair, multi-model consistency management, modeling and simulation for robotics, digital fabrication, smart systems, and applications of machine learning in model-driven software engineering. You can contact the author at aru@hvl.no.

Yngve Lamo is a professor at the Department of Computer science, Electrical engineering and Mathematical sciences at the Western Norway University of Applied Sciences, Bergen. Lamo holds a Ph.D. in Computer Science from the University of Bergen, Norway. His research interests span from formal foundations of Model-Based Software Engineering to its applications, especially in Health Informatics. He is currently applying MDSE to create Adaptive Software Technologies for mental Health. You can contact the author at yla@hvl.no.

Patrick Stünkel is currently working as a postdoctoral researcher at Haukeland University Hospital in Bergen, Norway, where he is working on applying process mining, workflow modeling, and optimization techniques in digital pathology. Before that, Patrick did his Ph.D. at Western Norway University of Applied Sciences on the topics of semantic interoperability and consistency management among heterogeneous software models. You can contact the author at Patrick.Stunkel@helse-bergen.no or visit https://past.corrlang.io/.

PAPER C

A HIGHER-ORDER TRANSFORMATION APPROACH TO THE FORMALIZATION AND ANALYSIS OF BPMN USING GRAPH TRANSFORMATION SYSTEMS

Tim Kräuter, Adrian Rutle, Harald König, Yngve Lamo

Logical Methods in Computer Science, Vol. 20, Issue 4, 2024, https://doi.org/10.46298/lmcs-20(4:4)2024

Logical Methods in Computer Science
Volume 20, Issue 4, 2024, pp. 4:1–4:31
https://lmcs.episciences.org/

Submitted Nov. 10, 2023
Published Oct. 07, 2024

A HIGHER-ORDER TRANSFORMATION APPROACH TO THE FORMALIZATION AND ANALYSIS OF BPMN USING GRAPH TRANSFORMATION SYSTEMS *

TIM KRÄUTER [a], ADRIAN RUTLE [a], HARALD KÖNIG [b,a], AND YNGVE LAMO [a]

[a] Western Norway University of Applied Sciences, Bergen, Norway
e-mail address: tkra@hvl.no, aru@hvl.no, hkoe@hvl.no, yla@hvl.no

[b] University of Applied Sciences, FHDW, Hanover, Germany
e-mail address: harald.koenig@fhdw.de

Abstract. The Business Process Modeling Notation (BPMN) is a widely used standard notation for defining intra- and inter-organizational workflows. However, the informal description of the BPMN execution semantics leads to different interpretations of BPMN elements and difficulties in checking behavioral properties. In this article, we propose a formalization of the execution semantics of BPMN that, compared to existing approaches, covers more BPMN elements while also facilitating property checking. Our approach is based on a higher-order transformation from BPMN models to graph transformation systems. To show the capabilities of our approach, we implemented it as an open-source web-based tool.

1. Introduction

In today's fast-paced business environment, organizations with complex workflows require powerful means to accurately map, analyze, and optimize their processes. Business Process Modeling Notation (BPMN) [Obj13] is a widely used standard to define these workflows. However, the informal description of the BPMN execution semantics leads to different interpretations of BPMN models and difficulties in checking behavioral properties [CFP+21]. Various studies have shown that business process models suffer from control-flow errors [Men09]. Formalizing BPMN can drastically reduce the cost of business process automation by facilitating the detection of errors and optimization potentials in process models already during design time. For example, general behavioral properties such as *Safeness* and *Soundness* were adapted to BPMN in [CMRT18]. They can uncover control-flow errors in BPMN models leading to deadlocks, dead activities, or other undesirable execution states. To this end, we propose a formalization that covers nearly all of the BPMN elements used in practice and supports checking behavioral properties to uncover control-flow errors quickly.

Key words and phrases: BPMN, Higher-order model transformation, Graph transformation, Model checking, Formalization.

* This article is an extended version of [KRKL23].

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.46298/LMCS-20(4:4)2024

[FFK+11] identifies *coverage*, *immediacy*, and *consumability* as the main challenges for applying formal analysis to BPMN models in practice. In this paper, we focus on *coverage* while we touch upon immediacy (analysis runtime) and consumability (tool usability) when describing the implementation of our approach in section 4. Many formalizations of BPMN already exist, for example, based on Petri Nets [DDO08], first-order logic [HBPQ19, HBP+22], and graph transformation [VGD13]. However, they either do not cover commonly used elements in BPMN [DDO08, HBPQ19, HBP+22] or do not facilitate property checking [VGD13] (see section 5). Thus, we developed a new formalization that **(1)** covers all commonly used BPMN elements and **(2)** supports property checking to uncover control-flow errors. We can achieve both requirements using advanced graph transformation theory concepts. Implementing the same coverage, for example, using Petri Nets, is not straightforward.

In this article, we consider two fundamental concepts when formalizing the execution semantics of BPMN. First, *state structure*, i.e., how model instances are represented during execution. The state structure corresponds to the type graph in Graph Transformation (GT) systems. Second, *state-changing elements*, i.e., which elements in a model encode state changes. These elements are implemented using GT rules, which we automatically generate based on a Higher-Order model Transformation (HOT) [TJF+09] for each specific BPMN model, as shown in Figure 1. Our HOT defines a formal execution semantics of BPMN, like approaches that formalize BPMN by mapping to Petri Nets or other formalisms [DDO08].

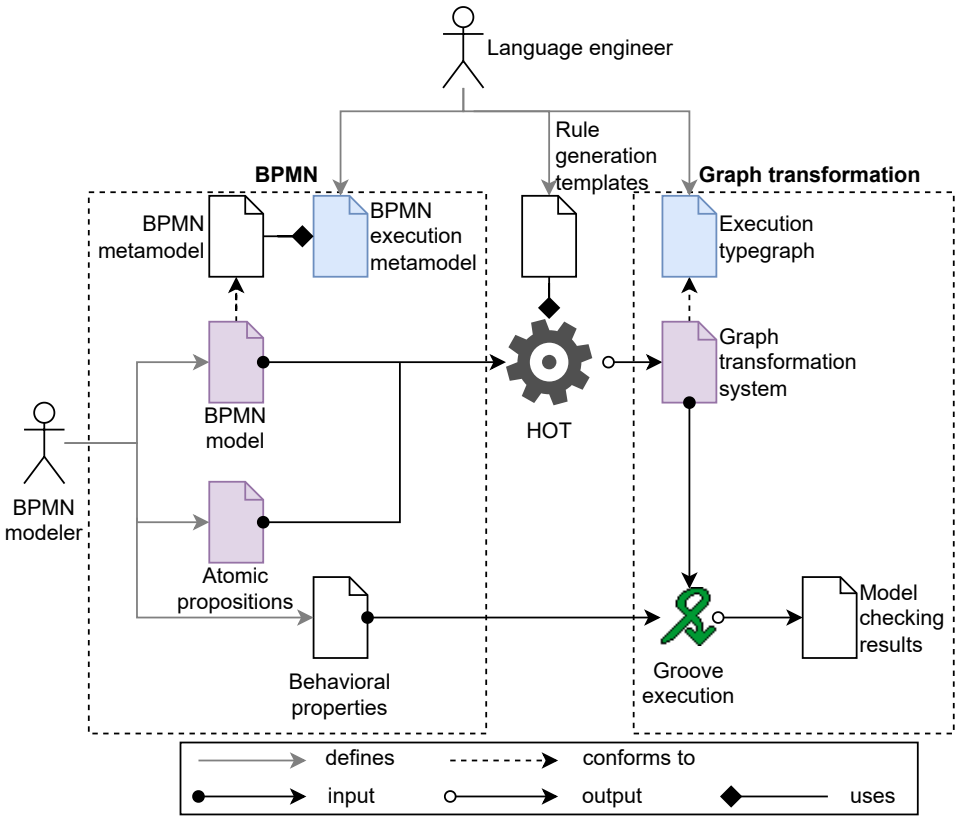


Figure 1: Overview of the approach

To begin the BPMN modeling process, a modeler first defines the BPMN model. The BPMN model may be checked against a predefined list of general behavioral properties, such as safeness and soundness. Furthermore, the modeler may also define custom behavioral properties specifically defined for the BPMN model. Custom properties require atomic propositions to describe desirable or undesirable states during execution, which the modeler can define using our concrete syntax based on the BPMN syntax. The defined BPMN model must adhere to the BPMN metamodel as outlined in the BPMN specification by the Object Management Group [Obj13]. The BPMN execution metamodel is defined by language engineers, utilizing the BPMN metamodel as a foundation to create the state structure for executing BPMN models.

We define a HOT from BPMN models and atomic propositions to GT systems (see purple-colored elements in Figure 1). We call the transformation *higher-order* since the resulting graph-transformation systems represent model-transformations themselves [TJF+09]. The HOT creates a GT system, i.e., GT rules and a start graph for a given BPMN model. It is defined using rule generation templates, which describe how GT rules should be generated for each state-changing element in BPMN (see section 2). The obtained GT system conforms to the execution type graph, which corresponds to the BPMN execution metamodel. In the figure, we have used the same color for artifacts that correspond to each other. Ultimately, we use Groove to execute the GT system and check the behavioral properties defined earlier. To facilitate model checking of custom behavioral properties, we create specific GT rules for the corresponding atomic propositions during the HOT.

The overview in Figure 1 is divided into two separated parts, denoted by dashed rectangles, to indicate the versatility of the approach as it can be applied to formalize other behavioral languages, such as activity diagrams and state charts [SSHK15, Obj17]. This formalization will require the language engineer to establish a new execution metamodel and a HOT for the new language. One could even change the *target* of the HOT from GT to a different formalism (term rewriting, Petri Nets, process algebras) if this makes sense for a given behavioral language [KKR+23].

This article consists of two main contributions. First, we introduce a new approach utilizing a HOT to generate GT rules — instead of providing fixed model-independent GT rules — to formalize the semantics of a behavioral language. Second, we apply our approach to BPMN, resulting in a formalization covering most BPMN elements that supports behavioral property checking. Furthermore, our formalization is implemented as a user-friendly, open-source web-based tool, the *BPMN Analyzer*, which can be used online without needing installation [Krä23].

Our contributions are practical, not theoretical. We build upon the comprehensive theory and tools available in the GT research field. Concretely, we utilize the single-pushout (SPO) approach with negative application conditions (NAC) [EHK+97], as implemented in Groove [Ren04]. In addition, we utilize *nested rules* with quantification to make parts of a rule repeatedly applicable or optional [Ren06, Ren17]. Moreover, we utilize the NACs to implement more intricate parts in the BPMN execution semantics, such as the termination of processes. Formal definitions of SPO rules, their application, and the corresponding extensions of the theory (NACs, nested rules) are well-known, see [EHK+97, Ren06]. We do not repeat them and instead focus on our practical contribution.

This article extends [KRKL23] as follows. (i) We explain many more BPMN elements, which are covered by our approach (see elements highlighted in blue in Figure 3). (ii) We enhance the explanation of the custom properties in section 3 by using an order handling

process to illustrate use cases for these properties. (iii) We detail the extensively improved BPMN analyzer tool in section 4 in which modelers can use our new atomic proposition editor. (iv) We test the scalability of our approach with 300 synthetically generated BPMN models of increasing size in section 4.

Outline The remainder of this article is structured as follows. First, we describe the BPMN semantics formalization using the HOT (section 2) before explaining how this can be utilized for model checking general BPMN and custom properties (section 3). Then, we detail the BPMN Analyzer, which implements our approach in section 4 and describe how we tested its performance and scalability. Finally, we discuss related work regarding BPMN element coverage in section 5 and conclude in section 6.

2. BPMN syntax & BPMN semantics formalization

Figure 2 depicts the structure of BPMN models with the corresponding concrete syntax BPMN symbols contained in clouds. A BPMN model is represented by a Collaboration that has participant Processes and MessageFlows between InteractionNodes. Each participant is a Process containing FlowElements. A FlowElement is either a FlowNode or SequenceFlow. A FlowNode is either an Activity, a Gateway, or an Event and can be connected to other FlodNodes using SequenceFlows. Many types of activities, gateways, and events exist, such as call activities, parallel gateways, and start events. Activities represent certain tasks to be carried out during a process, while events may happen during the execution of these tasks. Furthermore, gateways model conditions, parallelizations, and synchronizations [FR19].

Figure 2 is a simplified excerpt of the BPMN metamodel described in the BPMN specification [Obj13]. Each class depicted in Figure 2 maintains consistent naming with the BPMN metamodel, and all associations are directly found in [Obj13] or simplified, i.e., given by compositions of multiple existing associations.

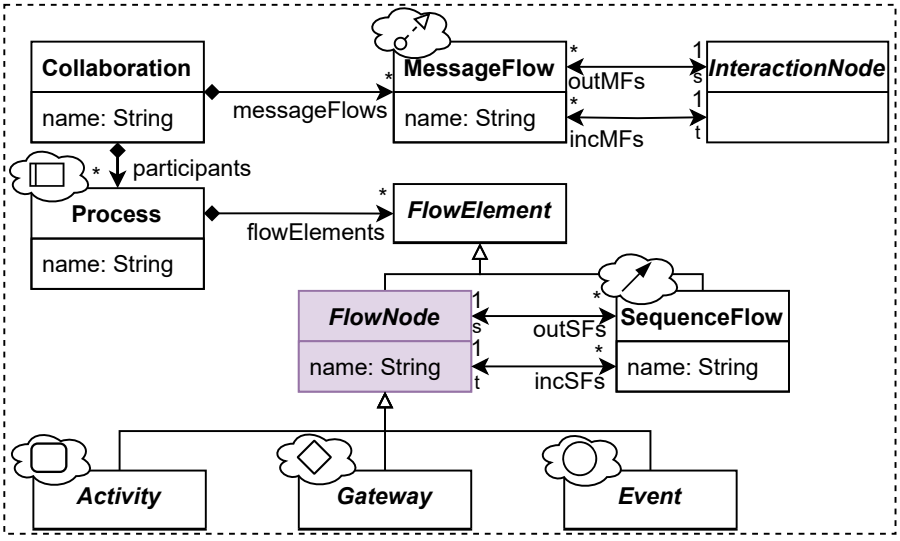


Figure 2: Simplified excerpt of the BPMN metamodel [Obj13]

Our approach supports all the BPMN elements depicted in Figure 3. These BPMN elements are divided into Events, Gateways, Activities, and Edges. Events and Activities are further divided into subgroups. An extensive overview of BPMN and its elements can be found in [FR19]. Although all these elements have been implemented and tested (see [Krä23]), we only explain the realization of the elements marked with a blue background due to space limitations. In the following, first, we define the BPMN execution metamodel to represent the BPMN state structure, and then we explain our formalization of the elements in Figure 3.

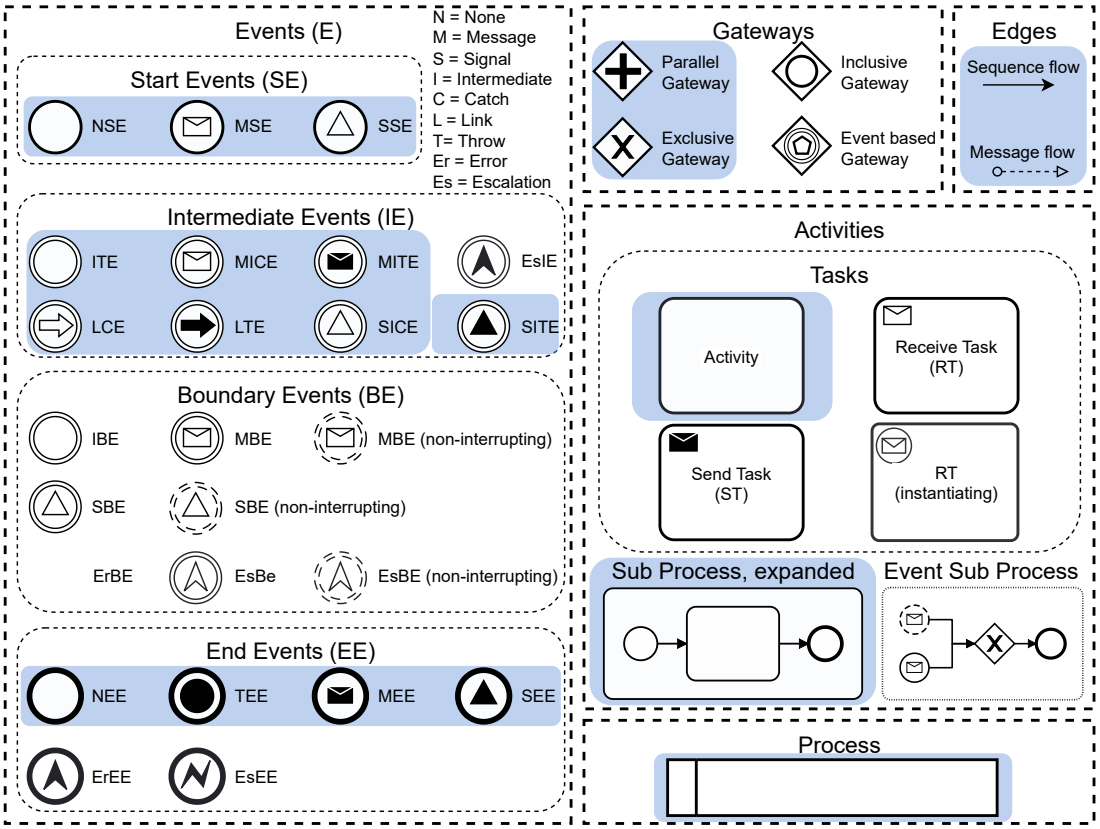


Figure 3: Overview of the supported BPMN elements (structure adapted from [HBP+22])

2.1. **BPMN execution metamodel.** The BPMN execution semantics is described using the concept of *tokens* [Obj13, FR19], which can be located at sequence flows and specific flow nodes. Tokens are consumed and created by flow nodes according to the flow node's type and the connected sequence flows. The FlowNode is colored purple in Figure 2 since it represents the *state-changing elements* of BPMN, as described in section 2. In our formalization of BPMN, we follow this token-based representation of the execution semantics.

To describe processes holding tokens during execution, we define the execution metamodel shown in Figure 4, depicted as a UML class diagram. The first task of a language engineer in our approach is to define the execution metamodel (see Figure 1).

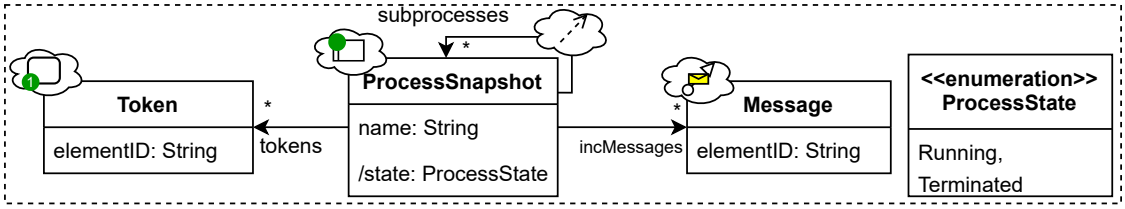


Figure 4: BPMN execution metamodel

The BPMN execution metamodel was not created by extending the BPMN metamodel and adding missing concepts such as tokens and messages. We created a minimal execution model that only contains concepts needed during execution. This is only possible since the HOT generates our rules for each specific BPMN model such that the structure of each model is already implicitly encoded in the rules. This design choice leads to smaller states in the GT system compared to an execution metamodel that extends the BPMN metamodel.

In Figure 4, we use ProcessSnapshot to denote a running BPMN process with a specific token distribution that describes one state in the history of the process execution. Every ProcessSnapshot has a set of tokens, incoming messages, and subprocesses. A ProcessSnapshot has the state Terminated if it has no tokens or subprocesses. Otherwise, it has the state Running. A Token has an elementID, which points to the BPMN Activity or the SequenceFlow at which it is located. A Message has an elementID pointing to a MessageFlow. To concisely depict graphs conforming to this type graph, we introduce a concrete syntax in the clouds attached to the elements. Our concrete syntax extends the BPMN syntax by adding process snapshots, subprocess relations, tokens, and messages. Tokens are represented as colored circles drawn at their specified positions in a model. In addition, we use colored circles at the top left of the bounding box, representing instances of the BPMN Process; these circles represent process snapshots. The token's color must match the color of the process snapshot holding the token. The concrete syntax was inspired by the bpmn-js-token-simulation [Cam23c].

The execution metamodel in Figure 4 is a UML class diagram without operations, which can be seen as an attributed type graph [HT20]. We keep the execution metamodel and the execution type graph separate (see Figure 1) because the execution metamodel should be independent of the formalism used to define the execution semantics. One can reuse the execution metamodel when changing the formalism or concrete tool implementing the formalism (in our case, Groove) by adjusting how the execution metamodel is transformed. Using the execution metamodel as the type graph, we can now define how the start graph and GT rules for the different BPMN elements are created.

Since our approach is based on a HOT from BPMN to GT systems, we generate a *start graph* and *GT rules* for each given BPMN model (see Figure 1). Generating the start graph for a BPMN model is straightforward. First, for each process in the BPMN model, we generate a process snapshot if the process contains a *None Start Event* (NSE). An NSE describes a start event without a trigger (none). Then, for each NSE, we add one token to each outgoing sequence flow. An example of a start graph is shown in Figure 5 using abstract and concrete syntax.

The HOT generates one or more GT rules for each FlowNode, i.e., state-changing element in a BPMN model. To better understand the transformation process, we will begin by

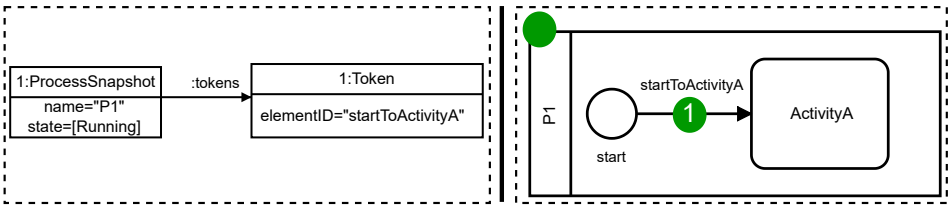


Figure 5: Example start graph in abstract (left) and concrete syntax (right)

presenting example results, namely the generated rules for an activity. Following this, we will explain how our HOT creates these rules and rules for the other elements in Figure 3.

Figure 6 depicts an example GT rule ($L \rightarrow R$) to start an activity in abstract syntax. The rule is straightforward: it moves a token from the incoming sequence flow to the activity.

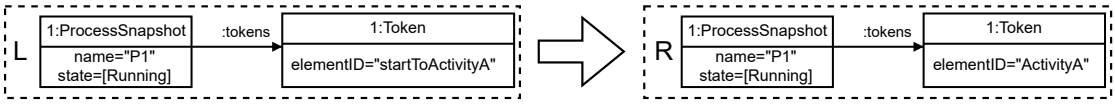


Figure 6: Example GT rule to start an activity (abstract syntax)

For the rest of the article, we will depict all rules in the concrete syntax introduced earlier. The rule from Figure 6 depicted in concrete syntax is shown on the top in Figure 7. The rule on the bottom in Figure 7 implements the termination of an activity, which will move one token from the activity to the outgoing sequence flow.

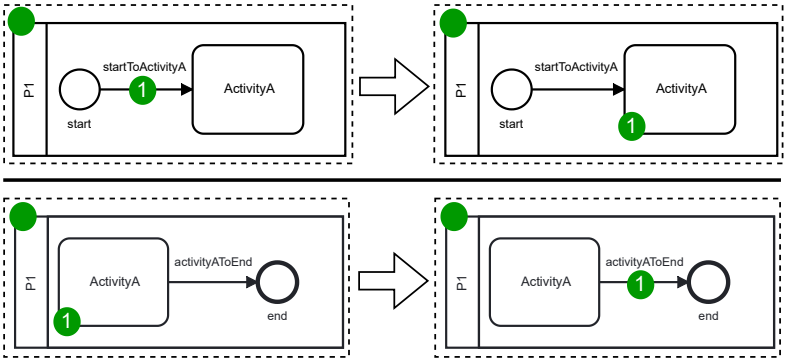


Figure 7: Example GT rules to start (top) and terminate (bottom) an activity

In the following subsections, we use our concrete syntax to define how our HOT generates these rules and rules for other flow nodes. Elements of the HOT are depicted using rule generation templates that show how specific rules are created for various flow nodes. Defining the rule generation templates and, thus, the HOT from BPMN to GT systems is the second task of the language engineer in our approach (see Figure 1).

2.2. **Process instantiation and termination.** Start events do not need GT rules since the generated start graph of the GT system will contain a token for each outgoing sequence flow of an NSE. Other types of start events are triggered in corresponding throw event rules.

Figure 8 depicts the rule generation template for *None End Events* (NEEs in Figure 3). All rule generation templates show a state-changing element (FlowNode) with surrounding flows in the left column and the applicable rule generation in the right column. The left column shows instances of the BPMN metamodel (Figure 2), and the right column shows the generated rules typed by the BPMN execution metamodel (see Figure 4). If more than one rule is generated from a FlowNode, an expression defines how each rule is generated. For example, the expression ∀sf ∈ E.incSFs for the rule generation template of end events (see Figure 8) generates one rule for each incoming sequence flow *sf* of the end event *E*. We use "." in expressions to navigate along the associations of the BPMN metamodel shown in Figure 2. In the example, E.incSFs means following all incSFs links for a FlowNode object, resulting in a set of SequenceFlow objects.

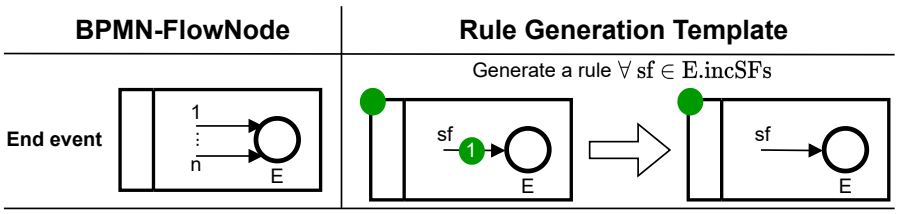


Figure 8: Rule generation template for *None End Events*

For example, if a BPMN model contains one NEE with two incoming sequence flows, the HOT will generate two GT rules as defined in the rule generation template in Figure 8. Thus, each rule generation template defines part of the HOT, and all rule generation templates together represent the HOT, which can be applied to a given BPMN model to generate concrete rules as specified in each template.

The generated end event rules delete tokens individually for each incoming sequence flow. However, they do not terminate processes. Process termination is implemented with a generic rule—independent of the input BPMN model—which applies to all process snapshots. The termination rule in Figure 9 is automatically generated once during the HOT. The rule changes the state of the process snapshot from running to terminated if it has neither tokens nor subprocesses. We use Groove Syntax instead of our concrete syntax since it is a special, more complex rule, and we do not want the concrete syntax to become too complex.

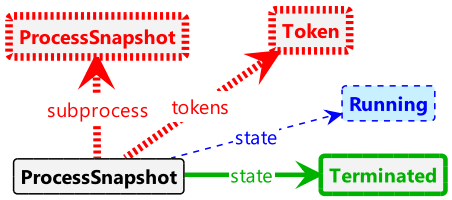


Figure 9: Termination rule in Groove

The Groove syntax is the following. The thin black elements in Figure 9 need to be present and will be preserved during transformation, while the dashed blue elements need to

be present but will be removed. Furthermore, the fat green elements will be created, and the dashed fat red elements represent the NACs, whose presence prevents the rule from being applied.

2.3. **Activities & Subprocesses.** Activities represent work to be performed within a BPMN process, while subprocesses group parts of a BPMN model together, allowing for reusability and separation of concerns [Obj13].

Figure 10 depicts the rule generation templates for activities and subprocesses (see Figure 3). Activity execution is divided into two steps implemented in parts **(a)** and **(b)** in the rule generation template **(1)**. Part **(a)** generates one rule for each incoming sequence flow to start the activity. An activity can be started using a token positioned at any of its incoming sequence flows. This part generates the sample rule on the left of Figure 7. Having multiple incoming or outgoing sequence flows for a flow node is considered bad practice since the implicitly encoded gateways should be explicit to avoid confusion. Our formalization still supports those models not to force modelers to rewrite them, but we recommend using static analyzers to avoid such models [Cam23e].

Part **(b)** generates one rule that terminates the activity. It deletes a token at the activity and adds one at each outgoing sequence flow. This implicitly encodes a parallel gateway (see Figure 11) but should be avoided, as described earlier.

Subprocess execution is like activity execution. Part **(a)** of the template generates one rule for each incoming sequence flow. The rule deletes an incoming token and adds a process snapshot representing a subprocess. The created process snapshot is represented with a colored circle on the top left corner of the subprocess with a token at each outgoing sequence flow of its start events (similar to start graph generation). There is a *subprocess* link between the process snapshots to depict the subprocesses relation in Figure 4. If the subprocess has no start events, a token will be added to every activity and gateway with no incoming sequence flows.

Part **(b)** of the template generates one rule to delete a terminated process snapshot and adds tokens at each outgoing sequence flow. Subprocesses are terminated by the termination rule (see section 2.2).

2.4. **Gateways.** Parallel gateways represent forking and joining in the sequence flow. Exclusive gateways represent exclusive choices and merges in the sequence flow [Obj13].

Figure 11 depicts the rule generation templates for parallel and exclusive gateways (see Figure 3). A parallel gateway can synchronize and fork the control flow simultaneously. Thus, one rule is generated that deletes one token from each incoming sequence flow and adds one to each outgoing sequence flow.

Exclusive Gateways are triggered by exactly one incoming sequence flow, and exactly one outgoing sequence flow will be triggered as a result. In practice, boolean conditions using data attached to the process are attached to exclusive gateways that decide which outgoing sequence flow to follow. We do not model data flow in our formalization and instead allow each outgoing sequence flow to be explored. Thus, one rule must be generated for every combination of incoming and outgoing sequence flows. However, the resulting rule is simple since it only deletes a token from an incoming sequence flow and adds one to an outgoing sequence flow.

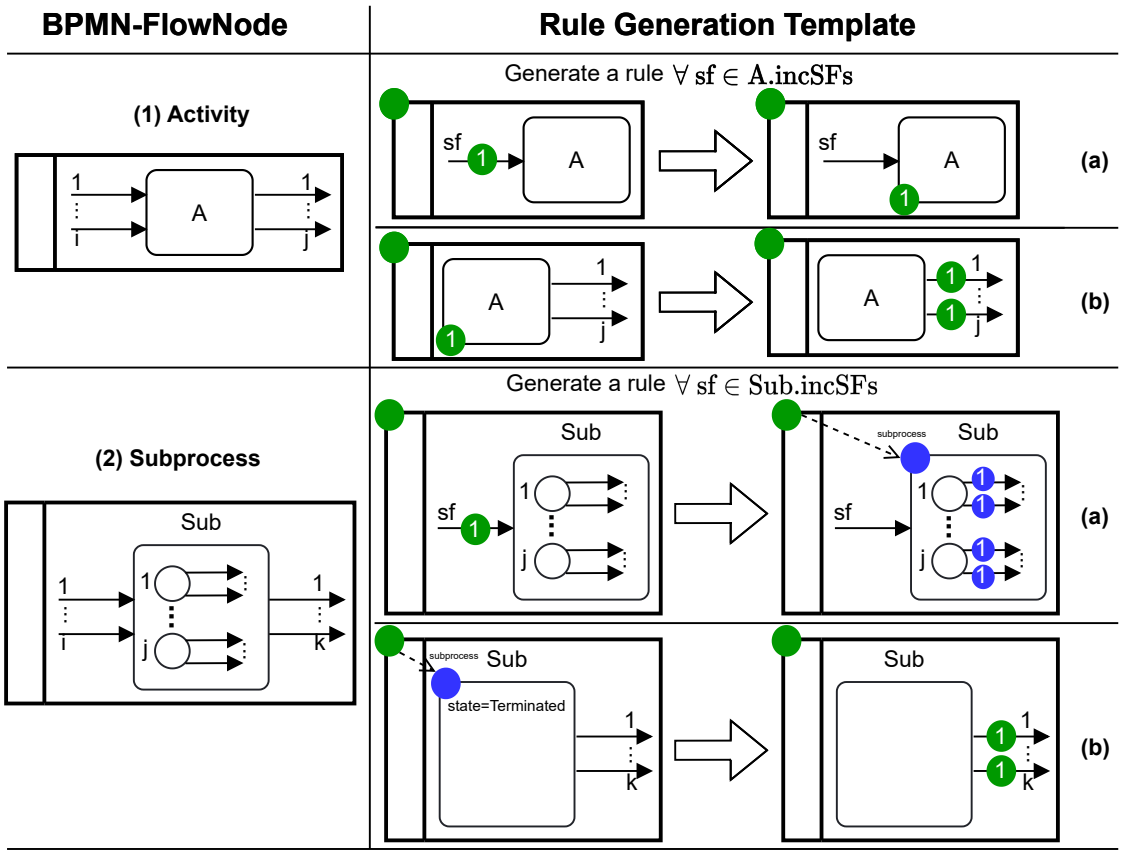


Figure 10: Rule generation template for activities and subprocesses

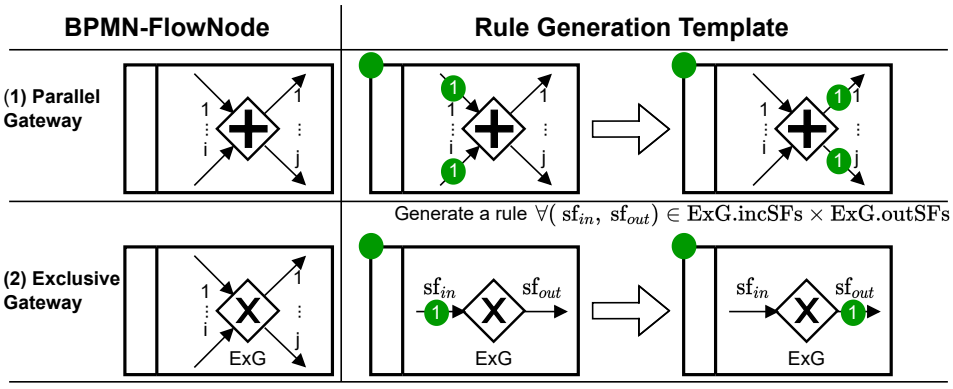


Figure 11: Rule generation template for gateways

2.5. **Message Events.** Message events are events directed at a single recipient. Thus, they are unicast compared to broadcast signal events discussed later in subsection 2.6. Figure 12 depicts the rule generation templates for *Message Intermediate Throw Events* (MITE in Figure 3). Rule generation template **(1)** describes how MITEs interact with *Message*

Intermediate Catch Events (MICEs). A MITE deletes an incoming token and adds one at each outgoing sequence flow. In addition, it sends one message to each process by adding it to the incoming messages of the process. However, sending each message is optional, meaning that if a process is not ready to consume a message immediately, the message is not added. A process can consume a message if its MICE has at least one token at an incoming sequence flow (see rule template (1) in Figure 12). We implement optional message sending using nested rules with quantification. Concretely, we use an optional existential quantifier [Ren06] (see dotted rectangle labeled Optional in Figure 12) to send a message only if the receiving process is ready to consume it.

Rule generation template **(2)** describes how MITEs interact with *Message Start Events* (MSEs). For each MSE, a new process snapshot is created with tokens located at its outgoing sequence flows. We split the interaction of MITEs with MICEs and MSEs into two rule templates for better understanding. However, a MITE might interact with MICEs and MSEs simultaneously. Thus, our HOT implements a merge of both templates. *Message End Events* (MEE) behave similarly to MITEs but only delete incoming tokens and do not add outgoing tokens.

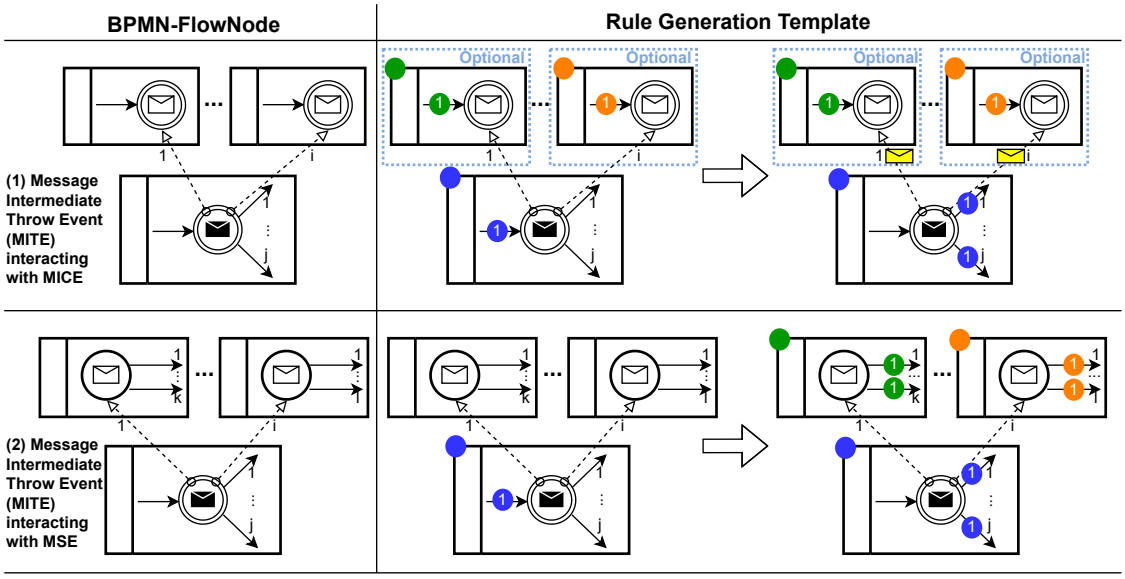


Figure 12: Rule generation templates for MITEs interacting with MICEs **(1)** and MSEs **(2)**

The rule generation template in Figure 13 shows the behavior of MICE (see MICE in Figure 3). To trigger a MICE, only one message at an incoming *message flow* is needed. Thus, one rule is generated for each incoming *message flow*. The rule template shows that MICEs delete one message and one token, as well as add a token at each outgoing sequence flow.

2.6. **Link Events.** Link events are similar to "Go To" statements since they move tokens from link throw events to link catch events in the same process level (cannot link to subprocesses). They are meant to avoid long sequence flows and connect BPMN models spanning multiple pages but can also be used to create loops due to their "Go To" nature

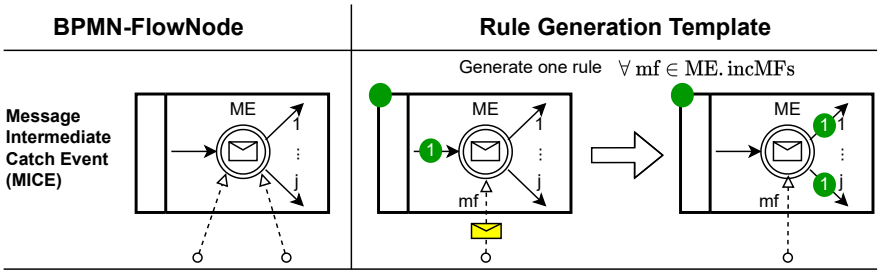


Figure 13: Rule generation templates for MICEs

[Obj13]. Figure 14 depicts the rule generation template for *Link Throw Events* (LTEs), see LTE in Figure 3. It shows how LTEs interact with *Link Catch Events* (LCEs).

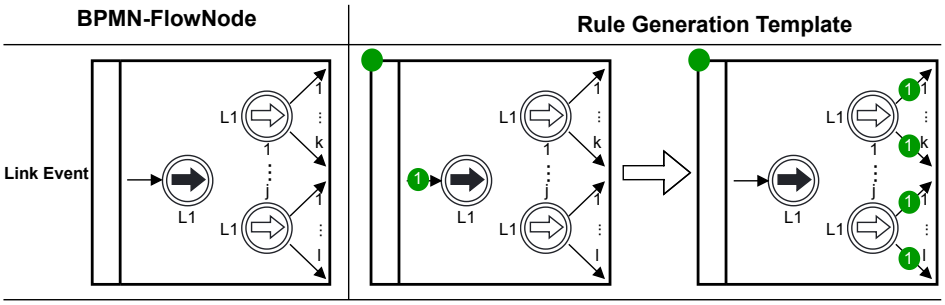


Figure 14: Rule generation template for LTEs interacting with LCEs

Each rule deletes a token at that sequence flow and adds tokens to all outgoing sequence flows of matching LTEs. An LTE matches an LCE if they have the same name or event definition (see [Obj13]). Our HOT automatically finds matching LTEs during transformation and then applies the rule template shown in Figure 14.

2.7. **Signal Events.** Each signal event is assigned a signal name. Signal throw events *broadcast* to all signal catch events with the same signal name. Signal broadcasts have a global scope, i.e., they can communicate across process levels and pools [Obj13].

Figure 15 depicts the rule generation template for *Signal Intermediate Throw Events* which interact with *Signal Intermediate Catch Events* and *Signal Start Events* (SITE, SICE, and SSE in Figure 3). *Signal End Events* (SEE) behave similarly to SITEs but only consume incoming tokens and do not add outgoing tokens.

Rule generation template **(1)** describes how SITEs interact with SICE. Like other intermediate events, the incoming token is consumed while one token is added for each outgoing sequence flow. Due to its broadcast semantics, a SITE interacts with all matching SICEs with an incoming token. A SITE and SICE match if they have the same signal name. In our templates, we assume that the signal name is the SITE/SICE name. For each matching SICE, a universally quantified nested rule consumes the incoming token and adds a token for each outgoing sequence flow. We use a universal quantifier (All in Figure 15) since one process snapshot might have multiple tokens waiting before a SICE. Then, a SITE should trigger this SICE multiple times.

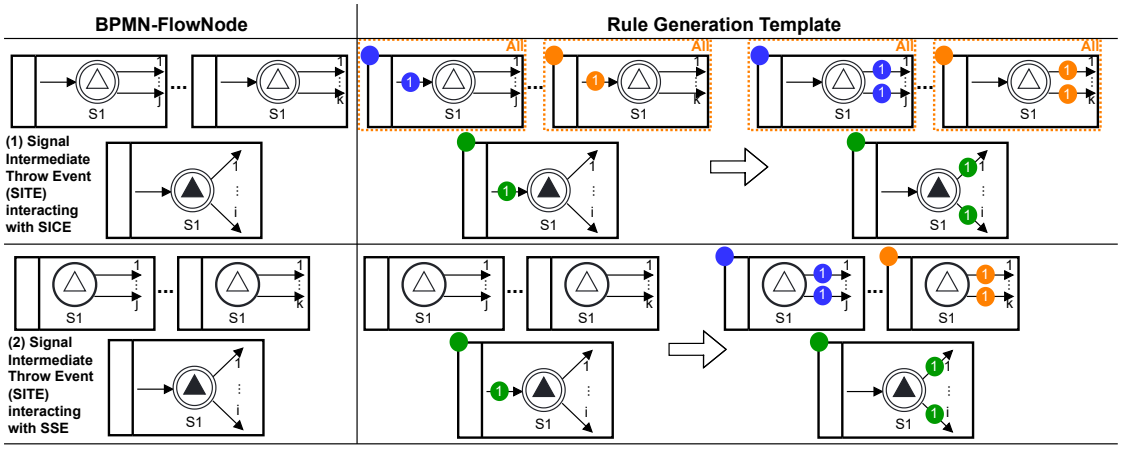


Figure 15: Rule generation templates for SITEs interacting with SICE **(1)** and SSE **(2)**

Rule generation template **(2)** describes how SITEs interact with SSEs. Analogous to MITEs and MSEs, new process snapshots with tokens at the outgoing sequence flows of the SSEs are added for each matching SSE. Each matching SSE is only triggered once, meaning we do not need any quantified nested rules. We split the interaction of SITEs with SICEs and SSEs into two rule templates for better understanding. However, a SITE might interact with SICEs and SSEs simultaneously. Thus, our HOT implements a merge of both templates.

2.8. **Terminate Events.** A *Terminate End Event* (TEE) abnormally terminates the running process [Obj13], meaning the process changes its state to terminated, and all its tokens are consumed. Figure 16 depicts the rule generation template for TEEs (see TEE in Figure 3).

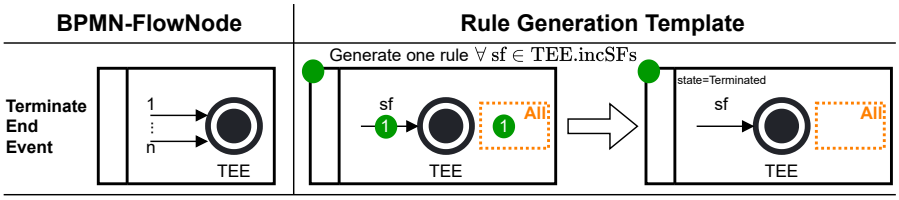


Figure 16: Rule generation template for terminate end events

One rule is generated for each incoming sequence flow of a TEE. The rule consumes the incoming token, similar to the rules for end events, but also changes the process snapshot state to Terminated. In addition, the rule deletes all other tokens of the process snapshot using a universally quantified nested rule (see dotted rectangle labeled **All** in Figure 16). Terminating a process must also terminate its subprocesses, which is not shown in the rule template in Figure 16 for brevity; it is described in our wiki [Krä23].

3. Model checking BPMN

Model checking —and verification in general— of BPMN models is necessary to ensure the correctness and reliability of business processes, which ultimately leads to increased efficiency, reduced costs, and user satisfaction. Using our formalization approach, BPMN models may be verified against behavioral properties —both general and custom— by utilizing the generated GT system (see Figure 1). These behavioral properties are defined using temporal logic, such as Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) [BK08, CHVB18]. As mentioned in section 1, modelers may use our approach to specify custom properties consisting of atomic propositions and operators from CTL/LTL. The atomic propositions are transformed to graph conditions by the HOT. A graph condition is a GT rule which does not delete or add elements. A proposition holds in a given state if a match of the graph condition exists in the graph representing the state [KR06].

We differentiate between two types of behavioral properties: *general BPMN properties* defined for all BPMN models and *custom properties* tailored towards a particular BPMN model. We do not consider structural properties (like conformance to BPMN syntax) since they can be checked using a standard modeling tool without implementing execution semantics. We will now give an example of predefined general BPMN properties and show how our approach can check them. Then, we describe how custom properties can be defined and checked.

3.1. General BPMN properties.

Safeness and *Soundness* properties are defined for BPMN in [CMRT18]. A BPMN model is *safe* if, during its execution, at most one token occurs along the same sequence flow [CMRT18]. Soundness is further decomposed into (i) *Option to complete*: any running process instance must eventually complete, (ii) *Proper completion*: after completion, each token of the process instance must be consumed by a different end event, as well as (iii) *No dead activities*: each activity can be executed in at least one process instance [CMRT18]. Process completion is synonymous with process termination. In the following, we will describe how to implement the *Safeness* and *Option to complete* using CTL, as well as *Proper completion* and *No dead activities* by analyzing the GT system's state space.

We specify *Safeness* as the following CTL property:

$$AG(\neg\,\text{Unsafe}) \tag{3.1}$$

The path quantifier A means the following proposition $G(\neg\,\text{Unsafe})$ should hold for *all* paths starting from the current state. The temporal operator G means the following proposition $\neg\,\text{Unsafe}$ should hold at all states in the future [CHVB18]. More detailed information about CTL can be found in [CHVB18, BK08]. Combining the path quantifier A and temporal operator G in (3.1) means $\neg\,\text{Unsafe}$ should hold for all states in all paths starting from the initial state. Thus, (3.1) describes that a state labeled as Unsafe should not be reachable. The atomic proposition Unsafe is true if two tokens of one process snapshot point to the same sequence flow. Atomic propositions are either fulfilled Figure 17 shows how Unsafe is represented as a graph condition in Groove.

Option to complete is specified using the following CTL property:

$$AF(\text{AllTerminated}) \tag{3.2}$$

The temporal operator F means the following proposition AllTerminated should hold in some state in the future [CHVB18]. Thus, (3.2) describes that a state labeled as

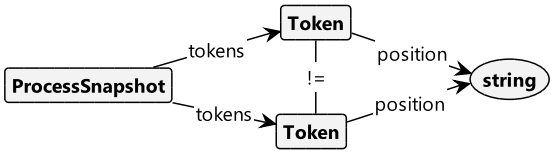


Figure 17: The atomic proposition Unsafe as a Groove graph condition.

AllTerminated should be reached for all paths starting from the initial state. The atomic proposition AllTerminated is true if there exists no process snapshot in the state Running, i.e., all process snapshots are Terminated. Figure 18 shows how Terminated is represented as a graph condition in Groove.

Figure 18: The atomic proposition AllTerminated as a Groove graph condition.

Checking the properties *Safeness* and *Option to complete* is implemented by checking the CTL properties above using Groove [KR06, Ren08]. The property *Proper Completion* is implemented by checking the GT system's state space for two executions of an end event in the same path. Similarly, *No dead activities* is implemented by analyzing the GT system's state space to see if each activity has been executed at least once [Krä23].

3.2. **Custom properties.** To make model checking user-friendly, we enable modelers to define atomic propositions using the concrete syntax of the extended BPMN execution metamodel introduced in Figure 4 (see Figure 19). An atomic proposition is defined as a process snapshot with a token distribution, which we can automatically convert to a graph condition in Groove (see Figure 20). Recall that graph conditions are GT rules that do not add or delete elements. Atomic propositions may be connected by CTL operators to create temporal formulas that should hold in the given BPMN model. Furthermore, modelers may forbid certain states in the BPMN model by specifying that a certain token distribution should not exist. These situations would lead to Negative Application Conditions (NACs) in the graph conditions.

For example, the token distribution shown in Figure 19 defines a process snapshot with two tokens at activity *Ship goods*. A modeler could use this atomic proposition to check if the activity *Ship goods* is executed twice by creating an appropriate CTL property. Shipping goods twice but only receiving one payment during an order-handling process would be a critical error for a business. The order handling process in Figure 19 is taken from [Rüc21] but changed to contain a modeling error. It contains an exclusive gateway instead of a parallel gateway. The modeling error could lead to shipping goods twice if the process is not corrected before deployment.

Another proposition for the same process with a different error is shown in Figure 21. The proposition *noShipment* defines that the activity *Ship goods* should not run (has no token). "Has no token" is depicted by crossing out the token symbol and represents an extension of our concrete syntax introduced for defining propositions. This proposition can be used to define a CTL property to check if shipping always occurs. In this case, the error

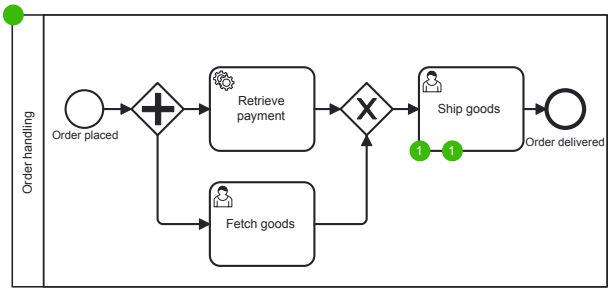


Figure 19: Atomic proposition *shipGoodsTwice* defining shipping goods twice.

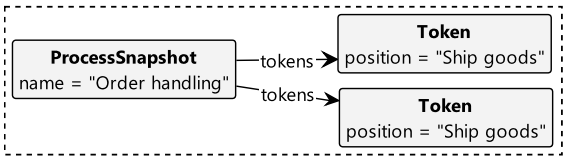


Figure 20: Generated Groove graph condition describing the atomic proposition in Figure 19.

in the order handling process prevents shipping from occurring. The GT systems for both variants of the order handling process containing the propositions can be found in [Krä23].

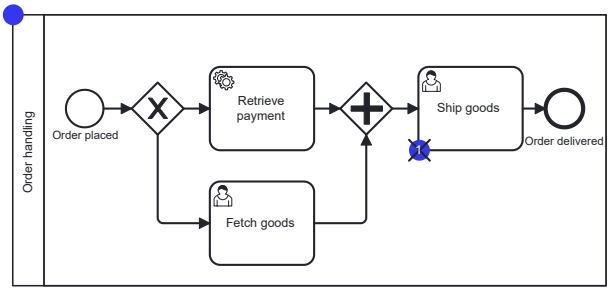


Figure 21: Atomic proposition *noShipment* defining no ongoing shipping of goods.

Using an atomic proposition editor based on the BPMN concrete syntax, modelers do not need extensive knowledge about the GT-based execution semantics. Although the expressiveness is not as powerful as in the GT-based execution semantics in Groove—e.g., one can use nested rules with quantification in graph conditions—we favor *simplicity* over *expressiveness*. In addition, we attempt to stay as independent as possible from the framework and tools used for the execution semantics (see the right part in Figure 1).

Finally, the modeler must still know temporal logic to specify custom properties. To combat this problem, we added temporal logic templates to our tool to generate commonly occurring propositions without knowledge about temporal logic. The next section about the BPMN Analyzer discusses this feature in detail. A domain-specific property language for BPMN would further lessen the knowledge required from the modeler [MDL+14].

4. BPMN Analyzer

Our approach is implemented as a web-based tool called *BPMN Analyzer*, which is open-source, publicly available, and does not require any installation [Krä23, KRKL23]. A demonstration of the BPMN Analyzer is available online[1]. Figure 22 depicts a screenshot of the BPMN Analyzer. We use the order handling process from [Rüc21] as an example. It is the same BPMN model as in Figure 19 and Figure 21 but without modeling errors.

The modeler can create or upload a BPMN model, which can then be verified using either general BPMN properties or custom properties formulated in CTL. BPMN Analyzer generates a GT system for the supplied BPMN model and runs model checking against the specified properties in Groove [KR06, Ren08]. We have created a comprehensive test suite [Krä23], which verifies that rules are generated as defined by the rule generation templates in the previous section. The test suite covers over 90% of our source code.

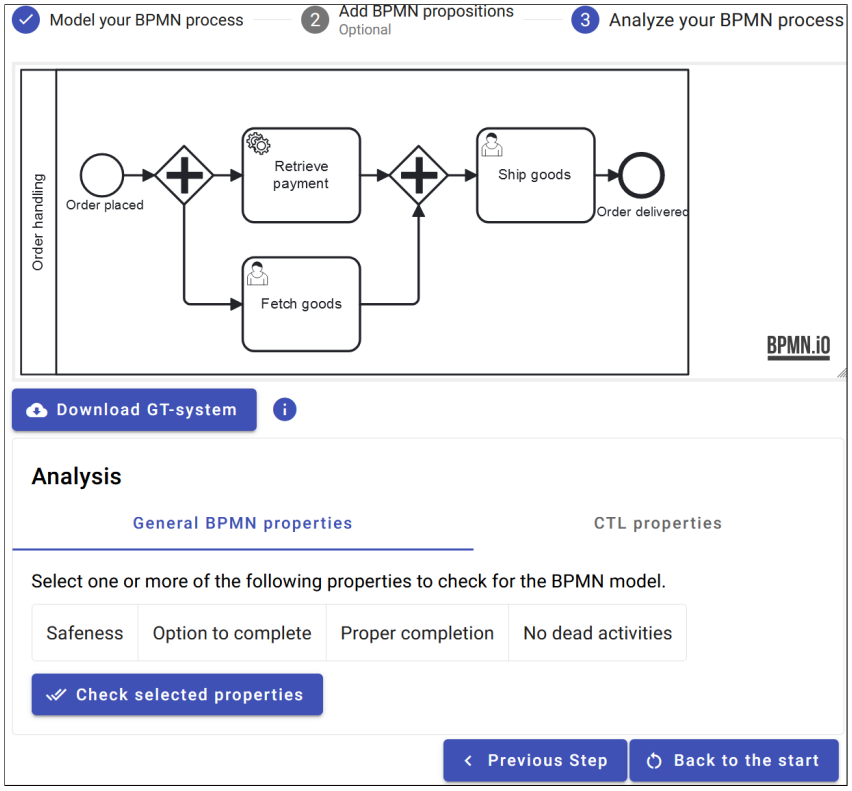


Figure 22: Screenshot of the *analysis step* in the BPMN Analyzer

The BPMN Analyzer interface is structured into three steps that guide the user transparently through the modeling and analysis process.

(1) The **Modeling** step lets users upload or define the BPMN model. We utilize a properties panel in the modeling step so that IDs of BPMN elements can be viewed and edited directly in the model editor. This allows for better traceability between BPMN elements and generated GT rules if a user inspects the GT system.

[1] https://youtu.be/MxXbNUl6IjE

(2) The **BPMN Propositions** step contains our custom *Token Editor* and is shown in Figure 23. In this step, users may create atomic propositions, which can be used as ingredients in the custom CTL properties in the analysis step. In Figure 23, the user is editing one of two created propositions. Users who are only interested in general BPMN properties may skip this step. Atomic propositions are created using the concrete syntax detailed in subsection 3.2, implemented in our *Token Editor*. As mentioned, users attach tokens and process snapshots to the BPMN model created in the modeling step to create a proposition.

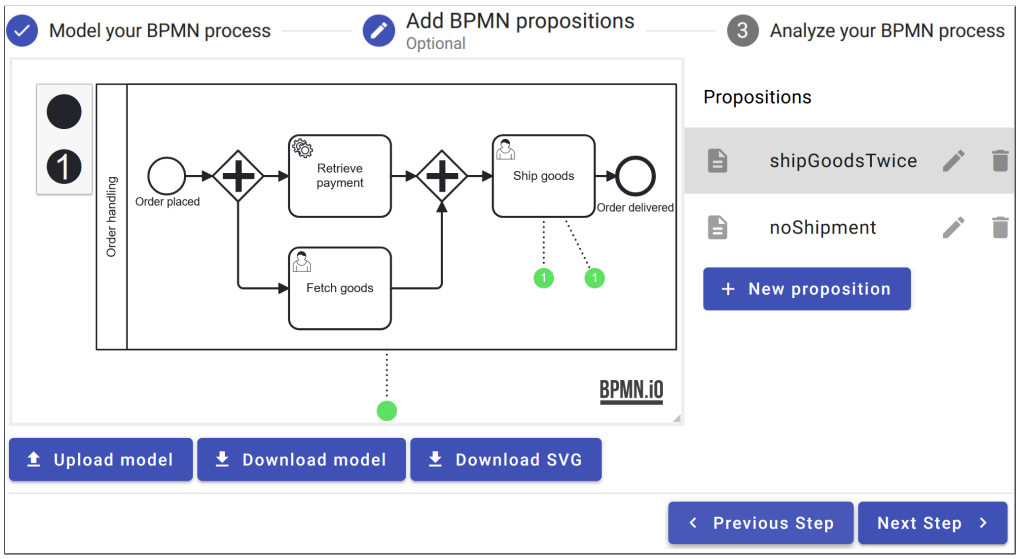


Figure 23: Screenshot of the *propositions step* in the BPMN Analyzer

(3) In the **Analysis** step, users may check the general BPMN properties and build custom CTL properties using the atomic propositions. The properties builder utilizes the textual CTL syntax implemented in Groove to specify custom properties using the atomic propositions from the previous step, see Figure 24. The two atomic propositions created in the previous step (see Figure 23) are available to the user. The CTL properties builder comes with CTL templates to facilitate commonly occurring CTL properties. These templates allow users to check whether a state (described by an atomic proposition) can be reached or is never reached (see Figure 24). Thus, simple safety and liveness properties can be checked using these templates. Model-checking experts in an organization can define more templates in the future and share them for reuse. Users who download the generated GT system may inspect and edit the graph conditions generated from the atomic propositions; they may also specify more properties and check them using Groove.

4.1. **Reusable libraries.** In addition to the BPMN Analyzer, we published multiple parts of our application as libraries that can be reused seamlessly by other researchers and practitioners. The BPMN metamodel extension needed to define atomic propositions is published as an npm module (*token-bpmn-moddle*) [Krä23]. Npm is the default package manager for the JavaScript programming language. In addition, the Token Editor to create

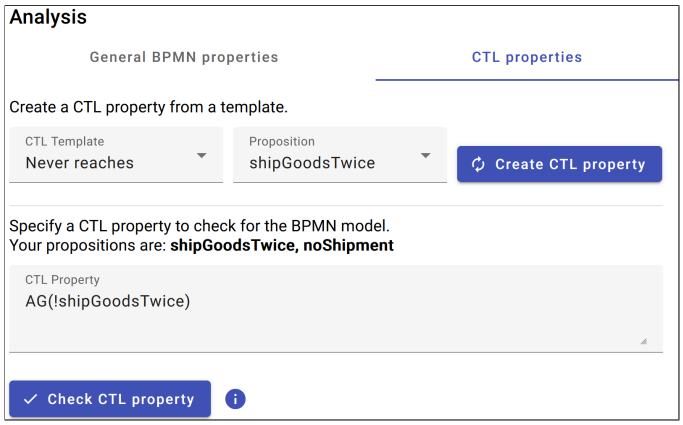


Figure 24: CTL properties builder in the *analysis step* of the BPMN Analyzer

atomic propositions is also published as an npm module (*token-bpmn*) [Krä23]. Furthermore, we published our *graph-rule-generation* library[2] to generate Groove GT systems to the Maven central repository [Krä23]. The Maven central repository is the standard repository for developing JVM-based applications. We explain each library in detail in the following sections.

4.1.1. *BPMN metamodel extension.* Our implementation *token-bpmn-moddle* [Krä23] extends *bpmn-moddle* [Cam23d], which implements the BPMN specification. Our extension adds the Token and ProcessSnapshot types from the BPMN execution metamodel shown in Figure 4 to the BPMN metamodel. Listing 1 shows an example BPMN XML snippet, where a token and process snapshot was added.

Listing 1: XML snippet showing the BPMN metamodel extension (simplified)

```
1 <process id="Process_1">
2   <extensionElements>
3     <bt:processSnapshot id="ProcessSnapshot_1" />
4     <bt:token id="Token_1" processSnapshot="ProcessSnapshot_1"
5         elementID="Task_A"/>
6   </extensionElements>
7   <task id="Task_A" />
8 </process>
```

The library allows one to create tokens and process snapshots and stores them in the BPMN extension elements (lines 2-7 in the XML example in Listing 1). This is the recommended way to extend the BPMN metamodel [Obj13].

The extension of the BPMN metamodel is realized by letting Token and ProcessSnapshot extend from Element, which is the type of elements of the extension elements, see Figure 25. Each Token points to the FlowElement it is currently positioned at, which can be an Activity or SequenceFlow, see BPMN metamodel in Figure 2. In Listing 1, this is realized using the attribute elementID.

[2] https://mvnrepository.com/artifact/io.github.timKraeuter/graph-rule-generation

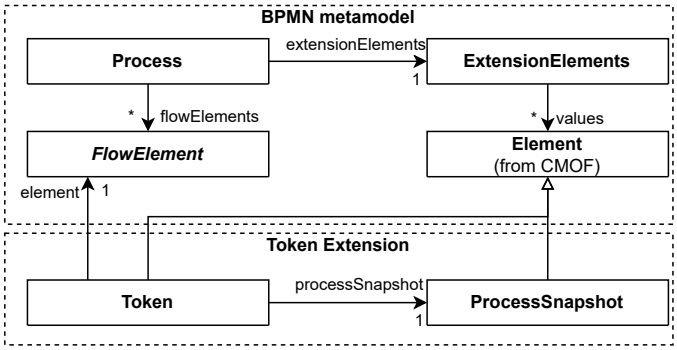


Figure 25: Token Extension of the BPMN metamodel (simplified)

The XML model in Listing 1 can be used by our HOT to generate atomic propositions for Groove (see Figure 1). However, following sound model-driven principles and creating an extended metamodel, the XML model could also be used in other applications or for model checking with different tools.

4.1.2. *Token Editor.* The Token Editor implements the concrete syntax for tokens and process snapshots described in Figure 4. Using our Token Editor, a user does not need to write XML, which hides the complexity of extending the BPMN metamodel in the graphical editor.

Figure 23 shows the Token Editor embedded in the second step of the BPMN Analyzer. We simplify the user interface so users can only edit tokens and process snapshots, not the underlying BPMN model. Process snapshots are automatically assigned distinct colors, and tokens held by a process snapshot have the same color (see concrete syntax in Figure 4). Tokens can be assigned to process snapshots, which changes the token's color to match the snapshot.

Our implementation is based on *bpmn-js* [Cam23b], which provides a BPMN rendering toolkit and uses the *token-bpmn-moddle* library described in the previous section to persist and load our models. Since both implementations are published as libraries, they can be easily reused in other applications.

4.1.3. *Graph rule generation.* The graph-rule-generation library offers various Java classes to generate graphs, GT rules, or entire GT systems, following the *builder pattern* [GHJV95]. Listing 2 shows an example code snippet to generate a GT rule using the GT rule builder implemented for Groove. One could also implement the GT rule builder for a different GT tool than Groove, which would only result in changes in the first two lines of Listing 2.

Listing 2: Code snippet to generate a GT rule using the GT rule builder

```
1 GraphTransformationRuleBuilder ruleBuilder
2     = new GrooveRuleBuilder();
3 // Start a new GT rule with the name "sampleRule"
4 ruleBuilder.startRule("sampleRule");
5 // Create context nodes A and B and edge AB from A->B.
6 GraphNode a = ruleBuilder.contextNode("A");
```

```
7  GraphNode b = ruleBuilder.contextNode("B");
8  ruleBuilder.contextEdge("AB", a, b);
9  // Delete nodes C and D and edge CD from C->D.
10 GraphNode c = ruleBuilder.deleteNode("C");
11 GraphNode d = ruleBuilder.deleteNode("D");
12 ruleBuilder.deleteEdge("CD", c, d);
13 // Add nodes E and F and edge EF from E->F.
14 GraphNode e = ruleBuilder.addNode("E");
15 GraphNode f = ruleBuilder.addNode("F");
16 ruleBuilder.addEdge("EF", e, f);
17 // Create NAC nodes G and H and edge GH from G->H.
18 GraphNode g = ruleBuilder.nacNode("G");
19 GraphNode h = ruleBuilder.nacNode("H");
20 ruleBuilder.nacEdge("GH", g, h);
21 // Build the GT rule.
22 GraphTransformationRule gtRule = ruleBuilder.buildRule();
```

Using the rule builder, one can construct a GT rule by defining which nodes and edges should be present (lines 6-9), deleted (lines 10-13), added (lines 14-17), or NACs (lines 18-21), see Listing 2. Figure 26 shows the resulting GT rule specified by Listing 2 in Groove syntax.

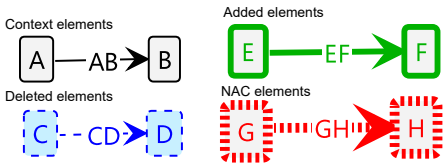


Figure 26: Groove GT rule generated by the code snippet in Listing 2

Similarly to the groove rule builder in Listing 2, we also provide a builder for constructing graphs to create start graphs of GT systems [Krä23]. Finally, the library provides a builder for GT systems using the graph and GT rule builders. It also automatically lays out graphs and GT rules using the Eclipse Layout Kernel (ELK). The Groove UI and the Groove command-line tools can then consume the generated GT systems.

4.2. **Performance testing.** Model checking is a valuable technique but can fall short when applied in the industry due to insufficient performance [CHVB18]. Inadequate performance might have many reasons, most notably large models leading to state space explosion. We assess the performance of our implementation for two sets of BPMN models. First, we pick ten BPMN models from the literature, including realistic ones. Second, we generated ten BPMN models with exponential state space growth to test how our tool deals with increasing complexity. We explain each benchmark and its results in the following subsections and provide all necessary information and artifacts to reproduce them in [Krä23]

To calculate the average runtime, we use the hyperfine benchmarking tool [Pet23] (version 1.18.0), which runs the HOT/state space exploration for each BPMN model ten or more times. The experiment was run on Windows 11 (AMD Ryzen 7700X processor, 32 GB RAM) using Groove version 6.1.0 [Krä23].

4.2.1. *BPMN models from the literature.* We randomly picked ten different BPMN models from [HBP+22] to assess the performance of our implementation. The models include realistic BPMN models (e.g., 001, 002, and 020) [HBP+22].

First, we ran our HOT for the BPMN models. The HOT takes approximately half a second to generate a GT system for each model. Thus, the generation of the GT systems for these models is fast enough. In addition, Table 1 states how many GT rules are generated for each BPMN model.

Second, we ran a full state exploration using the resulting ten GT systems, see Table 1 (runtime only includes state space exploration). The exploration takes around one second for most of the models. Only model *020* needs nearly two seconds due to its larger state space. Furthermore, up to one second is spent on startup, not model checking. For example, Groove reports only 722 ms for state space exploration for model *020*.

Table 1: Results for a full state space exploration of realistic models

BPMN model	Processes	Nodes (gw.)	GT Rules	States	Transitions	Runtime
001	2	17 (2)	26	68	118	$\sim$ 1.00 s
002	2	16 (2)	24	62	108	$\sim$ 0.97 s
007	1	8 (2)	14	45	81	$\sim$ 0.92 s
008	1	11 (2)	17	49	85	$\sim$ 0.93 s
009	1	12 (2)	17	137	308	$\sim$ 1.01 s
010	1	15 (2)	20	162	357	$\sim$ 1.04 s
011	1	15 (2)	20	44	69	$\sim$ 0.97 s
015	1	14 (2)	20	53	86	$\sim$ 0.95 s
016	1	14 (2)	19	44	68	$\sim$ 0.94 s
020	1	39 (6)	59	3060	8584	$\sim$ 1.75 s

4.2.2. *BPMN models with increasing complexity.* To increase the state space complexity, we generate BPMN models with a growing number of parallel branches, similar to [CFP+21]. Figure 27 shows our schema to generate models. The possible interleavings of executing activities in parallel lead to an exponential increase in the state space. Concretely, we generated ten BPMN models with one to ten parallel branches containing one activity [Krä23]. We use these models to benchmark our implementation.

First, we ran our HOT for the BPMN models. The HOT takes less than a second to generate a GT system for each model. Thus, the generation of the GT systems for these models is fast enough.

Second, we ran a full state exploration using the resulting ten GT systems, see Table 2 (runtime only includes state space exploration). The models take one to nine seconds to explore. One can see an exponential increase in runtime due to the exponential increase in state space complexity.

We conclude that our approach is sufficiently fast for models of average size and complexity. In the next section, we test the scalability of our approach when models increase in size. Furthermore, we discuss potential performance and scalability improvements. However, a comprehensive benchmark, including a detailed comparison to other tools, is left for future work.

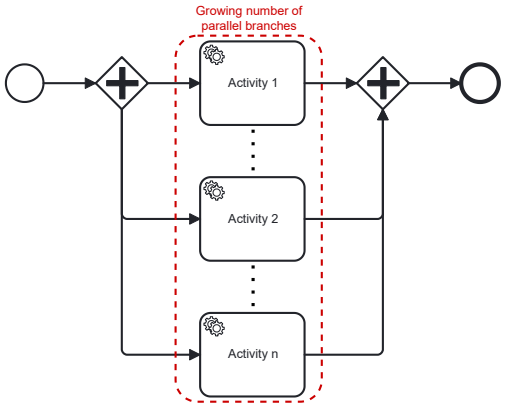


Figure 27: BPMN model generation with an increasing number of parallel branches

Table 2: Results for a full state space exploration of models with increasing complexity

Branches	Nodes (gw.)	GT Rules	States	Transitions	Runtime
1	5 (2)	9	7	7	∼ 0.87 s
2	6 (2)	11	13	17	∼ 0.86 s
3	7 (2)	13	31	59	∼ 0.88 s
4	8 (2)	15	85	221	∼ 0.95 s
5	9 (2)	17	247	815	∼ 1.03 s
6	10 (2)	19	733	2921	∼ 1.15 s
7	11 (2)	21	2119	10.211	∼ 1.49 s
8	12 (2)	23	6.565	34.997	∼ 2.13 s
9	13 (2)	25	19.687	118.103	∼ 3.85 s
10	14 (2)	27	59.053	393.665	∼ 9.16 s

4.3. **Scalability testing.** In this section, we test the scalability of our approach by applying it to 300 heterogeneous BPMN models with increasing model sizes.

4.3.1. *Setup.* We generated 300 BPMN models to test the scalability of our approach. We used the following strategy to include different BPMN elements in the models. We generated the models incrementally, increasing the number of *blocks* they contain. Thus, model one contains one block, model two contains two blocks, and so forth until the last model contains 300 blocks. A block is defined as one of the three BPMN model parts shown in Figure 28. During the generation, we alternate between the three different blocks.

For example, the BPMN model with three blocks is depicted in Figure 29. Blocks two and three are shown in a new line for better visualization. However, the generated models are expanding horizontally in one line. We then repeat adding one block at a time for each new model until we reach 300 models.

Table 3 states the characteristics of the generated BPMN models, such as the number of gateways, flow nodes, and sequence flows. One can deduce from the table that adding fifty blocks adds around 400 BPMN elements to a model. All models, including their

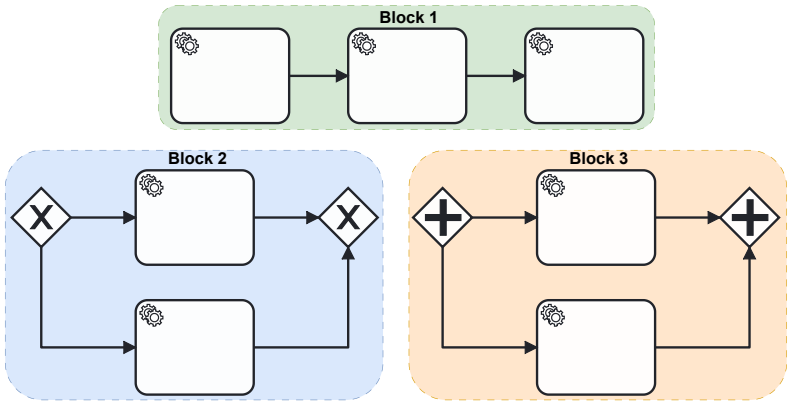


Figure 28: The three different blocks used for BPMN model generation

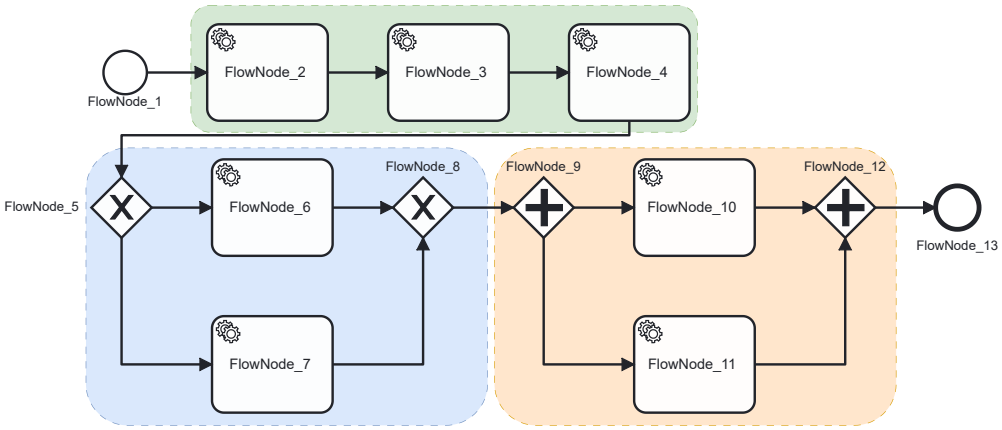


Figure 29: A generated BPMN model with three blocks

characteristics and how to generate them, can be found in [Krä23]. Our BPMN model generation uses the camunda BPMN model API [Cam23f].

BPMN models in practice tend to be much smaller since large models are usually divided into smaller submodels [FFK+11], i.e., subprocesses, to ensure they are understandable by modelers. Each of these subprocesses can then be analyzed independently. From our experience and referring to other studies [FFK+11], this best practice leads to models with less than 400 total elements (comparable to less than 50 blocks in Table 3). We ran our scalability test for models with up to 300 blocks since we wanted enough data to see trends in the average runtime. We did not go beyond 300 blocks since the whole test should still run in a reasonable time.

4.3.2. *Results.* Figure 30 depicts the results of benchmarking our HOT with the generated BPMN models. It shows the average runtime of five runs for transforming each BPMN model into a GT system using our HOT. We used the same machine and setup as discussed for our performance experiments in section 4.

Table 3: Characteristics of the generated BPMN models

BPMN model / Blocks	Gateways	Flow nodes	Sequence flows	Total elements
1	0	5	4	9
50	66	185	217	402
100	132	368	433	801
150	200	552	651	1203
200	266	735	867	1602
250	332	918	1083	2001
300	400	1102	1301	2403

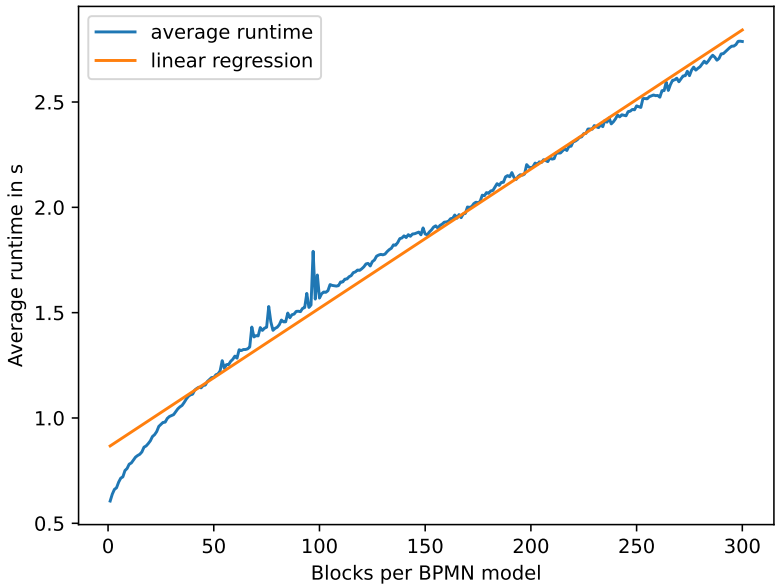


Figure 30: Scalability testing result of the GT system generation

The average HOT runtimes data fits the linear regression shown in Figure 30 well. This makes sense since the HOT algorithm has *linear runtime complexity* because it iterates over all flow nodes of a BPMN model to generate GT rules. We conclude that the HOT is fast enough (around one second or less) for models of reasonable size (50 blocks or less).

Figure 31 depicts the results of benchmarking the state space generation in Groove for the GT systems obtained by our HOT. It shows the average runtime of five runs, calculated by hyperfine [Pet23].

The increase in the runtime of the state space generation looks worse than linear. However, models of reasonable size (50 blocks or less) are handled in less than two seconds. To summarize, using our approach, we can conservatively estimate that these models can be checked (HOT followed by a full state space generation) in around three seconds or less. In

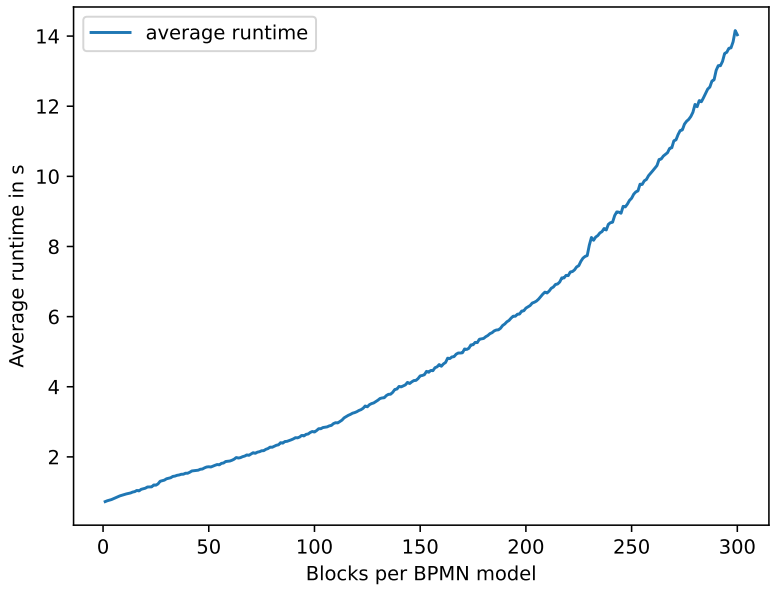


Figure 31: Scalability testing result of the state space generation in Groove

addition, full-state space exploration might not be needed if *on-the-fly* model checking is used [CHVB18].

On-the-fly model checking allows temporal properties to be checked incrementally while the state space is constructed. If a property violation is detected, there is no need to further complete the construction of the state space. This can considerably reduce the time and memory required for verification [CHVB18].

Plenty of optimization potential exists, starting from the HOT and ending with the state space generation in Groove. Currently, neither our HOT nor Groove are specifically optimized for performance with this use case in mind. Regarding the HOT, multiple optimizations come to mind. First, one can parallelize the generation of GT rules since each rule is independent. Second, one can change the rule-generation templates to reduce the number of generated rules and the state space. For example, two rules are currently generated to represent starting and finishing an activity (see Figure 10), which fits the description in the BPMN specification. However, one could instead generate only one rule, which represents the whole execution of an activity. Thus, one less rule is generated, and the intermediate state representing the activity executing is no longer part of the state space. If there are many activities, especially when they are executed in parallel, this can lead to large reductions in the state space. However, a less granular state space could prohibit checking certain properties. A trade-off exists between staying close to the BPMN execution specification and overall runtime (HOT and state space generation).

Groove is a powerful tool with good out-of-the-box performance. However, there is still optimization potential. First, GT rules should not be written to and read from the disk to interact with Groove. Each GT rule is saved to a new file, and the number of generated GT rules increases with the size of a BPMN model. Integrating our HOT and Groove more

tightly can eliminate these costly I/O operations since the generated rules can stay in the main memory. Second, *partial order reduction* methods could greatly reduce the time and space required for model checking [CHVB18]. Third, Groove could use *on-the-fly* model checking as mentioned by the Groove authors [KR06]. If one combines verification and state space exploration and finds a counterexample, there is no need to continue the state space generation [KR06, CHVB18].

5. Related work

The most common formalizations of BPMN execution semantics use Petri Nets. For example, [DDO08] formalizes a subset of BPMN elements by defining a mapping to Petri Nets conceptually close to our HOT-based formalization. Especially Petri net model checkers such as LoLA show great performance when analysing business process models [FFK$^+$11] but supporting the same variety of BPMN elements can be problematic. Encoding basic BPMN modeling elements into Petri Nets is generally straightforward, but for some advanced elements, it can be complicated to define [HA02]. For example, representing *Termination End Events* and *Interrupting Boundary Events*, which interrupt a running process, is usually unsupported because of the complexity of managing the non-local propagation of tokens in Petri Nets [CFP$^+$21]. We solve these situations by using nested graph conditions, for example, to remove all tokens when reaching a *Termination End Event*.

A BPMN formalization based on in-place GT rules is given in [VGD13]. The formalization covers a substantial part of the BPMN specification, including complex concepts such as inclusive gateways and compensation. In addition, the GT rules are visual and thus can be aligned with the informal description of the execution semantics of BPMN. A key difference to our approach is that the rules in [VGD13] are general and can be applied to every BPMN model, while we generate specific rules for each BPMN model using our HOT. Thus, our approach can be seen as a program specialization compared to [VGD13] since we process a concrete BPMN model before its execution. However, they do *not* support property checking since their goal is only to formalize the BPMN execution semantics.

The tool *BProVe* is based on formal BPMN semantics given in rewriting logic and implemented in the Maude system [CFP$^+$21]. Using this formal semantics, *BProVe* can verify custom LTL properties and general BPMN properties, such as Safeness and Soundness. However, *BProVe* only supports the most common BPMN elements, as shown later. Regarding performance, [CFP$^+$21] describes a timeout (runtime greater than 600 seconds) for state space generation of a model with five parallel branches. In subsection 4.2, we show that our tool takes less than ten seconds for the state space exploration of a model with ten parallel branches.

The verification framework fbpmn uses first-order logic to formalize and check BPMN models [HBP$^+$22]. This formalization is then realized in the TLA$^+$ formal language, which can be model-checked using TLC. TLC is an explicit state model checker for TLA$^+$ specifications. Like BProVe, fbpmn allows checking general BPMN properties, such as Safeness and Soundness. Furthermore, *fbpmn* focuses on different communication models besides the standard in the BPMN specification and supports time-related constructs. In our approach, we currently disregard time-related constructs [DS17, HBP$^+$22] and data flow [CMR$^+$22, El-15] but rather support more BPMN elements. Regarding performance, [HBP$^+$22] report 3.66-10.26s on a machine with less powerful hardware compared to our 1-1.75s for the models in section subsection 4.2. Thus, including the previous comparison to BProVe, we conclude

that our tool performs well. However, assessing tool performance is difficult when there are no standardized benchmarks that allow for a direct comparison of results in the same environment.

Table 4 shows which BPMN elements are supported by our and the abovementioned approaches. Compared to the other approaches, we cover most BPMN elements. The coverage of BPMN elements significantly impacts how practical each approach is in checking properties in real life [FFK+11]. In addition, we cover the most important elements found in practice since we come close to the element coverage of popular process orchestration platforms such as Camunda [Cam23a].

The BPMN elements that our approach does not support, compared to Camunda, are transactions, cancel events, and compensation events. These elements are rather complex, but [VGD13] shows how cancel and compensation events can be formalized. We plan to support these elements by extending our implementation and test suite in the future.

6. Conclusion & future work

This article reports two main practical contributions. First, we conceptualize a new approach utilizing a Higher-Order model Transformation (HOT) to formalize the semantics of behavioral languages. Our approach moves complexity from the GT rules to the rule templates, which constitute the HOT. Furthermore, the approach can be applied to other behavioral languages as long as one can define the *state structure* and identify *state-changing elements* of the language.

Second, we apply our approach to BPMN, resulting in a comprehensive formalization regarding element coverage (compared to the literature and industrial process engines) that supports checking behavioral properties. Furthermore, our contribution is implemented in an open-source web-based tool to make our ideas easily accessible to other researchers and practitioners. In addition, our performance and scalability testing indicates that the tool can handle most BPMN models found in practice.

Future work targets both of our main contributions. First, we plan a detailed comparison of our HOT approach with approaches that utilize fixed model-independent rules. It will be interesting to investigate how the two approaches differ, for example, in runtime during state space generation. Second, we aim to improve our formalization and the resulting tool in multiple ways. We intend to extend our formalization to support the remaining few BPMN elements used in practice and want to turn the modeling environment of our tool into an interactive simulation environment. In addition, we can use this environment to visualize potential counterexamples in cases where behavioral properties are violated.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and helpful suggestions.

References

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[Cam23a] Camunda Services GmbH. BPMN 2.0 Implementation Reference. https://docs.camunda.org/manual/7.19/reference/bpmn20/, October 2023.

[Cam23b] Camunda Services GmbH. Bpmn-js. https://github.com/bpmn-io/bpmn-js, October 2023.

Table 4: BPMN elements supported by different formalizations (based on [VGD13]).

BPMN element/feature	Dijkman [DDO08]	Van Gorp [VGD13]	Corradini [CFP+21]	Houhou [HBP+22]	This article
Instantiation and termination					
Start event instantiation	X	X	X	X	X
Exclusive event-based gateway instantiation		X			X
Parallel event-based gateway instantiation					
Receive task instantiation					X
Normal process completion	X	X	X	X	X
Activities					
Activity	X	X	X	X	X
Loop activity	X	X			
Multiple instance activity					
Subprocess	X	X		X	X
Event subprocess					X
Transaction					
Ad-hoc subprocesses					
Gateways					
Parallel gateway	X	X	X	X	X
Exclusive gateway	X	X	X	X	X
Inclusive gateway (split)	X	X	X	X	X
Inclusive gateway (merge)		X		X	X
Event-based gateway			X[1]	X	X
Complex gateway					
Events					
None Events	X	X	X	X	X
Message events	X	X	X	X	X
Timer Events				X	
Escalation Events					X
Error Events	X	X			X
Cancel Events		X			
Compensation Events		X			
Conditional Events					
Link Events		X			X
Signal Events		X			X
Multiple Events					
Terminate Events		X	X	X	X
Boundary Events		X[2]		X[3]	X

[1] Does not support receive tasks after event-based gateways.
[2] Only supports interrupting boundary events on tasks, not subprocesses.
[3] Only supports message and timer events.

[Cam23c] Camunda Services GmbH. Bpmn-js Token Simulation. https://github.com/bpmn-io/bpmn-js-token-simulation, October 2023.

[Cam23d] Camunda Services GmbH. Bpmn-moddle. https://github.com/bpmn-io/bpmn-moddle, October 2023.

[Cam23e] Camunda Services GmbH. Bpmnlint. https://github.com/bpmn-io/bpmnlint, October 2023.

[Cam23f] Camunda Services GmbH. Camunda BPMN model API. https://github.com/camunda/camunda-bpm-platform/tree/master/model-api/bpmn-model, October 2023.

[CFP+21] Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re, Francesco Tiezzi, and Andrea Vandin. A formal approach for the analysis of BPMN collaboration models. *Journal of Systems and Software*, 180:111007, October 2021. doi:10.1016/j.jss.2021.111007.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8.

[CMR+22] Flavio Corradini, Chiara Muzi, Barbara Re, Lorenzo Rossi, and Francesco Tiezzi. Formalising and animating multiple instances in BPMN collaborations. *Information Systems*, 103:101459, January 2022. doi:10.1016/j.is.2019.101459.

[CMRT18] Flavio Corradini, Chiara Muzi, Barbara Re, and Francesco Tiezzi. A Classification of BPMN Collaborations based on Safeness and Soundness Notions. *Electronic Proceedings in Theoretical Computer Science*, 276:37–52, August 2018. doi:10.4204/EPTCS.276.5.

[DDO08] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294, November 2008. doi:10.1016/j.infsof.2008.02.006.

[DS17] Francisco Durán and Gwen Salaün. Verifying Timed BPMN Processes Using Maude. In Jean-Marie Jacquet and Mieke Massink, editors, *Coordination Models and Languages*, volume 10319, pages 219–236. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-59746-1_12.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach.*, pages 247–312. World Scientific, February 1997. doi:10.1142/9789812384720_0004.

[El-15] Nissreen A. S. El-Saber. *CMMI-CM Compliance Checking of Formal BPMN Models Using Maude*. PhD thesis, University of Leicester, January 2015.

[FFK+11] Dirk Fahland, Cédric Favre, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data & Knowledge Engineering*, 70(5):448–466, May 2011. doi:10.1016/j.datak.2011.01.004.

[FR19] Jakob Freund and Bernd Rücker. *Real-Life BPMN: Using BPMN and DMN to Analyze, Improve, and Automate Processes in Your Company*. Camunda, Berlin, 4th edition, 2019.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[HA02] Arthur Hofstede and Wil Aalst. Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In *Proceedings of Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*, volume 560, August 2002.

[HBP+22] Sara Houhou, Souheib Baarir, Pascal Poizat, Philippe Quéinnec, and Laïd Kahloul. A First-Order Logic verification framework for communication-parametric and time-aware BPMN collaborations. *Information Systems*, 104:101765, February 2022. doi:10.1016/j.is.2021.101765.

[HBPQ19] Sara Houhou, Souheib Baarir, Pascal Poizat, and Philippe Quéinnec. A First-Order Logic Semantics for Communication-Parametric BPMN Collaborations. In Thomas Hildebrandt, Boudewijn F. van Dongen, Maximilian Röglinger, and Jan Mendling, editors, *Business Process Management*, pages 52–68, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-26619-6_6.

[HT20] Reiko Heckel and Gabriele Taentzer. *Graph Transformation for Software Engineers: With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-43916-3.

[KKR+23] Tim Kräuter, Harald König, Adrian Rutle, Yngve Lamo, and Patrick Stünkel. Behavioral consistency in multi-modeling. *The Journal of Object Technology*, 22(2):2:1, 2023. doi:10.5381/jot.2023.22.2.a9.

[KR06] Harmen Kastenberg and Arend Rensink. Model Checking Dynamic States in GROOVE. In Antti Valmari, editor, *Model Checking Software*, volume 3925, pages 299–305. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi:10.1007/11691617_19.

[Krä23] Tim Kräuter. LMCS-2024: Artifacts. Zenodo, October 2023. doi:10.5281/ZENODO.10018457.

[KRKL23] Tim Kräuter, Adrian Rutle, Harald König, and Yngve Lamo. Formalization and Analysis of BPMN Using Graph Transformation Systems. In Maribel Fernández and Christopher M. Poskitt, editors, *Graph Transformation*, volume 13961, pages 204–222. Springer Nature Switzerland, Cham, 2023. doi:10.1007/978-3-031-36709-0_11.

[MDL+14] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A Framework for Generating Domain-Specific Property Languages. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, volume 8706, pages 1–20. Springer International Publishing, Cham, 2014. doi:10.1007/978-3-319-11245-9_1.

[Men09] Jan Mendling. Empirical Studies in Process Model Verification. In Kurt Jensen and Wil M. P. Van Der Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II*, volume 5460, pages 208–224. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-00899-3_12.

[Obj13] Object Management Group. Business Process Model and Notation (BPMN), Version 2.0.2. https://www.omg.org/spec/BPMN/, December 2013.

[Obj17] Object Management Group. Unified Modeling Language, Version 2.5.1. https://www.omg.org/spec/UML, December 2017.

[Pet23] David Peter. Hyperfine. https://github.com/sharkdp/hyperfine/, October 2023.

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062, pages 479–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-25959-6_40.

[Ren06] Arend Rensink. Nested Quantification in Graph Transformation Rules. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations*, Lecture Notes in Computer Science, pages 1–13, Berlin, Heidelberg, 2006. Springer. doi:10.1007/11841883_1.

[Ren08] Arend Rensink. Explicit state model checking for graph grammars. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, pages 114–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-68679-8.

[Ren17] Arend Rensink. How Much Are Your Geraniums? Taking Graph Conditions Beyond First Order. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd*, volume 10500, pages 191–213. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-68270-9_10.

[Rüc21] Bernd Rücker. *Practical Process Automation: Orchestration and Integration in Microservices and Cloud Native Architectures*. O'Reilly, Beijing Boston Farnham Sebastopol Tokyo, first edition edition, 2021.

[SSHK15] Martina Seidl, Marion Scholz, Christian Huemer, and Gerti Kappel. *UML @ Classroom*. Undergraduate Topics in Computer Science. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-12742-2.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562, pages 18–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-02674-4_3.

[VGD13] Pieter Van Gorp and Remco Dijkman. A visual token-based formalization of BPMN 2.0 based on in-place transformations. *Information and Software Technology*, 55(2):365–394, February 2013. doi:10.1016/j.infsof.2012.08.014.

PAPER D

TOWARDS THE COORDINATION AND VERIFICATION OF HETEROGENEOUS SYSTEMS WITH DATA AND TIME

Tim Kräuter, Adrian Rutle, Yngve Lamo, Harald König, Francisco Durán

Submitted to the ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS), 2025

Towards the Coordination and Verification of Heterogeneous Systems with Data and Time

Tim Kräuter
Adrian Rutle
Yngve Lamo
tkra@hvl.no, aru@hvl.no, yla@hvl.no
Western Norway University of Applied Sciences
Bergen, Norway

Harald König
harald.koenig@fhdw.de
University of Applied Sciences, FHDW
Hanover, Germany
Western Norway University of Applied Sciences
Bergen, Norway

Francisco Durán
fdm@uma.es
University of Málaga
Málaga, Spain

***Abstract*—Modern software systems are often realized by coordinating multiple heterogeneous parts, each responsible for specific tasks. These parts must work together seamlessly to satisfy the overall system requirements. To verify such complex systems, we have developed a *non-intrusive* coordination framework capable of performing *formal analysis* of *heterogeneous* parts that *exchange data* and include *real-time* capabilities. The framework utilizes a linguistic extension—which is implemented as a central broker and a domain-specific language—for the integration of heterogeneous languages and coordination of parts. Moreover, abstract rule templates are reified as language adapters for non-intrusive communications with the broker. The framework is implemented using rewriting logic (Maude), and its applicability is demonstrated by verifying certain correctness properties of a heterogeneous road-rail crossing system.**

***Index Terms*—Coordination, Verification, Heterogeneous Systems, Data, Time**

I. Introduction

Software systems are integral to nearly every aspect of modern life. To meet their ever-growing requirements, these systems are often made by coordinating separate parts, which are implemented using the most appropriate tools for each of them. Consequently, modern software systems consist of multiple heterogeneous parts, each responsible for specific tasks that must work together to fulfill the overall system requirements. This rising complexity has made the development of software systems more challenging, while their ubiquitous nature amplifies their need for safety, reliability, and correctness [1].

To verify properties—for example, safety and reliability—of systems consisting of *heterogeneous* parts that *exchange data* and can involve *real-time* capabilities, we have developed a coordination framework designed for *formal analysis*. Here, heterogeneous means that each system part may be specified using a different *real-time behavioral* modeling language. We refer to behavioral modeling languages as languages that specify the dynamic aspects of a system, such as statecharts, Petri Nets, and process models. In contrast, structural modeling languages, such as UML class diagrams or entity-relationship models, focus on data representation. Behavioral languages can include *real-time* features, where actions are influenced by the passage of time. This means one can define when an action can be executed and how long it takes to complete. For instance, an action may occur periodically or after a specific time while in a particular state. In addition, *data exchange* is a unique feature currently missing in other coordination frameworks [2]–[4].

Our coordination framework enables integration while upholding *separation* of the different parts of the system by using a mediator, referred to as *broker*. Embracing separation makes the framework suitable for a wide range of scenarios and facilitates the addition of new modeling languages.

The main idea behind our approach is based on three key ingredients, namely *language integration*, *coordination*, and *verification*, which are depicted in Figure 1 and explained in more detail in section IV:

Language Integration. Integrating a behavioral language into our framework requires implementing a *language adapter*. The adapter mediates between the behavioral language and the common language of the *broker*. Thus, we can use new languages by defining a *behavioral interface* for the given language and then building an adapter that uses it to communicate with the broker. The interface definition remains non-intrusive, leveraging a *linguistic extension* for each modeling language. Each adapter involves aligning the data model of the specific language with the broker's *canonical data model* by defining data transformation functions.

Coordination. The system model consists of individual models conforming to the previously integrated behavioral languages. These models are then instantiated and coordinated to create a *system configuration*. Since there might be multiple instances of a model, coordination is defined at the instance level. Coordination relies on the *channels* of each instance, i.e., specific entities where data can be read from or written to as identified in the behavioral interface of the corresponding language. In our framework, we provide a domain-specific language (DSL) to define Input/Output (I/O) bindings, i.e., asynchronous data exchange between the channels.

Verification. Once a system configuration is available, global properties of interest, i.e., *system properties*, can be defined and checked. In our framework, the verification takes *real-time features* into account. In this work, we propose using reachability analysis and Linear Temporal Logic (LTL) model

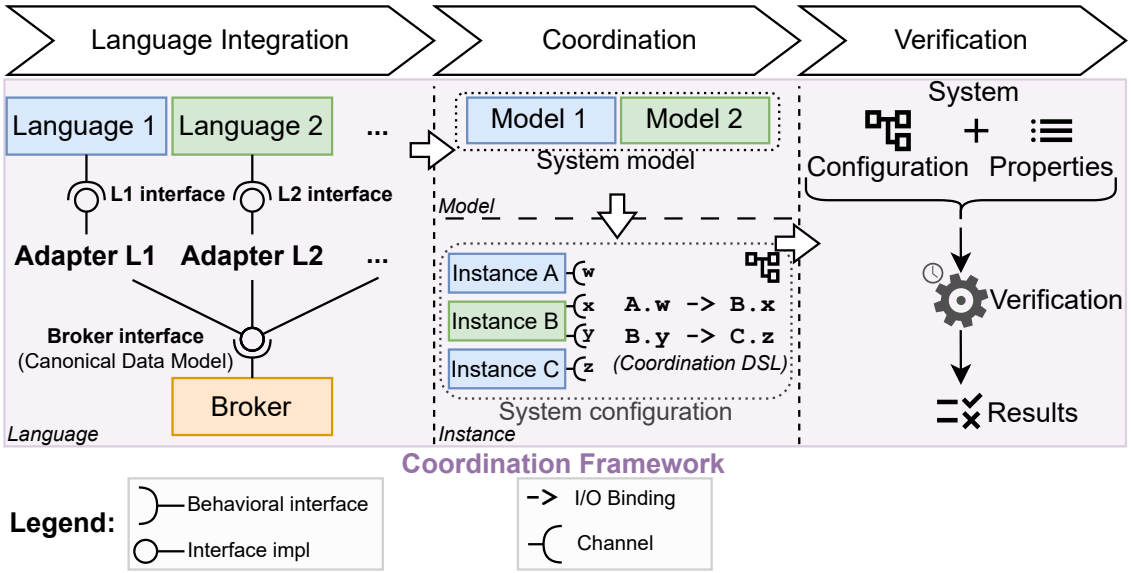


Fig. 1. Overview of the Approach

checking. The result of the analysis is either a confirmation that the property holds or the identification of a counterexample.

The coordination framework provides an architecture built on general concepts like language adapters, I/O bindings, and a canonical data model. To function, this framework must be instantiated with a formalism that can express these concepts and support verification involving data exchange and real-time features. Once instantiated, it allows the analysis of heterogeneous systems by applying the methods of the chosen formalism. As explained in section V, our framework has been developed using rewriting logic and is implemented in Maude [5]. Maude allows us to provide semantics to different modeling languages and to formally check properties using its reachability analysis tool and model checker for real-time systems [6], [7]. Our framework may also be implemented using other formalisms provided that the following requirements are satisfied:

(1) coping with the (operational) semantics of the modeling languages used to specify the different parts,
(2) supporting data exchange between the used modeling languages,
(3) modeling time and its passage affecting the dynamics of the global system, and
(4) providing analysis capabilities for the global system.

The main contributions are summarized as follows. **(i)** Our coordination framework provides a methodology to define coordination between multiple *heterogeneous* behavioral models in a *non-intrusive* manner, allowing for formal *analysis* of the resulting *global system*. **(ii)** The framework enables *data exchange* between different languages, a capability missing in previous coordination frameworks [2]–[4]. **(iii)** Moreover, the framework supports *real-time functionality* while maintaining a clear separation between the integrated languages.

The remainder of the paper is structured as follows. First, we provide the necessary background for our contribution in section II. Then, we introduce a use case to motivate our framework in section III. Afterward, we describe the coordination framework in section IV before instantiating it using rewriting logic (Maude) to obtain a concrete implementation, which we apply to the use case in section V. Finally, we conclude in section VI.

II. Background

Multiple research fields have investigated the coordination, verification, or simulation of software or even the combination of software and hardware systems. In this section, we provide a brief overview of the three related research areas: *coordination languages*, *architecture description languages*, and *co-simulation*.

Coordination languages can be broadly categorized into two families. The Linda approach enables communication between software programs by providing a global shared memory, commonly called a tuple space, along with operations to read and write to this shared space [8]. The coordination languages of the Linda approach are usually embedded in a general-purpose programming language [9]–[11]. In contrast to *data-driven* coordination in the Linda approach, a family of *control-driven* languages emerged [9] including for example Manifold [12], [13] and REO [14]. These languages define interfaces, i.e., ports for each entity, which can be connected to facilitate communication [15].

The goal of **Architecture Description Languages** (ADLs) is to describe the overall structure of a system by focusing on high-level system components and their connections [16]–[18]. One uses an ADL to define which *components* and *connectors* exist before combining them in a concrete *architectural*

configuration to describe a system's structure [17]. Often, ADLs support verification, such as checking if an architectural configuration is free of deadlocks and starvation, by employing process algebras such as CCS, CSP, and π-calculus [19].

Co-Simulation approaches facilitate the information exchange between multiple simulations running concurrently, ensuring temporal relationships. A simulation is based on a simulation unit with black-box behavior but a predefined simulation interface of inputs and outputs. Typically, an *orchestrator* uses these interfaces to transfer data between simulations and dictate the passage of simulated time [20]. Co-simulation approaches are classified into *Discrete Event* (DE), *Continuous Time* (CT), or *Hybrid* when they combine both DE and CT elements. At present, the two primary standards for co-simulation are the *Functional Mock-up Interface* (FMI) [21] and the *High Level Architecture* (HLA) [22].

The methodology in our approach outlines a **coordination framework**. The unique characteristic of coordination frameworks is that they embrace *model heterogeneity* by operating on the language level [4], i.e., not only providing one formalism or modeling language that must be used exclusively, such as coordination languages and ADLs. We do not consider co-simulation approaches as coordination frameworks since they compose executable programs (simulation units) that embrace heterogeneity at the execution level, not the model level. Coordination frameworks described in the literature include Ptolemy [23], BCOoL [3], [4], and the approach in [2].

However, because Ptolemy is focused on execution, it relies on a general-purpose programming language, which limits the ability to verify the coordinated system formally. The other coordination frameworks are built on sound formalisms. Still, they lack support for data exchange during coordination, which is essential and a natural form of communication in modern software systems. Both approaches identify data exchange as a key focus for future work [2]–[4]. Our coordination framework enables formal analysis of heterogeneous systems, incorporating *data exchange* and *real-time capabilities* while remaining *non-intrusive* and upholding separation.

Our contribution focuses on software systems that can include real-time characteristics. We exclude CT or hybrid systems, typically involving hardware components modeled by differential equations. By applying the principles of Real-Time Maude, we can model continuous-time software systems that can be "discretized" using time-sampling strategies available in Maude [7].

III. Use Case

In this section, we present a use case to illustrate the motivation behind our coordination framework. We begin by introducing the use case, followed by explaining its specification. Finally, we discuss several properties that must be verified for the specification.

A. Description

The use case is a traffic management system for a *level crossing* (also called road-rail crossing). Level crossings account for over 400 accidents in the European Union every year [24]. The challenge with level crossings is that trains have a much larger mass relative to their braking capability and, thus, a far longer braking distance than road vehicles. As a result, trains typically do not stop at a level crossing and depend on vehicles and pedestrians to clear the tracks beforehand. *Passive* level crossings are only equipped with a traffic sign and are associated with a higher number of accidents compared to *active* crossings [24].

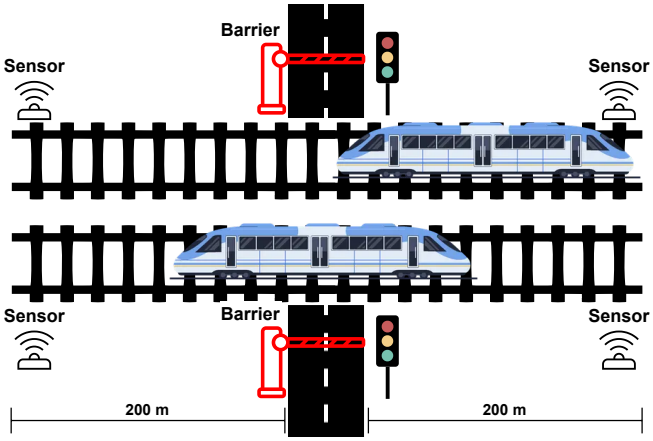


Fig. 2. Use case: Active Level Crossing (two train tracks)

This use case aims to design an *active* level crossing system that warns cars about incoming trains using traffic lights and barriers. Figure 2 provides an overview of the scenario with two train tracks. The case study was kept as simple as possible (trains do not communicate with the system) to illustrate the key concepts of our approach. In this case, the primary objective is to ensure that the proposed specification enables cars and trains to *safely pass* through the crossing.

B. Specification

The case study is realized by coordinating three system parts specified using two different modeling languages, namely Colored Petri Nets (CPN) [25] and statecharts. This corresponds to the model level in the coordination step in Figure 1. We will explain the system parts as sketched in the diagram in Figure 3. By default, all distances will be expressed in meters, time in seconds, and speed in meters per second.

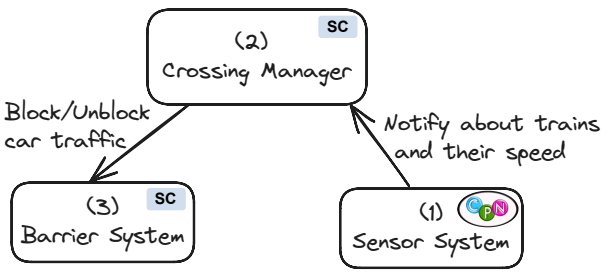


Fig. 3. System parts in the use case

a) The sensor system: shown in Figure 2 detects the speed of incoming and outgoing trains. In general, trains can arrive in any order and at different speeds, and there may be level crossings with either one or multiple tracks. To model train behavior flexibly, the model was developed using CPN as shown in Figure 4. It consists of two parts: a train simulation (blue) and the sensor system (green).

In a CPN, places are represented with ellipses and transitions with rectangles. Each token carries a data value that belongs to a given type, indicated at the bottom right of a place (`TIMED_REAL` means real value and a timestamp). In the state in Figure 4, the place `New train can approach` (upper left corner) has two tokens with values 25 and 40 (train speed). Transitions can have a time inscription displayed at the top right, increasing the timestamp of produced tokens. For example, when a token moves from `New train waiting to approach` to `New train can approach`, its timestamp increases by 10. The arc labels are expressions that reference the values of tokens as they traverse the arcs. In the given use case, they ensure that the measured speed is transmitted correctly. Finally, the places with double lines (the two on the right side of the figure) are *port places*, representing the interface through which the model communicates with its surroundings. The blue tag indicates the place's type; in this case, both are output ports.

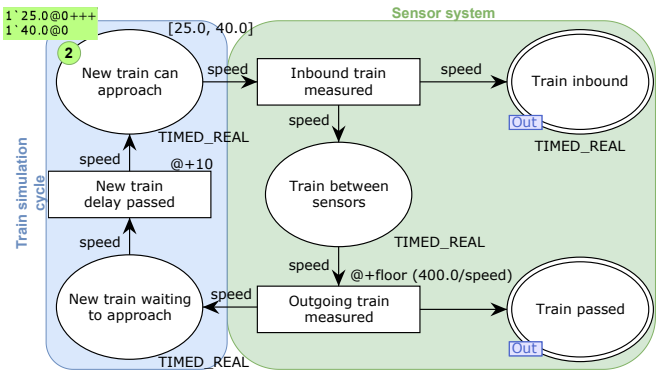


Fig. 4. Train simulation & Sensor system (CPN)

The number of initial tokens in the CPN model determines the number of tracks being modeled, assuming that there is only one train at a constant speed on each track at any given time (25 m/s and 40 m/s in Figure 4). The two main transitions are `Inbound train measured` and `Outgoing train measured`, which refer to the sensors detecting a train as it enters or leaves the 400-meter corridor. Sensors are assumed to be placed on the tracks 200 meters on each side of the barriers. These transitions produce tokens in `Train inbound` and `Train passed`, which are the above-mentioned interfaces. After a short cooldown period (see `New train delay passed`), new trains can start approaching the crossing, resulting in an endless cycle of trains passing over time; see the blue train simulation part on the left of Figure 4. Adjusting either the number of tokens or the train simulation within the CPN model can change the simulated behavior.

b) The crossing manager: monitors the information from the sensors to detect whether any trains are within the 400-meter zone. Based on this, it manages car traffic by sending signals to the barrier system. The crossing manager is specified as a statechart in Figure 5. Transitions are defined as usual using three optional components in the following format: *trigger [guard] / effect*. Here, the *trigger* can be either time-based or event-driven, the *guard* is a boolean expression, and the *effect* consists of a series of statements (separated by semicolons) that raise an event or modify the statechart's variables.

The system starts in the state `No Trains` and transitions to the state `Trains inbound` when a `trainInbound` event is received. It also tracks the number of trains in the corridor using the `trains` counter. Additionally, it calculates how many seconds it will take for the trains to reach the level crossing, factoring in a safety margin: $closeIn = 200/trainSpeed - safetyBuffer$.

In the `Trains inbound` state, the *closeIn* variable decreases every second until it reaches 0. At that point, the system transitions to the `Trains are passing` state, triggering the `blockCarTraffic` event. However, additional trains may arrive while still in the `Trains inbound` state and could reach the level crossing before the previous train. In such cases, the `closeIn` variable must be updated accordingly.

In the `Trains are passing` state, the `trains` variable is either decreased when trains pass or increased when trains approach. When the variable reaches 0, the `unblockCarTraffic` event is triggered, and the system transitions to the `No Trains` state, allowing the process to restart. The events `blockCarTraffic` and `unblockCarTraffic` represent the data flowing from the crossing manager to the barrier system in Figure 3.

c) The barrier system: manages car traffic by controlling the barriers and corresponding traffic lights. The system is specified as a statechart in Figure 6. It depends on two *external* events, `openBarrier` and `closeBarrier` (incoming data in Figure 3), which trigger an intermediate state lasting two seconds, representing the time required for the barrier to move.

All three systems—the sensor system, barrier system, and crossing manager—must work together to ensure safe and efficient traffic management at the level crossing. This is more complex than it appears in Figure 3, as CPN models typically do not interact natively with statecharts. Moreover, even the event names in the statecharts do not align, such as `blockCarTraffic` and `closeBarrier`. To address this challenge, the modeling languages must be integrated and proper coordination must be established, as illustrated in Figure 1. In the next section, we describe the intended verification to ensure correct coordination.

C. Verification

We aim to analyze the proposed specification from the previous section to identify potential issues early on. To ensure the system behaves as intended, we focus on verifying *two*

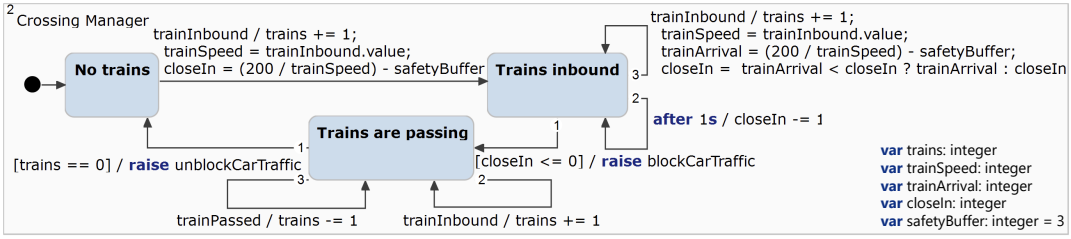


Fig. 5. Crossing Manager (statechart)

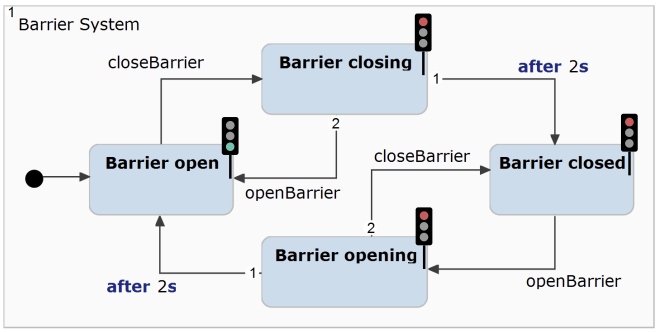


Fig. 6. Barrier System (statechart)

key properties. These properties are expressed using Linear Temporal Logic (LTL), employing state-based model checking as our primary verification method. Nonetheless, in general, other analysis approaches may also be worth exploring.

First, we must ensure that the system does not create unsafe situations, such as allowing trains to pass while the barriers remain open. This safety property can be expressed in LTL with Property (1), which expresses that a train never passes when the barriers are open.

$$\square \neg (\textit{Barriers-open} \wedge \textit{Train-passing}) \tag{1}$$

Second, we must ensure that when the barrier closes, it will eventually reopen so that car traffic is not indefinitely halted and Property (1) is trivially upheld. This requirement is captured by Property (2), which represents a specific case of the *response* pattern [26]. This pattern dictates that once the system enters the first state (*Barriers-closed*), it must eventually transition to the second state (*Barriers-open*).

$$\square (\textit{Barriers-closed} \rightarrow \Diamond \textit{Barriers-open}) \tag{2}$$

Verifying these system properties is challenging for several reasons, and as a result, none of the approaches, even coordination frameworks discussed in section II, currently support this. First, the system consists of three parts which are designed using *heterogeneous modeling languages* (statecharts and CPN in our case study). Second, *data is exchanged* between these parts (train speed in our example), which affects how the system behaves (waiting times). Third, the system's behavior is influenced by *global time*, which needs to remain consistent across the different modeling languages. Finally, we do *not* want to be *intrusive*, meaning imposing model changes or requiring a specific modeling language.

IV. Coordination Framework

The driving design goals for our coordination framework are *non-intrusiveness* and upholding the *separation* of participating models and instances. Furthermore, we want the integration of additional languages to require minimal effort, for which separation across multiple key aspects is needed. Separation, or loose coupling, is essential in software architecture because it simplifies making changes to the system by minimizing interdependencies [27], [28].

We focus on the separation of the following key aspects: *languages*, *models*, *time*, and *data*. Firstly, each system part should be flexible in choosing the most suitable modeling language, provided the underlying formalism can support the language. Secondly, every part of the system, i.e., each model, must be able to evolve independently. Thirdly, each model should be able to use its native real-time features while time progresses uniformly during execution for all model instances, never skipping real-time events. Finally, each model should be able to use its own data model yet still be able to exchange data with other system parts. In the following sections, we detail our approach by explaining the key aspects highlighted in Figure 1 and explain how it meets the design goals of non-intrusiveness and separation.

Language integration, coordination, and verification are distinct tasks, each typically handled by different roles. For instance, a language engineer is responsible for language integration, a system architect manages coordination, and the quality assurance team oversees the verification process.

A. Language Integration

To facilitate coordination while upholding separation, we introduce a central *broker*, which acts as a mediator. We define the metamodel for the broker on the left in Figure 7. As shown in its metamodel, the broker is just a collection of bindings, which, as we will see below, are handled independently, without interactions between them. To integrate heterogeneous languages, we use a *linguistic extension* [29], [30]. The purple part of the broker metamodel defines the linguistic extension, which is used to augment behavioral languages non-intrusively. It introduces the concept of a `Channel`, which refers to any location where data can either be read from or written to. Linguistic extension is a widely used technique

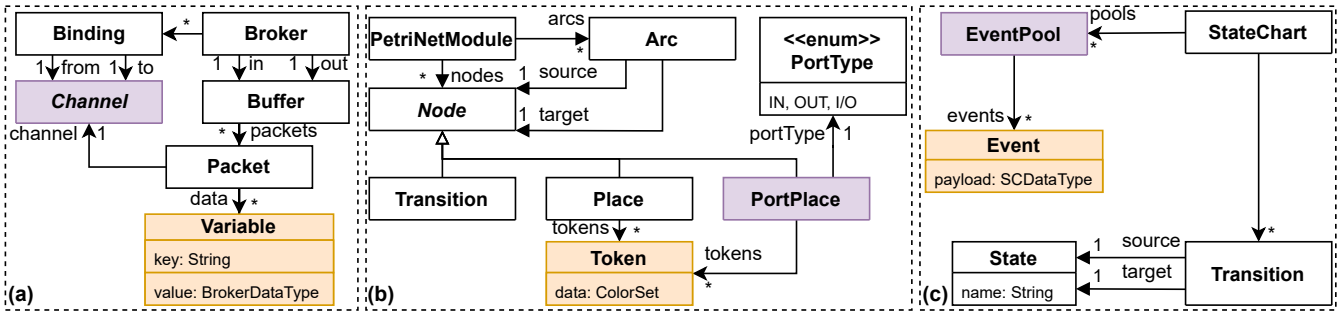


Fig. 7. Broker Metamodel **(a)** and CPN/Statechart Metamodel excerpts **(b)/(c)**

in the multi-level modeling domain, where new concepts can be added a posteriori to existing metamodels at any level of a metamodeling hierarchy. In our case, we use the broker metamodel to define the concept (`Channel`), which can be used in the participating metamodels. The channels are used to identify the behavioral interface for the participating languages, e.g., `PortPlace` for CPNs and `EventPool` for statecharts are both typed by `Channel` (see the purple colored elements in Figure 7).

The broker metamodel also introduces `Packet` containing data in a *canonical data model* [27] to enable data exchange among coordinated models while upholding separation. The *canonical data model* serves as a common intermediate to enable conversion between the varying data models of the used languages. Data-related concepts are shown in orange in Figure 7. A packet is contained in an input or output `Buffer`, which holds the packet until it is transmitted to the recipient. A `Packet` contains `data`, i.e., `Variables`, which are key-value pairs and references the `Channel` from which it was ingested or to which it is about to be delivered.

The broker has a set of bindings that connect `Channels`. As mentioned earlier, a `Channel` is a connection point in a given language that the broker can use to read or write. If a binding is defined between two channels, the broker will facilitate coordination, i.e., data exchange. The broker must know how to interface with each behavioral language to adapt between the heterogeneous models while maintaining language separation. By augmenting a specific element in a metamodel with the concept of `Channel`, made possible by the linguistic extension, one can define the behavioral interface for that metamodel [4].

Language integration (see Figure 1) can be further subdivided into two steps. First, one must identify the behavioral interface for the new language, i.e., define which parts of the metamodel are exposed by extending `Channel`. As a result, we obtain an *augmented* metamodel, such as the metamodel for CPNs in Figure 7 (b), where `PortPlace` is identified as a `Channel` (purple coloring). Since port places constitute the *interface* through which a CPN exchanges tokens with its environments, we utilize port places as the behavioral interface for our framework. More broadly, one can examine how different model instances within a language interact to determine its behavioral interface. For example, in the case of statecharts, instances communicate using events dispatched through *event pools* [31]. Therefore, event pools serve as the behavioral interface for statecharts, i.e., an `EventPool` is a `Channel`, as highlighted by the purple color in the statechart metamodel in Figure 7 **(c)**.

Second, one utilizes the augmented metamodel to create a *language adapter*, inspired by the adapter [32] or channel adapter [27] patterns. This language adapter facilitates the integration of heterogeneous languages by mediating between the specific language and the broker.

A key part of each language adapter is the translation between different data representations used by different languages. To enable seamless data exchange, the broker employs a *canonical data model* as a standardized format. Each adapter is responsible for implementing two functions that handle the data translation process: (i) The function $toBroker$ translates data from the specific data model to the broker's canonical data model, and (ii) the function $fromBroker$ translates data from the broker's canonical data model back to the specific data model. These two functions should be inverse functions of one another. Using a *canonical data model* addresses the problem where normally the number of translators needed to enable communication between each pair of participants increases quadratically with the number of participants [27]. The canonical data model, i.e., packets consisting of key-value variable pairs, is kept abstract but sufficient to showcase the framework's data exchange capabilities.

In addition to *syntactic mapping* between different data models, as discussed so far, schema matching, i.e., *semantic mapping* [33], is also vital to enable system interoperability. Semantic mapping is needed since systems can represent the same information structurally differently. One system might use different field names or even split information into two fields compared to another system. Semantic mapping allows reconciling these differences when data is transferred between the systems. In our coordination framework, semantic mapping can be included in the translation functions and customized for each binding or even generated from relations between the data models, often described graphically in data integration tools. However, we do not investigate this further in this paper.

Language adapters can be defined as follows. A language adapter reads from the channels of model instances to create an intermediate packet, provided that a suitable binding source is defined (**ingest**). Additionally, for binding targets, an adapter writes to the channels and removes the previously created

packets (**deliver**). These two actions can be described by rule templates [34], [35], using a Henshin-like notation [36], as shown in Figure 8, which can then be specialized for different languages to build a concrete language adapter. The rules can be interpreted as follows: the black elements represent objects that will remain unchanged during the transformation. The red elements will be deleted during the transformation process, while the green elements will be added. The dotted elements need to be filled in for a specific language adapter.

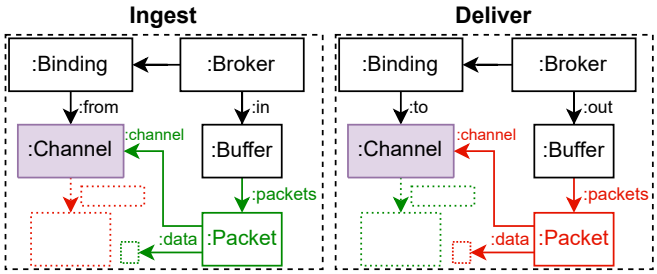


Fig. 8. Language Adapter Rule Templates **Ingest** and **Deliver**

For instance, Figure 9 shows adapter rules for CPN and statecharts, as well as the generic move rule for the broker. The **ingest** rule for CPN is displayed on the left, and the **deliver** rule for statecharts is displayed on the right. Both **ingest** and **deliver** rules require converting between different data models, for which we employ the previously defined translation functions. Specifically, the functions $toBroker_{cpn}$ and $fromBroker_{sc}$ are used for translation. Consequently, an event generated from a CPN token containing data x will have the payload $fromBroker_{sc}(toBroker_{cpn}(x))$.

The **move** rule, depicted in the middle, demonstrates how the broker transfers packets from its input buffer to its output buffer, effectively isolating the language adapters. Following the rules from left to right shows how a token is converted into an event, including associated data.

In summary, introducing a broker as a mediator achieves the *separation* of languages, models, and data. Due to the concept of language adapters and the canonical data model, we keep languages, models, and data as separated as possible. Additionally, identifying a behavioral interface for each language through a *linguistic extension* ensures *non-intrusiveness*, as models do not have to be changed for coordination. Notably, incorporating a new language into the framework involves a **one-time integration effort**. We will now detail how time separation is achieved, and we guarantee consistent passage of time across heterogeneous formalisms.

Time. Many modeling languages include the real-time functionality to specify, for example, that actions take a certain amount of time, are periodic, or can only happen after a specific amount of time. Each language will provide its own real-time elements, and the behavioral semantics of these elements will be given as part of the semantics of the corresponding languages. For example, time inscriptions for CPNs [25] and real-time triggers in statecharts are employed in the use case. However, to guarantee consistent time treatment, individual model instances that are part of the coordinated system cannot perform timed actions as they please since all actions must be consistent with time elapse.

Individual model instances cannot progress time independently. There must be a model of time that is followed by all language definitions participating in the framework. As we will discuss in section V, our framework has been implemented using rewriting logic (concretely the Maude system). Therefore, we expect that the real-time features of the different language definitions adhere to the principles of Real-Time Maude [7]. We assume a *global clock*, which will advance and then synchronize with the clocks of the individual instances. The global clock is only changed by the *tick* rule, meaning all other rules can be seen as *instantaneous*. For instance, to model some duration, we will have a start action and an end action, both instantaneous. All time-related actions will have a timer or a scheduled time that will be used to fire them. Instead of keeping a centralized scheduled-time sorted list of actions, we will assume functions calculating the time to the first action, or the maximum amount of time that may elapse without an action happening (*mte*), and another one applying the pass of time (*delta)*. Given these functions, the passage of time can then be modeled by a unique tick rule that calculates the maximum time elapse without an executable action (*mte*) and applies the *delta* function to the entire system. Under these assumptions, the control of time can be easily split between the different parts of the system. The *mte* function will be defined as the minimum of its results for each part of the system. Then, the delta function on the global system will simply consist of applying it to each of its parts. The definitions of the *mte* and *delta* functions for each modeling language will be part of the definition of each language adapter. Time-related actions will then be fired when a timer reaches time zero or a clock matches a local or global clock. As already said, this approach to real-time is based on Real-Time Maude [7], which is similar to the models used for DE-based co-simulation methods [20], where an orchestrator controls global time.

B. Coordination

The second step in our approach (see Figure 1) is the *coordination* of the system parts (models). This step can be further subdivided into the following three steps. First, the *system model* is created, which consists of individual models conforming to the previously integrated behavioral modeling languages.

Second, one instantiates the system model by instantiating its respective individual models. In the use case, each model (the barrier system, sensor system, and crossing manager) has only one instance, although multiple instances of the same model are possible.

The third step is based upon the augmented metamodel obtained from the language integration step (see Figure 1). The linguistic extension defines each language's behavioral interface, specifying which channels can be linked via bindings. We utilize a textual DSL to connect channels from different modeling languages uniformly. For example, for the

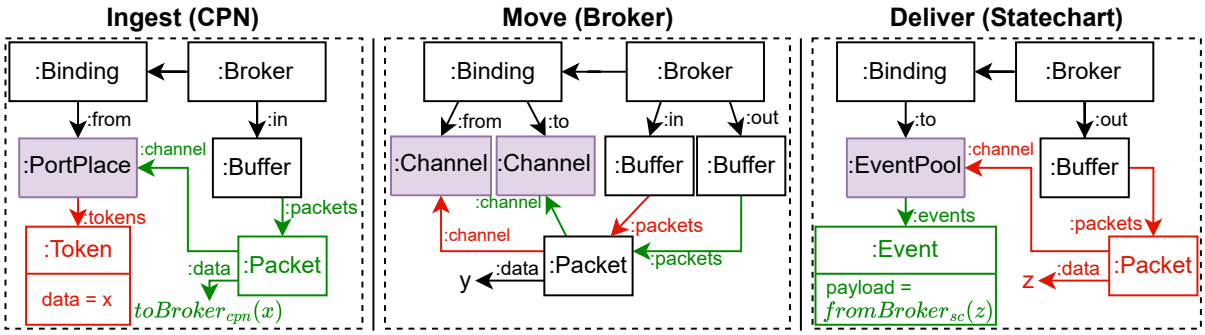


Fig. 9. Broker and Language Adapter Rules

```
Sensor.Train_inbound  ->  Crossing.trainInbound,
Sensor.Train_passed  ->  Crossing.trainPassed,
Crossing.unblockCarTraffic  ->  Barrier.openBarrier,
Crossing.blockCarTraffic  ->  Barrier.closeBarrier
```

Listing 1. Bindings for the use case

use case, we define the bindings as shown in Listing 1. Each line defines one binding, following the structure *bindingSource* `->` *bindingTarget* in our DSL. Finally, the model instances, along with their corresponding bindings, constitute the *system configuration*, which is utilized for verification.

C. Verification

Verification in our approach (see (iii) in Figure 1) is further subdivided into specification and verification of system properties.

First, one specifies the *system properties* that must be verified based on the *system configuration*. To specify the properties of the system's global state, it is necessary to first define the *state structure* for each participating modeling language, as explained, for example, in [2], [37]. The state structure, which is also needed for execution, leads to atomic propositions for each language that can be combined to express temporal logic properties for the global system, e.g., the above formulas (1) and (2). The global state is a tuple of the local states of each model based on the system configuration. For example, the initial state in the use case is a triple consisting of the statechart start states and the two tokens in the place `Train prepared to approach`.

Second, we use the system properties and the system configuration from the previous steps to perform *automatic* verification, for example, LTL model checking [1], where the states are the tuples mentioned in the previous step. This generates verification results, including counterexamples, if any properties are unsatisfied.

V. Implementing & Verifying the Use Case

We implemented the framework using rewriting logic and its implementation in Maude [5] as the foundational formalism. The full implementation in [38] includes the use case in addition to definitions of Labeled Transition Systems (LTS) [1] and Business Process Model and Notation (BPMN) [39]—in these cases without time or data—as well as several examples using different combinations of the available languages. Maude meets the formalism requirements outlined in the introduction: it can handle a variety of operational semantics (1) [40], [41], supports the modeling of data and data exchange (2) [5], enables time modeling through Real-Time Maude [7] (3), and offers built-in verification capabilities (4). Verification in Maude includes techniques such as reachability analysis, explicit-state LTL model checking [6], and time-bounded LTL model checking [7]. We now present key implementation details from each step of our approach (see Figure 1), using the use case as a practical example.

In Maude, object-oriented systems may be specified using the usual elements in object-oriented languages, which allows us to specify the models in Figure 7. For instance, there is a class Broker with `in` and `out` attributes of type `Packets`, representing buffers of packets, and an attribute `bindings` of channel bindings.

```
class Broker | in : Packets,
               out : Packets,
               bindings : Set{Binding} .
```

A. Language Integration

Each formalism integrated into the framework will have its own representation. For example, following the descriptions in Figure 7, CPNs will be represented by classes `CPN` and `CPNInstance`. An object of class `CPN` has a set of places, a set of transitions, and a set of arcs. In some CPN objects, a subset of the places will be port places. An object of `CPNInstance` is associated with its CPN model (`cpn`) and has a marking (multiset of tokens).

```
class CPN | places : Set{CPNPlace},
            transitions : Set{CPNTransition},
            arcs : Set{CPNArc} .
class CPNInstance | cpn : Oid,
                    marking : CPNTokens .
```

Given appropriate class declarations, objects are then represented with syntax `< O : C | Atts >`, with `O` an object identifier, `C` a class identifier, and `Atts` a comma-separated set of attribute-value pairs with the form `a : v`, with `a` the attribute's name and `v` its value.

The behavior of these objects is then specified by rewriting rules. For the language integration, the specification includes rules **ingest** and **deliver** similar to those in Figure 9. For example, the language adapter rule **ingest** for CPNs is implemented as shown in Listing 2. Objects of classes `CPNId` of class `CPN`, `IId` of class `CPNInstance`, and `Br` of class `Broker` are involved in the rules. In it, we can see how the `CPNInstance` object `IId` corresponds to the `CPN` object `CPNId` (`cpn` attribute). We can also see how, if there is a token in the place `PlaceId` in the marking of the CPN instance object, with a binding for `PlaceId`, the token is removed from the CPN instance and added in the `in` buffer of the broker object. Notably, the data transformation to the canonical data model occurs in the operation `cpnToBroker` (lines 11 and 17) and is not fully shown in Listing 2. Furthermore, the rule presented in Listing 2 is a simplified version and does not account for the token's availability at the current global time.

```
1  omod CPN-ADAPTER is
2   inc BROKER .
3   inc CPN-SEM .
4   op cpnToBroker : CPNData -> VariableSet .
5   ...
6   rl [token_to_packet] :
7      < CPNId : CPN |
8        places :
9          (place(PlaceId, InArcs, OutArcs, Type),
10          Places) >
11     < IId : CPNInstance | cpn : CPNId,
12       marking : (token(PlaceId, Data, 0), Marking) >
13     < Br : Broker | in : Packets,
14       bindings : (PlaceId -> ChannelId, Bindings) >
15  => < CPNId : CPN | >
16     < IId : CPNInstance | marking : Marking >
17     < Br : Broker |
18       in : (packet(PlaceId, cpnToBroker(Data)),
19             Packets) > .
20 endom
```

Listing 2. CPN Language Adapter Rule **Ingest** in Maude [38]

A similar Maude rule implements the **deliver** rule, specified in Figure 9. It is important to highlight that each adapter is implemented in its own Maude module, demonstrating how the separation of the languages is reflected at the code level.

To apply our implementation to the use case, we developed a language adapter for both CPNs and statecharts in [38]. These language adapters are built on our broker metamodel (see Figure 7 (a)), and the identification of behavioral interfaces within the CPN and statechart metamodels (see Figure 7 (b)/(c)). Implementing language adapters involves converting the data models of CPN and statecharts into the broker's canonical data model. Additionally, language integration ensures that the semantics of both CPN and statecharts are aligned with the externally provided global clock.

Global Time Elapse. As already discussed in section IV, the model of time is provided by Real-Time Maude [7]. In summary, time passing is realized by a single *tick rule*, which models the time elapse of the global clock based on the *mte* (maximum time elapse) function, and the *delta* function, which applies the time pass to each model instance. The key element in this model of time and the approach followed is that, as already said, the tick rule is the only rule that models the passage of time. All other rules in the specification are instantaneous, meaning they can use time values but not advance time. This approach allows us to keep as many clocks and timers as necessary but with one single global clock that is used to synchronize the others. Moreover, the specification of the different formalisms gets simplified because all we have to do, regarding time, is to provide the appropriate equations defining the behavior of the *mte* and *delta* functions on its instances and time-related features.

B. Coordination

The coordination is straightforward in Maude. We define models and their instances separately, as illustrated in Listing 2, where an instance links back to its model (`cpn` attribute). We define each model from the use case, instantiate it once, and add bindings to obtain the system configuration. To add bindings, we use the broker metamodel from the language integration, which implements the bindings DSL we introduced earlier (see Listing 1).

The Maude rules corresponding to the rules in Figure 9 handle the coordination. The adapter definition identifies the elements used in the specific language for communication and how to handle them. In the case of the CPN, we have seen how the **ingest** rule is in charge of taking a token from a port place (a place for which there is a binding), and placing it into the input buffer of the broker. The **move** rule then moves packets from the input buffer to the output buffer. Finally, the adapter of the formalism on which the target part is specified will handle such a packet. That is, it will take the packet and transform it into the communication element of the specified channel. For example, in our example, the target is specified as a statechart. Thus, the statechart adapter deletes the packet and creates an event in the target channel of the statechart.

C. Verification

In the case of Maude, we can perform verification using different tools, including reachability analysis, model checking and statistical model checking. Given that the system has an infinite state space, we need to use time-bounded LTL model checking [7] to verify its behavior. Although we could also check timed CTL properties, we chose time-bounded LTL for its simplicity and easier understanding. To specify LTL properties, we first must define the necessary atomic propositions. For example, consider the property (1) specifying that a train never passes the crossing while the barriers are open. To specify this property, we can define the propositions `Train-passing` and `Barriers-open`. As shown in Listing 3, given a satisfaction predefined operator _|=_, we define a proposition by specifying when a state satisfies that proposition. For example, the `Train-passing` proposition is satisfied if there is a token in the `"Train passed"` place of the CPN instance object (lines 5-7). Similarly, the

`Barriers-open` proposition is satisfied if the statechart is in the state `"Barrier open"` (lines 11-13).

```
1  mod PREDICATES is
2    inc BROKER-EX .
3    pr SATISFACTION .
4    --- Sensor system propositions
5    op Train-passing : -> Prop .
6    ceq { < Id : CPNInstance |
7          marking :
8              (token("Train passed", data(Int), T)) >
9          C, GT } |= Train-passing = true
10   if GT ge T .
11   --- Barrier system propositions
12   op Barriers-open : -> Prop .
13   eq { < Id : SCInstance |
14        state : scToken("Barrier open", T) >
15        C, GT } |= Barriers-open = true .
16   --- Otherwise, propositions are false
17   var S : System .
18   var P : Prop .
19   eq S |= P = false [owise] .
20 endm
```

Listing 3. Atomic Propositions in Maude [38]

Given the propositions in Listing 3, we can then verify the desired Property (1) with the following command.

```
red modelCheck(system,
               []~(Barriers-open /\ Train-passing)) .
result Bool: true
```

Property (2) can be encoded similarly in Maude by defining the missing proposition `Barriers-closed` accordingly. Both properties are fulfilled.

To use the model checker, in addition to the executability of the specification (i.e., the equational part of the specification must be terminating and Church-Rosser, and equations and rules must be coherent), the reachable state space must be finite. In this case, the search space is finite only if we limit the global time. The upper time bound for the verification is defined in the Maude rule that advances the global time in the system; it is not part of the property specification.

Atomic propositions can be defined in different Maude modules and then later combined into a system property as shown Listing 3. By instantiating single models, adjusting bindings, or stubbing models, one can verify not only system properties but also properties for each individual model or a certain subset of the system model.

VI. Conclusion & Future Work

We introduced a *non-intrusive* coordination framework that can coordinate system parts described in *heterogeneous* behavioral modeling languages into a *global system* and allows for formal analysis of the system's properties. The framework uses language adapters and a broker to integrate the heterogeneous languages, allowing for *data exchange* and *real-time capabilities* while upholding the *separation* of the languages and models. Furthermore, we implemented our framework using rewriting logic in Maude and used it to verify the correctness of an active level crossing system modeled with Colored Petri Nets and statecharts. Additionally, the framework provides a general methodology for coordination, making it independent of any particular formalism for implementation; that is, the coordination DSL will ultimately hide the details of the underlying implementations. It can be applied to software systems with real-time capabilities, i.e., continuous time that can be approximated or discretized by time-sampling in Maude.

As future research directions, we plan to improve our approach to address the following observations. The coordination DSL supports only binding instance channels; with user-friendliness in mind, it should also include syntax and constructs to support the specification of properties which the system configuration should satisfy—while hiding the underlying implementation. Currently, our implementation in Maude is only applied to one specific use case, and further enhancements are needed to support all possible CPN and statechart features. Nevertheless, it effectively demonstrates the overall architecture proposed in our approach and the proposed patterns for integrating various behavioral languages into our framework. Following the same architecture, we investigated additional behavioral languages (BPMN, LTS) and synchronous communication besides asynchronous communication (see [38]). Furthermore, the broker must not be a single, centralized component as it is currently presented since multiple message brokers exist in a realistic distributed system. In addition, we want to make bindings multi-ary to model advanced communication patterns directly, such as broadcasting and publish/subscribe. Currently, such patterns must be implemented in a dedicated model that duplicates messages together with multiple corresponding bindings.

References

[1] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Cham: Springer International Publishing, 2018.

[2] T. Kräuter, H. König, A. Rutle, Y. Lamo, and P. Stünkel, "Behavioral consistency in multi-modeling." *The Journal of Object Technology*, vol. 22, no. 2, p. 2:1, 2023.

[3] M. Vara Larsen, "BCOol : The behavioral coordination operator language," Ph.D. dissertation, Université Nice Sophia Antipolis, Apr. 2016.

[4] M. E. Vara Larsen, J. Deantoni, B. Combemale, and F. Mallet, "A Behavioral Coordination Operator Language (BCOoL)," in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ottawa, ON, Canada: IEEE, Sep. 2015, pp. 186–195.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude - A High-Performance Logical Framework*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, vol. 4350.

[6] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL Model Checker," *Electronic Notes in Theoretical Computer Science*, vol. 71, pp. 162–187, Apr. 2004.

[7] P. C. Ölveczky, "Real-Time Maude and Its Applications," in *Rewriting Logic and Its Applications*, S. Escobar, Ed. Cham: Springer International Publishing, 2014, vol. 8663, pp. 42–79.

[8] N. Carriero and D. Gelernter, "Linda in context," *Communications of The Acm*, vol. 32, no. 4, pp. 444–458, Apr. 1989.

[9] G. Ciatto, S. Mariani, M. Louvel, A. Omicini, and F. Zambonelli, "Twenty Years of Coordination Technologies: State-of-the-Art and Perspectives," in *Coordination Models and Languages*, G. Di Marzo Serugendo and M. Loreti, Eds. Cham: Springer International Publishing, 2018, vol. 10852, pp. 51–80.

[10] L. J. B. Nixon, E. Simperl, R. Krummenacher, and F. Martin-Recuerda, "Tuplespace-based computing for the Semantic Web: A survey of the state-of-the-art," *The Knowledge Engineering Review*, vol. 23, no. 2, pp. 181–212, Jun. 2008.

[11] D. Rossi, G. Cabri, and E. Denti, "Tuple-based Technologies for Coordination," in *Coordination of Internet Agents*, A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 83–109.

[12] F. Arbab, I. Herman, and P. Spilling, "An overview of manifold and its implementation," *Concurrency: Practice and Experience*, vol. 5, no. 1, pp. 23–70, Feb. 1993.

[13] G. A. Papadopoulos and F. Arbab, "Modelling activities in information systems using the coordination language MANIFOLD," in *Proceedings of the 1998 ACM Symposium on Applied Computing - SAC '98*. Atlanta, Georgia, United States: ACM Press, 1998, pp. 185–193.

[14] F. Arbab, "Reo: A channel-based coordination model for component composition," *Mathematical Structures in Computer Science*, vol. 14, no. 3, pp. 329–366, Jun. 2004.

[15] G. A. Papadopoulos and F. Arbab, "Coordination Models and Languages," in *Advances in Computers*. Elsevier, 1998, vol. 46, pp. 329–400.

[16] P. Clements, "A survey of architecture description languages," in *Proceedings of the 8th International Workshop on Software Specification and Design*. Schloss Velen, Germany: IEEE Comput. Soc. Press, 1996, pp. 16–25.

[17] N. Medvidovic and R. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[18] N. Medvidovic and R. N. Taylor, "A framework for classifying and comparing architecture description languages," in *Software Engineering — ESEC/FSE'97*, G. Goos, J. Hartmanis, J. van Leeuwen, M. Jazayeri, and H. Schauer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, vol. 1301, pp. 60–76.

[19] M. Ozkaya and C. Kloukinas, "Are We There Yet? Analyzing Architecture Description Languages for Formal Analysis, Usability, and Realizability," in *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*. Santander, Spain: IEEE, Sep. 2013, pp. 177–184.

[20] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, "Co-Simulation: A Survey," *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–33, May 2019.

[21] Modelisar, "Functional Mock-up Interface Specification," https://fmi-standard.org/docs/3.0.1/, Jul. 2023.

[22] J. Dahmann, "High level architecture for simulation," in *Proceedings First International Workshop on Distributed Interactive Simulation and Real Time Applications*, 1997, pp. 9–14.

[23] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation: Using Ptolemy II*, 1st ed. Berkeley, Calif: UC Berkeley EECS Dept, 2014.

[24] European Union Agency for Railways., *Report on Railway Safety and Interoperability in the EU 2024*. LU: Publications Office, 2024.

[25] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[26] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International Conference on Software Engineering*. Los Angeles California USA: ACM, May 1999.

[27] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, ser. The Addison-Wesley Signature Series. Boston: Addison-Wesley, 2004.

[28] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1st ed. USA: Prentice Hall Press, 2017.

[29] C. Atkinson and T. Kühne, "Rearchitecting the UML infrastructure," *ACM Transactions on Modeling and Computer Simulation*, vol. 12, no. 4, pp. 290–321, Oct. 2002.

[30] J. de Lara and E. Guerra, "Generic Meta-modelling with Concepts, Templates and Mixin Layers," in *Model Driven Engineering Languages and Systems*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6394, pp. 16–30.

[31] Object Management Group, "Unified Modeling Language, Version 2.5.1," https://www.omg.org/spec/UML, Dec. 2017.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[33] M. Cheatham and C. Pesquita, "Semantic Data Integration," in *Handbook of Big Data Technologies*, A. Y. Zomaya and S. Sakr, Eds. Cham: Springer International Publishing, 2017, pp. 263–305.

[34] F. Macías, U. Wolter, A. Rutle, F. Durán, and R. Rodriguez-Echeverria, "Multilevel coupled model transformations for precise and reusable definition of model behaviour," *Journal of Logical and Algebraic Methods in Programming*, vol. 106, pp. 167–195, Aug. 2019.

[35] J. Sánchez Cuadrado, E. Guerra, and J. De Lara, "Generic Model Transformations: Write Once, Reuse Everywhere," in *Theory and Practice of Model Transformations*, J. Cabot and E. Visser, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6707, pp. 62–77.

[36] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy, "Henshin: A Usability-Focused Framework for EMF Model Transformation Development," in *Graph Transformation*, J. De Lara and D. Plump, Eds. Cham: Springer International Publishing, 2017, vol. 10373, pp. 196–208.

[37] T. Kräuter, A. Rutle, H. König, and Y. Lamo, "A higher-order transformation approach to the formalization and analysis of BPMN using graph transformation systems," *Logical Methods in Computer Science*, vol. Volume 20, Issue 4, p. 12533, Oct. 2024.

[38] No Author Given, "Artifacts MODELS 2025," https://github.com/AnonymousUser198769/MODELS2025-Artifacts, Mar. 2025.

[39] Object Management Group, "Business Process Model and Notation (BPMN), Version 2.0.2," https://www.omg.org/spec/BPMN/, Dec. 2013.

[40] N. Martí-Oliet and J. Meseguer, "Rewriting Logic as a Logical and Semantic Framework," *Electronic Notes in Theoretical Computer Science*, vol. 4, pp. 190–225, 1996.

[41] J. Meseguer and G. Roşu, "The rewriting logic semantics project: A progress report," *Information and Computation*, vol. 231, pp. 38–69, Oct. 2013.

PAPER E

BPMN ANALYZER 2.0: INSTANTANEOUS, COMPREHENSIBLE, AND FIXABLE CONTROL FLOW ANALYSIS FOR REALISTIC BPMN MODELS

Tim Kräuter, Patrick Stünkel, Adrian Rutle, Yngve Lamo, Harald König

Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Forum at BPM 2024 co-located with 22nd International Conference on Business Process Management (BPM 2024), 2024, https://ceur-ws.org/Vol-3758/paper-11.pdf

BPMN Analyzer 2.0: Instantaneous, Comprehensible, and Fixable Control Flow Analysis for Realistic BPMN Models

Tim Kräuter[1], Patrick Stünkel[1], Adrian Rutle[1], Yngve Lamo[1] and Harald König[2,1]

[1]Western Norway University of Applied Sciences, Bergen, Norway
[2]FHDW Hannover, Germany

Abstract

Many business process models contain control flow errors, such as deadlocks or livelocks, which hinder proper execution. In this paper, we introduce a new tool that can instantaneously identify control flow errors in BPMN models, make them understandable for modelers, and suggest corrections to resolve them. We demonstrate that detection is instantaneous by benchmarking our tool against synthetic BPMN models with increasing size and state space complexity, as well as realistic models. Moreover, the tool directly displays detected errors in the model, including an interactive visualization, and suggests fixes to resolve them. The tool is open source, extensible, and integrated into a popular BPMN modeling tool.

Keywords

BPM, Verification, Control flow analysis, BPMN model checking, Soundness, Safeness

1. Introduction

Business Process Modeling Notation (BPMN) is becoming increasingly popular for automating processes and orchestrating people and systems. However, many process models suffer from control flow errors, such as deadlocks, livelocks, and starvation [1]. These errors hinder the correct execution of BPMN models and may be detected late in the development process, resulting in elevated costs.

In this paper, we describe a new tool, the *BPMN Analyzer 2.0*[1], for analyzing BPMN process models to detect control flow errors *already* during modeling. Figure 1 shows an overview of the tool. The UI is based on the popular *bpmn.io* ecosystem, while the analysis is implemented in *Rust* for optimal performance and memory efficiency. We perform a breadth-first state space exploration to check soundness and safeness [2] *on the fly* to uncover control flow errors. Consequently, the tool can detect deadlocks, livelocks, starvation, dead activities, and lack of synchronization in BPMN models. The BPMN Analyzer is open source and accessible online alongside a video demonstration[2] [3].

Proceedings of the Best BPM Dissertation Award, Doctoral Consortium, and Demonstrations & Resources Forum co-located with 22nd International Conference on Business Process Management (BPM 2024), Krakow, Poland, September 1st to 6th, 2024.

tkra@hvl.no (T. Kräuter); past@hvl.no (P. Stünkel); aru@hvl.no (A. Rutle); yla@hvl.no (Y. Lamo); harald.koenig@fhdw.de (H. König)

[1]In the following, we will use BPMN Analyzer to refer to the BPMN Analyzer 2.0.
[2]Tool: https://timkraeuter.com/bpmn-analyzer-js/, Video: https://www.youtube.com/watch?v=Nv2W-hXNZYA

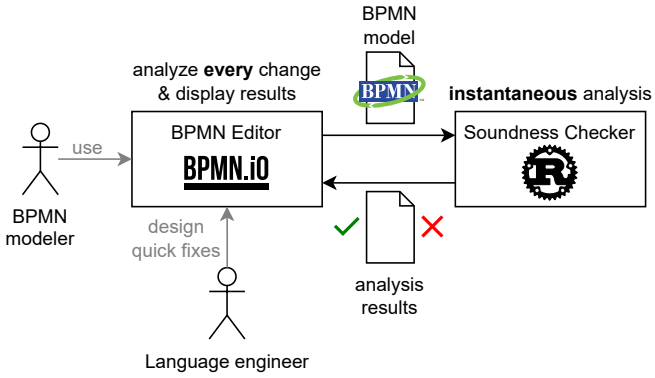


Figure 1: Overview of the BPMN Analyzer 2.0

The tool can check models after each change since analysis is *instantaneous* according to [1], i.e., it takes 500ms or less. Furthermore, we ensure the results are *comprehensible* by highlighting possible violations directly in the model and displaying an interactive counterexample visualization. Finally, the tool suggests *fixes* for the most common control flow errors and can be extended to suggest more fixes in the future.

Fahland et al. [1] describe *coverage*, *immediacy*, and *consumability* as the main challenges for users unaccustomed to formal analysis. The BPMN Analyzer addresses all these challenges since it supports the most common BPMN elements used in practice (coverage), provides *instantaneous* results (immediacy), and a *comprehensible* user interface (consumability), even including suggestions for fixes. Developers of industrial BPMN software also like our tool, especially the End-to-end user journey [3]. Thus, this supports our claim that the UI is understandable for users unfamiliar with formal analysis.

In the remainder of the paper, we describe how instantaneous, comprehensible, and fixable control flow error detection is achieved in section 2. Then, we discuss tool maturity in section 3 before concluding in section 4.

2. Innovations

The BPMN Analyzer has three main innovations: **instantaneous**, **comprehensible**, and **fixable** control flow error detection. In this section, we will present the innovations, and more details can be found in our extended paper [3].

2.1. Instantaneous Analysis

We demonstrate instantaneous control flow analysis by benchmarking our tool in *three* scenarios. For all our benchmarks, we use the hyperfine benchmarking tool (version 1.18.0), which calculates the average runtime when executing each control flow analysis ten or more times. We ran the benchmarks on Ubuntu 22.04.4 with an AMD Ryzen 7700X processor (4.5GHz) and 32 GB of RAM (5600 MHz). All used BPMN models, our tools to generate them, and benchmarking scripts to run them are available in [3].

First, we benchmarked how our tool handles **BPMN models of growing size**. We generated 500 synthetic BPMN models starting with five elements up to 4000. The models repeatedly contain three activities and an exclusive/parallel block with two branches containing one activity per branch (see [3]). The BPMN Analyzer spends from 1 ms up to 9 ms for the BPMN models [3] compared to 0.7 s up to 14 s in our previous tool [4]. In summary, the runtime grows linearly with the state space.

Second, we benchmarked the tool against a synthetic data set of models that led to a state space explosion. This represents a *worst-case* scenario for formal analysis. We generated a data set of models [3] with a growing number of parallel branches with increasing length, like [5].

Table 1 shows the average runtime of our tool when analyzing these models. The BPMN Analyzer explores the entire state space while simultaneously analyzing the control flow, i.e., verifying soundness properties. The models' state space grows exponentially, leading to the same order of growth in runtime. Our analysis is not instantaneous anymore when approaching 17 parallel branches of length 1 (see Table 1). However, analysis is still instantaneous for more reasonable models with five parallel branches of length 5 or 3 branches of length 20. Other tools report 2-3s of runtime for most soundness properties and 30s for a model with five parallel branches [5], which took milliseconds in our tool.

Table 1
Benchmark results of the parallel branches models

Branches	Branch Length	Runtime	States
5	1	1 ms	35
10	1	3 ms	1.027
15	1	161 ms	32.771
16	1	360 ms	65.539
17	1	790 ms	131.075
20	1	8.803 ms	1.048.579
5	5	14 ms	7.779
3	20	11 ms	9.264

Table 2
Benchmark results of the realistic BPMN models

Model name	Runtime	States
e001 [6]	1 ms	39
e002 [6]	1 ms	39
e020 [6]	10 ms	5356
credit-scoring-async[3]	1 ms	60
credit-scoring-sync[3]	1 ms	140
dispatch-of-goods[3]	1 ms	103
recourse[3]	1 ms	77
self-service-restaurant[3]	1 ms	190

Third, we applied our tool to eight **realistic models**, where three models (e001, e002, e020) are taken from [6], and the remaining five models are part of the Camunda BPMN for research repository[3]. Table 2 shows each model's average runtime and number of states. The BPMN Analyzer takes 1-10ms for e001, e002, and e020 [3], while [6] and [4] report 3.66-10.26s and 1-1.75s respectively. The benchmarks in [4] were run on the same hardware, while the machine used in [6] was less powerful. Our analysis is instantaneous for nearly all BPMN models since most have less than 1000 states, according to [1].

[3] https://github.com/camunda/bpmn-for-research

2.2. Comprehensible Analysis

We implemented two features to make control flow analysis understandable for everyone. **First**, we highlight the problematic elements that cause control flow errors by directly attaching red overlays to them in the model. In addition, there is a summary panel in the top-right stating if any errors are found.

Second, we use *tokens* to visualize errors *interactively*, i.e., show an execution leading to the error. Our analysis provides sample executions resulting in the found control-flow errors, which we visualize in the editor by showing how tokens move from the process start to an erroneous state. We are unaware of other tools that visualize errors directly and allow interactions, such as stopping/resuming and restarting.

In Figure 2, the visualization has been *paused* just before an *unsafe* state was reached. One token is already located at the marked sequence flow, while a second token is currently waiting at the exclusive gateway e1. The visualization can be resumed or restarted using the play and restart button on the left side. The gateway e1 will execute when resumed, resulting in two tokens at the subsequent sequence flow, i.e., an unsafe execution state. Furthermore, one can control the visualization speed using the bottom buttons next to the speedometer.

2.3. Fixable Analysis

Besides detecting, highlighting, and visualizing control flow errors, the BPMN Analyzer suggests fixes like *quick fixes* in IDEs. Quick fixes cannot be provided for all errors, but we currently cover many patterns leading to deadlocks, lack of synchronization, message starvation, and reused end events. The quick fixes we support are described in detail in [3] and can be extended independently of the formal analysis. We are unaware of other tools that can fix identified control-flow errors.

For example, Figure 2 shows a screenshot of our tool, where quick fixes are depicted as green overlays containing a light bulb icon. A user can apply a quick fix by clicking on a green overlay and instantly see the changes regarding control flow errors. If unhappy with the result, a user can undo all changes since each quick fix is entirely revertible due to the command pattern. A user might not like a quick fix if it not only fixes an error but also has unintended side effects, such as introducing a different control flow error.

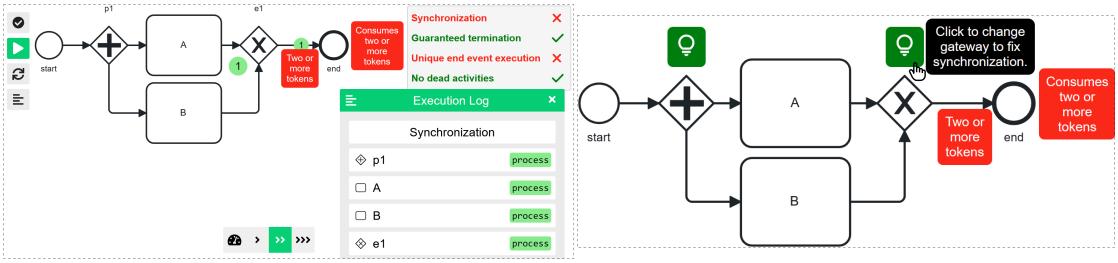


Figure 2: Execution visualization (left) and suggested *quick fixes* (right) in the BPMN Analyzer

3. Maturity of the Tool

The BPMN Analyzer incorporates many findings from our previous work [4] while focusing on instantaneous and understandable error detection, as described in the previous section. The tool is open source [3], and we ensure high code quality by employing industry best practices such as rigorous static analysis and testing. Furthermore, we received positive feedback from companies in the BPMN process orchestration space [3].

4. Conclusion & Future Work

In this paper, we describe the novel *BPMN Analyzer* that provides instantaneous, comprehensible, and fixable BPMN control flow error detection and is integrated into a popular BPMN modeling tool. We benchmarked our tool against synthetic and realistic BPMN models to demonstrate instantaneous soundness checking. We address the three main challenges, *coverage*, *immediacy*, and *consumability*, to provide formal analysis to non-expert users as identified in [1]. In addition, our tool offers quick fixes for common patterns that lead to control flow errors. One can understand the BPMN Analyzer as a BPMN-specific model checker, implemented in Rust paired with an intuitive user interface based on the popular *bpmn.io* ecosystem that is open for extension by design.

In future work, we aim to improve our tool by providing more quick fixes, considering advanced BPMN elements such as different events, and ranking quick fixes based on usefulness and previous user behavior. Finally, we aspire to test our tool in a real-world scenario to gather feedback and measure its impact on productivity.

References

[1] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, K. Wolf, Analysis on demand: Instantaneous soundness checking of industrial business process models, Data & Knowledge Engineering 70 (2011) 448–466. doi:10.1016/j.datak.2011.01.004.

[2] F. Corradini, C. Muzi, B. Re, F. Tiezzi, A Classification of BPMN Collaborations based on Safeness and Soundness Notions, Electronic Proceedings in Theoretical Computer Science 276 (2018) 37–52. doi:10.4204/EPTCS.276.5.

[3] T. Kräuter, P. Stünkel, A. Rutle, H. König, Y. Lamo, Instantaneous, Comprehensible, and Fixable Soundness Checking of Realistic BPMN Models, 2024. arXiv:2407.03965.

[4] T. Kräuter, A. Rutle, H. König, Y. Lamo, A higher-order transformation approach to the formalization and analysis of BPMN using graph transformation systems, 2024. arXiv:2311.05243.

[5] F. Corradini, F. Fornari, A. Polini, B. Re, F. Tiezzi, A. Vandin, A formal approach for the analysis of BPMN collaboration models, Journal of Systems and Software 180 (2021) 111007. doi:10.1016/j.jss.2021.111007.

[6] S. Houhou, S. Baarir, P. Poizat, P. Quéinnec, L. Kahloul, A First-Order Logic verification framework for communication-parametric and time-aware BPMN collaborations, Information Systems 104 (2022) 101765. doi:10.1016/j.is.2021.101765.